HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

COMPASS SECURITY

# Emil on Steroids

# Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2018


| | |
|---|---|
| Autoren: | Patrick Steinhäusl |
| | Sven Defatsch |
| Betreuer: | Cyrill Brunschwiler |
| Projektpartner: | Compass Security Schweiz AG |
| Experte: | Dr. Christian Folini |
| Gegenleser: | Prof. Stefan F. Keller |

# Inhaltsverzeichnis

# 1 Abstract

Die HSR nutzt das Hacking-Lab[1], eine Plattform mit diversen Security Challenges, für die praxisbezogene Ausbildung im Themengebiet der Informationssicherheit (Modul InfSi3, Challenge Projekt und CAS Frontend Engineering). Eine für mehrere Challenges genutzte Applikation ist der sogenannte "Glockenshop". Dieser fiktive Webshop beinhaltet verschiedene Sicherheitslücken, welche in diversen Aufgaben gefunden und ausgenutzt werden müssen. Zudem sind die Studenten angehalten, Vorschläge für die Vermeidung der Fehler vorzuschlagen.

Der in die Jahre gekommene Webshop wurde im Rahmen einer Studienarbeit neu aufgebaut. Der eingesetzte Technologie Stack basiert auf MongoDB, Express, AngularJS und Node.js, auch "MEAN"[2] Stack genannt. Mit der aktuellen Arbeit wurde dieses Grundgerüst überarbeitet, Fehler behoben und die Anwenderfreundlichkeit verbessert. Des Weiteren wurden sehr viele Sicherheitslücken in die Applikation eingebaut. Die gewählten Schwachstellen gehören zu den aktuellsten und häufigsten (OWASP Top 10)[3], um möglichst realistische Szenarien darzustellen. Nebst den bekannten Angriffsvektoren "Cross-Site Scripting (XSS)", "NoSQL Injection" oder "Remote Code Execution (RCE)" sind auch weniger geläufige wie "Cross-Site WebSocket Hijacking (CSWSH)" und De-Serialisierungs-Probleme implementiert worden.

Als Voraussetzung galt die Auslieferung der Anwendung als fertiges Docker Image. Dies garantiert die Lauffähigkeit auf Fremdsystemen und die reibungslose Integration in die bestehende Umgebung des Hacking-Labs. Mittels Docker kann jeder Benutzer seine private Instanz der Applikation starten. Das Ausnutzen der Schwachstellen hat somit keine Seiteneffekte für andere Anwender der Plattform. Der zweite grosse Aspekt der Arbeit war die Formulierung der Aufgabenstellungen. Für jede Lücke musste ein realistisches Szenario ausgedacht werden, für welches wiederum eine Aufgabenstellung inklusive Musterlösung erstellt wurde. Diese Aufgaben gilt es im Rahmen der praktischen Übungen zu lösen.

# 2 Management Summary

## 2.1 Ausgangslage

Aktuell bietet das Hacking-Lab für die Schulung seiner Benutzer verschiedene Anwendungen. Eine Applikation davon ist der sogenannte "Glockenshop", welcher als Beispiel für die verschiedenen Angriffsszenarien dient. Mittlerweile erscheint diese Webapplikation allerdings ein wenig angestaubt und basiert nicht auf Webtechnologien, welche im Jahre 2018 als modern gelten.
Aus diesem Grund wurde entscheiden, dass ein neuer "Glockenshop" erstellt wird, welcher auch wieder mit diversen Sicherheitslücken für das Training ausgestattet ist, aber neuere Technologien verwendet. Die erste Version davon entstand während einer Studienarbeit[4] und als Aufgabe einer Bachelorarbeit stand nun die Weiterentwicklung eben dieser als Ziel.

## 2.2 Vorgehen und Technologien

Als erstes musste mittels einer Analyse festgestellt werden, was alles an der ersten Version verbessert oder erweitert werden muss. Dafür wurde der Shop ausgiebig getestet und die entsprechend nötigen Änderungen umgesetzt.
Ein weiterer Punkt bestand in der Planung von möglichen neuen Schwachstellen und deren Angriffsszenarien. Das Ergebnis dieser Überlegungen floss zum einen in die Aufgabenstellungen und zum anderen in den geschriebenen Code ein.
Für die Umsetzung selbst war es wichtig, auf aktuelle Webtechnologien zu setzen, um der realen Welt näher zu sein. Dazu gehören auch die Einhaltung von "Best-Practices" in der Webentwicklung im Allgemeinen und im Aufbau des Codes und dessen Struktur.

## 2.3 Ergebnisse

Während der Arbeit entstand ein Webshop mit implementierten Schwachstellen, welcher zur Auslieferung auf beliebigen System bereit steht und somit einfach in die bestehende Hacking-Lab Umgebung eingebunden werden kann.
Für alle Lücken im System wurden Aufgabenstellungen geschrieben, welche ein Nutzer lösen kann. Sollte ein Benutzer dabei nicht mehr weiterkommen kann er auf ebenfalls bereitgestellte Musterlösungen zurückgreifen, welche detailliert beschrieben sind. Kurz gesagt wurde mit dem neuen "Glockenshop" eine Vorlage für viele weitere mögliche Lücken für das Training geschaffen. Somit stehen auch für zukünftige Challenges alle Tore offen.

## 2.4 Ausblick

In Zukunft könnte der Webshop mit diversen Ansätzen verbessert werden:

- Das Design kann überarbeitet, sowie die Usability verbessert werden.
- Eingesetzte Technologien können konstant in der Version aktualisiert werden.
- Aktuelle Schwachstellen können nach belieben ergänzt werden.
- Die Testumgebung und Testqualität kann noch erhöht werden.
- Dem Shop können mehr Testdaten hinterlegt werden, sodass es realistischer wirkt.
- Es können weitere Technologien eingebaut und somit Technologie-spezifische Lücken umgesetzt werden.

# 3 Einleitung

Dieser Abschnitt bietet eine Übersicht über das gesamte Projekt, dessen Ziele und die Motivation hinter der Aufgabe.

## 3.1 Bisherige Lösung

Das Hacking-Lab betreibt für seine Benutzer diverse Security Challenges auf einer Online Schulungsplattform. Mehrere dieser Challenges basieren auf dem sogenannten "Glocken Emil Shop".
Die bisherige Lösung des "Glockenshops" ist eine schlichte Webanwendung, welche bezüglich des Designs und der eingesetzten Technologien ein wenig in die Jahre gekommen ist. Auf der Plattform sind verschiedene Schwachstellen implementiert, welche ein Hacking-Lab Nutzer anhand einer Aufgabenstellung finden und gezielt ausnutzen muss. Ebenfalls muss der Nutzer sein Vorgehen und Vorschläge für die Behebung der Sicherheitsrisiken abgeben. Die Schwachstellen basieren zum Teil auf Lücken in den eingesetzten Technologien, oder aber auch generell auf Problemen, welche auf der OWASP Top 10[3] Liste zu finden sind.

## 3.2 Vorgängerarbeit

Im Zuge einer Studienarbeit wurde bereits an der Aufgabenstellung gearbeitet. Als Ergebnis ist eine erste Version des neuen Webshops mit 6 implementierten Schwachstellen entstanden. Als Basis für den Shop wurde konsequent auf moderne Webtechnologien gesetzt, um einige Schwächen gegenüber der bestehenden Lösung auszugleichen und allenfalls aktuelle, technologiebasierte Lücken einzubauen.
Details bezüglich des Source Codes, welcher während der Vorarbeit entstanden ist, können auf GitHub.com eingesehen werden: https://github.com/patricksteinhaeusl/Studienarbeit

## 3.3 Ziele und Umfang dieser Arbeit

Das primäre Ziel der Arbeit ist es, die Vorgängerarbeit gemäss des in der Aufgabenstellung vorgegebenen Umfangs weiterzuführen. Dazu gehört sowohl die Verbesserung bekannter Probleme der Vorgängerarbeit, wie auch das einarbeiten in neue Schwachstellen und deren Umsetzung als Code. Schlussendlich werden zu den neuen und bereits vorher implementierten Schwachstellen Aufgaben und Musterlösungen erstellt. Des weiteren soll der Webshop selbst als fertiges Docker Image ausgeliefert werden. Das fertige Produkt mit allen Aufgaben soll am Ende Im Hacking-Lab wieder der Schulung der Nutzer dienen.

# 4 Analyse

In diesem Abschnitt wird thematisiert, was schon in der Vorgängerarbeit erledigt wurde und was die Grundvoraussetzungen für diese Arbeit sind. Zum Schluss werden die Ergebnisse aus der Analyse zusammengefasst, um eine Übersicht über die zu erledigenden Aufgaben zu erhalten.

## 4.1 Ausgangslage und Technologien

Als Ausgangslage dient die in der Einleitung erwähnte Vorgängerarbeit, auf welche direkt aufgebaut wird. Das Projekt basiert auf folgenden drei Technologien:

- **AngularJS 1.6** im Frontend für die Benutzerinteraktion
- **Node.js 8.9 LTS** mit Express.js im Backend für die Businesslogik
- **MongoDB** als NoSQL Datenbank für die Datenhaltung des Shops

Die eingesetzten Technologien werden beibehalten, wo nötig und sinnvoll jedoch durch weitere ergänzt.

## 4.2 Architektur

In diesem Abschnitt wird die Architektur des bisherigen Webshops, sowie die des aus der Vorgängerarbeit entstandenen Webshops, analysiert.

### 4.2.1 Bisheriger Webshop

Die Architektur des alten Webshops unterscheidet sich im Vergleich zur Vorgängerarbeit massiv. Die eingesetzten Technologien sind aus heutiger Sicht veraltet und auch Verwundbarkeiten, welche aufgrund neuer Technologien entstehen, sind kaum zu implementieren.

**Deployment** Das System besitzt einen Apache Webserver mit integriertem Single-Sign-On Modul mit dem Namen mod_auth_tkt. Das Modul übernimmt in Zusammenarbeit mit einer MySQL Datenbank das Session Handling der ganzen Applikation. Hinter dem Apache Server befindet sich ein Tomcat Webserver mit einer Servlet API. Nicht authentifizierte Benutzer werden automatisch auf die Login Seite weitergeleitet. Die Authentifizierung am System erfolgt über eine MySQL Datenbank, welche die Benutzer der Shop Applikation beinhaltet, mit einem zusätzlichen Abgleich des Benutzers mittels LDAP. Nach erfolgreichen Authentifizierung kann dieser Benutzer auf die Applikation zugreifen. Die Daten der Webapplikation werden wiederum in einer MySQL Datenbank gehalten. [6]

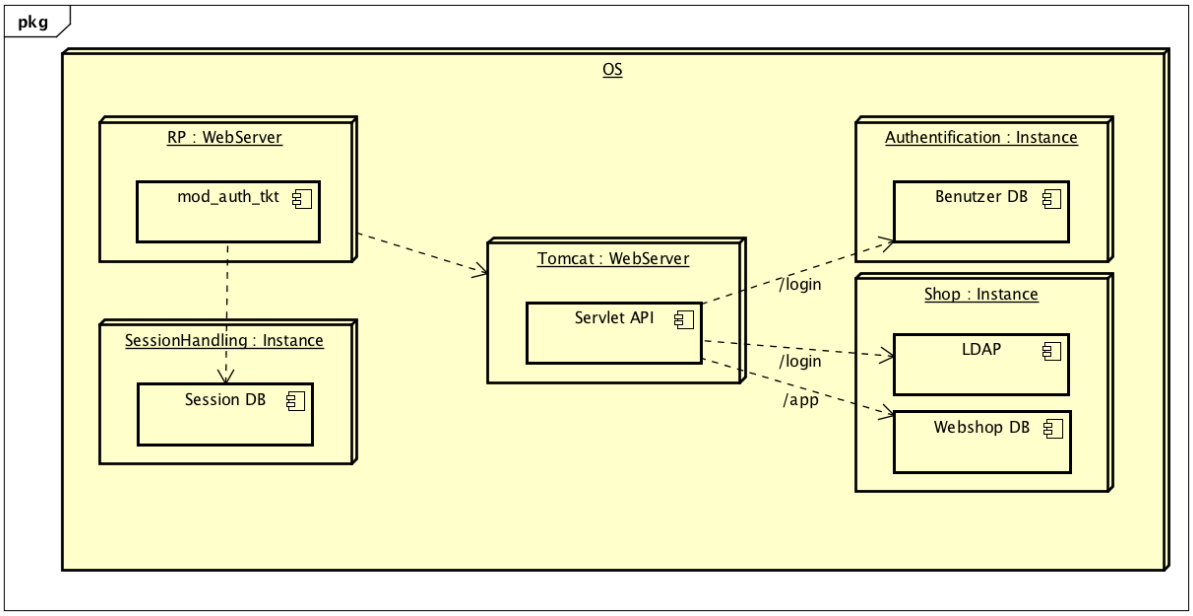


Abbildung 4.1: Deployment Diagramm des bisherigen Webshops[6]

Die Applikation musste zwingend überarbeitet werden, sodass auch Schwachstellen, welche auf neuen Technologien basieren, implementiert werden konnten.

**GUI** Die Darstellung der Benutzeroberfläche des bisherigen Webshops ist schlicht und entspricht nicht mehr den heutigen Standards in Usability und Design. Zudem werden in dieser Lösung die einzelnen Seiten als Frames geladen. Dies führt dazu, dass bei Aktionen des Benutzers die Seiten immer wieder neu geladen werden müssen, was wiederum die Usability des Systems beeinträchtigt. [6]

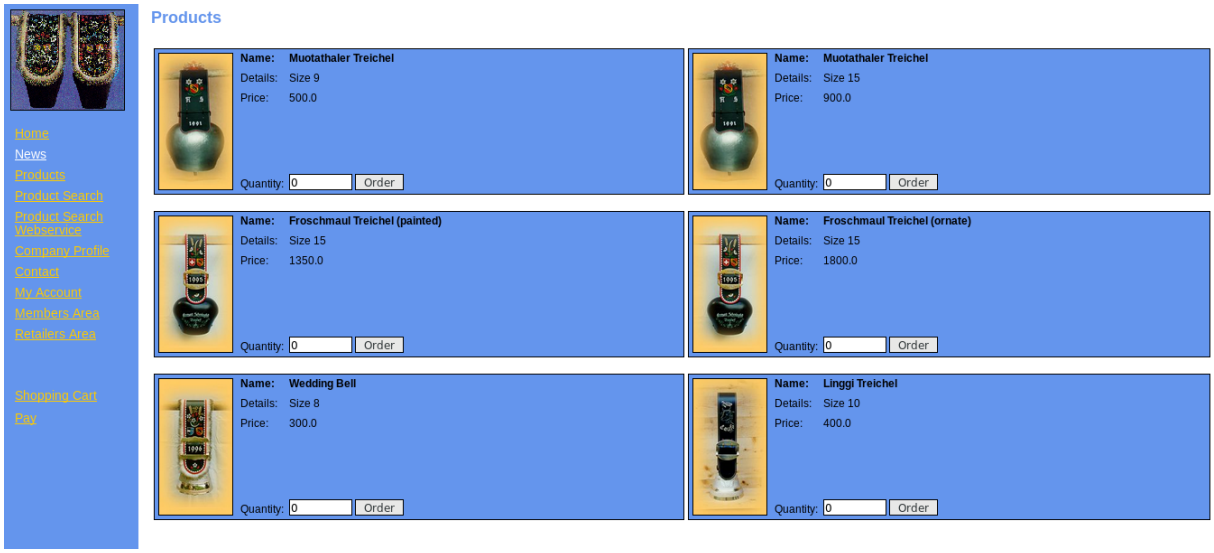


Abbildung 4.2: GUI der bisherigen Lösung[6]

### 4.2.2 Webshop nach Vorgängerarbeit

Der Webshop aus der Vorgängerarbeit wurde bereits mit neuen Technologien umgesetzt was im folgenden Abschnitt genauer beschrieben wird.

**Deployment** Die Applikation der Vorgängerarbeit besitzt bereits ein AngularJS Frontend und ein Node.js / Express.js Backend, welches die Daten in einer MongoDB persistiert. [4]

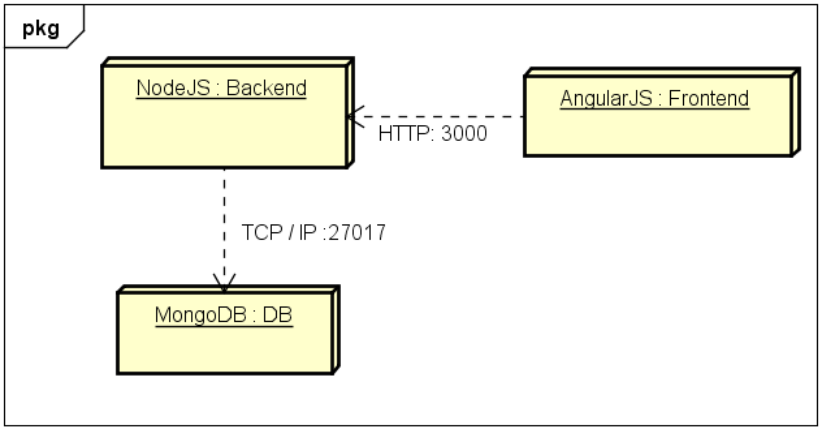


Abbildung 4.3: Deployment Diagramm der Vorgängerarbeit[4]

Die Applikation ist aus technologischer Sicht bereits soweit, dass auf diesem Stand weitergearbeitet werden kann und somit neue Schwachstellen implementiert werden können.

**Domain Modell** Das Domain Modell ähnelt demjenigen der bisherigen Lösung und ist folgendermassen aufgebaut:

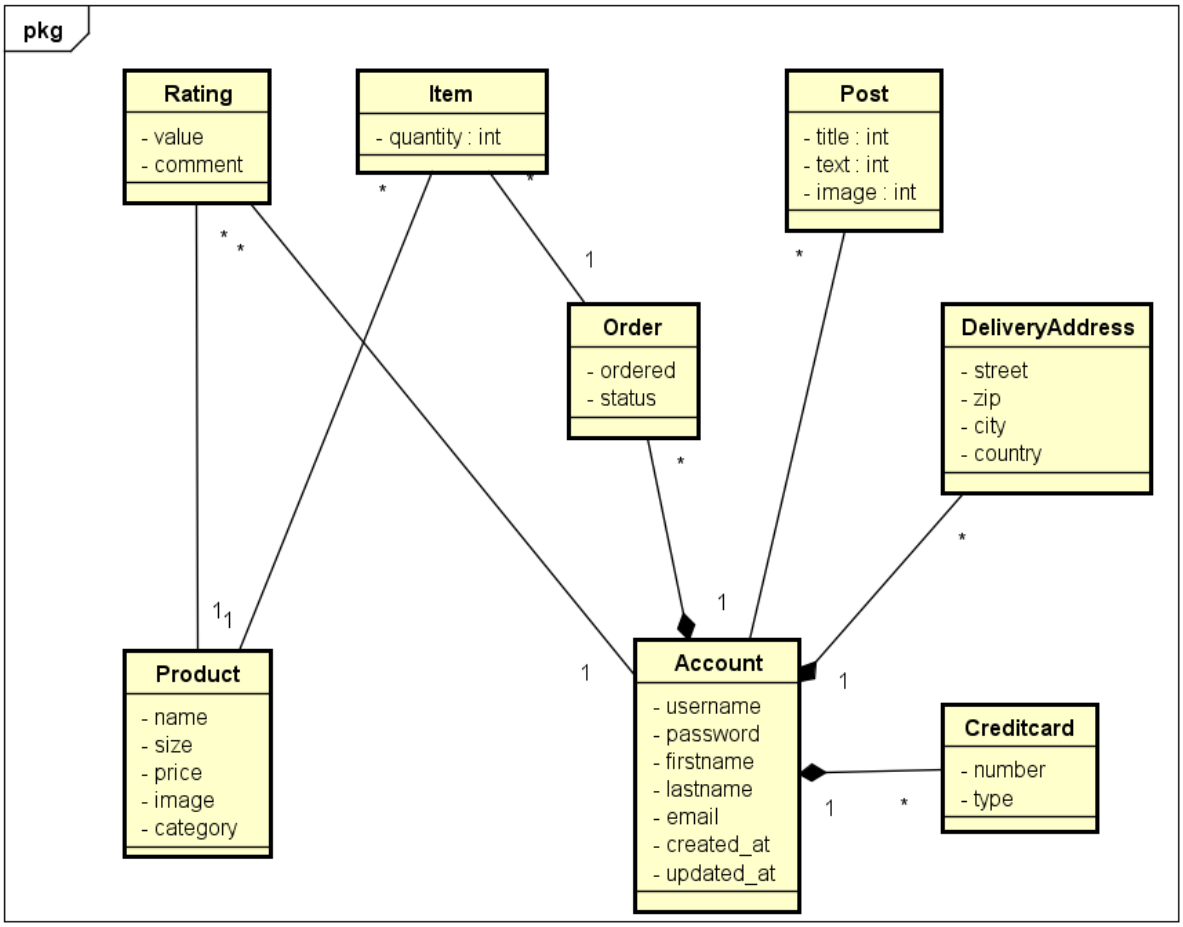


Abbildung 4.4: Domain Modell der Vorgängerarbeit[4]

Das Modell ähnelt in der Komplexität derjenigen der bisherigen Lösung. Es wurde um das Rating und die Benutzerdaten wie Kreditkarte und Lieferadresse erweitert. [4]

**GUI** Das grafische Benutzerinterface wurde einem Facelift unterzogen. Dadurch konnte die Usability bereits verbessert werden.

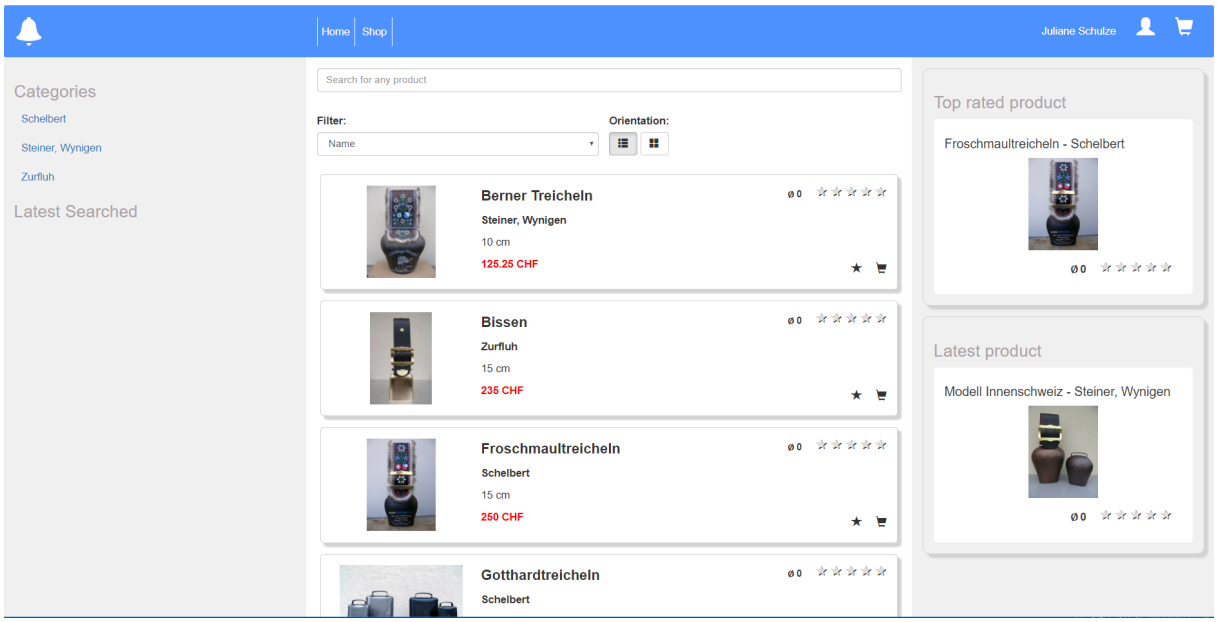


Abbildung 4.5: GUI der Vorgängerarbeit[4]

Durch den Einsatz von AngularJS läuft der Webshop nun als Single Page Applikation. Dies führt dazu, dass die Seite nicht bei jeder Interaktion neu geladen werden muss. Zudem wurde Bootstrap.js eingesetzt, um das Design einheitlich zu halten und zugleich mit einem Responsive Design auszustatten. [4]

## 4.3 Funktionale Anforderungen

Bei den funktionalen Anforderungen gilt es zwischen zwei Bereichen zu unterscheiden. Zum einen gibt es den Webshop, welcher normal wie jeder andere Shop funktionieren muss, zum anderen den Aspekt des Hackers, welcher versucht, Schwachstellen auszunutzen. Aus diesem Grund werden die Use Cases zwischen einem normalen Shop Benutzer und einem Hacker aufgeteilt. Ein Argument dies nicht zu tun wäre, dass der Hacker die Use Cases des normalen Anwenders ebenfalls nutzen kann und somit eine Aufteilung überfällig ist. Die Use Cases des normalen Nutzers liegen aber ausserhalb des Interessenbereichs des Aktors "Hacker" und daher ist eine Trennung sinnvoll.
Die detaillierten Use Case Beschreibungen im "Fully dressed" Format inklusive Use Case Diagramme sind im Anhang in Abschnitt *A Use Cases* zu finden.

### 4.3.1 Use Cases - Webshop

In diese Abschnitt wird auf die Use Cases, welche im Webshop enthalten sind, eingegangen.

**UC01: Benutzer erstellen** Ein nicht registrierter Benutzer kann sich mit einer Email Adresse, Passwort und seinen persönlichen Angaben einen neuen Benutzer erstellen.

**UC02: Benutzerdaten bearbeiten** Ein authentifizierter Benutzer kann seine persönlichen Daten anpassen. Löschen ist nicht möglich.

**UC03: Öffentliches Profil hochladen** Jeder authentifizierte Benutzer kann unter seinen Benutzerdaten ein öffentliches Profil im HTML Format hochladen, welches von anderen Benutzern betrachtet werden kann.

**UC04: Kreditkarte bearbeiten** Jeder authentifizierte Benutzer kann bis zu drei Kreditkarten hinzufügen und diese auch wieder ändern oder löschen. Mindestens eine Kreditkarte muss nach Erstellung zwingend bestehen bleiben.

**UC05: Lieferadresse bearbeiten** Jeder authentifizierte Benutzer kann bis zu drei Lieferadressen hinzufügen und diese auch wieder ändern oder löschen. Mindestens eine Lieferadresse muss nach Erstellung zwingend bestehen bleiben.

**UC06: Produkt bewerten** Jeder authentifizierte Benutzer kann einem im Webshop enthaltenen Produkt eine Bewertung abgeben.

**UC07: Produkt suchen** Ein Benutzer durchsucht den Webshop nach Produkten und hat dabei die Möglichkeit, seine Auswahl mittels Kategorien einzugrenzen, oder ein Produkt mittels Suchfunktion zu finden.

**UC08: Produkt bestellen** Ein Benutzer oder authentifizierter Benutzer kann im Webshop Produkte dem Warenkorb hinzufügen. Der authentifizierte Benutzer ist nun in der Lage, die Produkte des Warenkorbs zu bestellen. Der Bestellprozess beinhaltet eine Übersicht der bestellten Artikel, danach folgt die Auswahl der Lieferadresse und der Zahlungsmethode. Die Bestellung kann am Ende dieses Prozesses abgeschlossen werden.

**UC09: Frage zu Produkt erstellen** Ein authentifizierter Benutzer kann Fragen zu Produkten erfassen, welche auch von anderen Benutzern eingesehen werden können.

**UC10: Buchungen anzeigen** Ein authentifizierter Benutzer kann auf einer Übersichtsseite seine Buchungen anzeigen lassen. Diese Buchungen können über eine Exportfunktion durch Angabe von 2 Parametern als PDF exportiert werden.

**UC11: Beitrag bearbeiten** Jeder authentifizierte Benutzer kann im Community Bereich des Webshops einen neuen Beitrag erfassen. Der Beitrag muss ein Bild oder ein Link zu einem Bild enthalten. Nach Erfassung kann der Autor diesen Beitrag wieder löschen.

**UC12: Chatten** Die im System authentifizierten Benutzer können untereinander mit einem Chat kommunizieren.

### 4.3.2 Use Cases - Hacking

Wie bereits erklärt, sind die Use Cases des Hackers die Schwachstellen an sich. Der Webshop und seine Funktionalität dienen dem Hacker als Mittel zum Zweck, um Lücken ausnutzen zu können. Weil diese Use Cases nur schlecht im "Brief" oder "Fully dressed" Format abgebildet werden können, wird auf die klassische Darstellung verzichtet und stattdessen in den Abschnitten *4.8.1 Bereits implementierte Schwachstellen* und *4.8.2 Zu implementierende Schwachstellen* generell auf die geplanten Schwachstellen eingegangen.

## 4.4 Problembehebung der Vorgängerarbeit

Zu Beginn wird der aus der Studienarbeit übernommene Webshop auf Herz und Nieren geprüft, um Fehler oder Ungereimtheiten in der Bedienung zu finden. Das Hauptaugenmerk liegt dabei auf der funktionellen Ebene und nicht im visuellen Bereich. Einige Punkte sind bereits bei der Korrektur der Vorgängerarbeit aufgetaucht und werden als erstes nachgebessert. Die wichtigsten Resultate dieses Abschnitts sind im Abschnitt *5.1 Bugfixing der Vorängerarbeit* zu finden.

## 4.5 Benutzeroberfläche

Das Design der Vorgängerarbeit wurde mittels Bootstrap[5] erstellt und ist von der Darstellung an den alten Shop angelehnt. Der Fokus dieser Arbeit liegt nicht in der erneuten Überarbeitung des Designs. Kleinere Mängel oder Feinheiten werden jedoch als normale Bugs behandelt und behoben.

## 4.6 Docker

Entscheidet sich ein Hacking-Lab Benutzer für eine auf dem Glockenshop basierende Aufgabe, wird für jeden Nutzer eine eigene Instanz des Webshops benötigt, damit die Aktionen der verschiedenen Anwender nicht miteinander interferieren.

Um eine solche Vorgabe sicherzustellen, könnte auf virtuelle Maschinen gesetzt werden, was je nach Nutzeraufkommen Ressourcenintensiv werden kann. Eine andere Möglichkeit sind mehrere Instanzen des Shops auf einem Webserver laufen zu lassen. Dies wiederum könnte ein Problem mit der Stabilität nach sich ziehen. Da die Nutzer des Shops dazu angehalten sind aktiv Lücken im System zu finden, ist es im vornherein nie möglich alle Fälle, welche ein Nutzer probieren könnte, abzufangen. Einer dieser nicht angedachten Fälle könnte unter Umständen den Server zum Absturz bringen und damit wären alle anderen Nutzer des Glockenshops betroffen. Aus diesen Gründen wird eine andere Lösung angestrebt.

In den letzten Jahren ist die "Containerisierung" von Applikationen mittels Docker[7] immer beliebter geworden. Ein grosser Vorteil dieser Container ist, dass die ganze Applikation genau wie sie entwickelt worden ist verpackt und dem Endbenutzer ausgeliefert wird. Bis auf das Starten des Containers ist keine Installation der Software nötig.
Im Gegensatz zu einer klassischen Virtualisierungslösung erzeugt diese Methode deutlich weniger Overhead und gilt somit als "Lightweight". Neue Instanzen sind bei Bedarf schnell hochgefahren und können sich dank Isolation voneinander gegenseitig nicht beeinflussen.

## 4.7 Testing

Bis anhin existieren für das gesamte Projekt nur wenige Tests, was die Gefahr für "Breaking-Changes" erhöht. Zudem beinhaltet die bestehende Testing-Lösung zu viele verschiedene Frameworks, welche schlussendlich nicht alle eingesetzt werden. Aus diesen zwei Gründen ergab die Analyse, dass es am sinnvollsten ist, die wenigen vorhandenen Tests und die Frameworks zu löschen und eine neue, saubere Testumgebung mit End-to-End (E2E) Tests aufzubauen. E2E Tests bieten gegenüber Unit Tests den Vorteil, nebst den normalen Shop Funktionalitäten auch alle Lücken automatisiert testen zu können.

## 4.8 Schwachstellen

Der Webshop enthält bereits implementierte Schwachstellen, welche in der Vorgängerarbeit entstanden sind. Zudem sollen neue Schwachstellen im Rahmen dieser Arbeit ergänzt werden, welche in diesem Abschnitt behandelt werden.

### 4.8.1 Bereits implementierte Schwachstellen

Es folgt eine Übersicht über die in der Vorgängerarbeit implementierten Schwachstellen und deren Umsetzung. Da nicht alle Lückeen komplett oder mit einem schlüssigen Szenario entwickelt wurden, ist es nötig, dass einige Szenarien angepasst oder neu geschrieben werden.
Die Beschreibungen sind, wo referenziert, aus der Vorarbeit übernommen.

**NoSQL Injection**
Injection Angriffe gehören auch heute noch zu den häufigsten Angriffen im Web. Besonders bekannt waren früher vor allem SQL Injection Attacken, wobei NoSQL Datenbanken nicht von Injection Angriffen ausgeschlossen sind. In der Regel werden NoSQL Injection Angriffe möglich, weil die Benutzereingabe ungenügend validiert werden, was das ausführen von ungeplanten Abfragen zur Folge haben kann.[9]

**Szenario** Anstatt das sich der Hacker mit Benutzername und Passwort anmeldet, umgeht er die Prüfung mittels NoSQL Injection. Der Angriff sieht wie folgt aus:
Die Felder Benutzername und Passwort können mit dem NoSQL Befehl `"$gt": " "` abgefüllt werden. Für MongoDB bedeutet "$gt" "grösser als". Die Injection vergleicht den Datenbank Eintrag des Users oder Passwort mit dem leeren String " " und dies ergibt immer "true". Dies loggt den Angreifer mit dem ersten in der Datenbank gefundenen Benutzer ein.

**Hidden Functionality**
Im API existieren gewisse Funktionalitäten, welche nicht von Aussen aufgerufen werden sollten, oder vergessen wurden zu entfernen, wenn diese nicht oder nicht mehr benötigt werden. Trotzdem ist es möglich, diese Funktionalitäten von Aussen aufzurufen und somit sensible Daten zu erlangen oder das System zu schädigen.[4]

**Szenario** Es gibt einen Retailer Bereich, in welchem darauf hingewiesen wird, dass Retailer einen gewissen Rabatt für ihre Bestellungen erhalten. Weiter wird geschrieben, dass zurzeit keine neuen Retailer mehr angenommen werden. Zugleich wird erwähnt, dass

bestehende Retailer über die Retailer API ihren Rabatt normal einfordern können. Es gilt also, die versteckte API zu finden.
Diese Lücke ist kombiniert mit der "JWT Time Checking" Schwachstelle, in welcher die gefundene API ausgenutzt wird.

**Insecure Direct Object Reference (IDOR)**
Webanwendungen nutzen oft den internen Namen oder die Kennung eines Objektes, um auf dieses zu verweisen. Anwender können Parameter ändern, um diese Schwachstellen auszunutzen.
Beispiel: Das Profil für Nutzer A ist erreichbar unter *https://www.bank.ch/user?acc=4323*, wobei in diesem Fall 4323 die Kennung ist.
Mit dieser Information kann Nutzer A nun verschiedene Kombinationen von Zahlen ausprobieren, in der Hoffnung, damit direkt auf das Profil eines anderen Kunden zu gelangen.[10]

**Szenario** Im Shop kann ein User mehrere Kreditkarten mit Nummer, Datum und Prüfziffer hinterlegen. Wählt ein Benutzer eine Kreditkarte zur Bearbeitung aus, wird dieses Objekt direkt über die Kreditkartennummer referenziert. Der Angreifer muss dies erkennen und kann so mittels durchprobieren an fremde Kreditkarteninformationen gelangen, sofern die aufgerufene Nummer im System existiert. Im Shop sind einige Nummern fortlaufend gewählt und andere eher zufällig, somit sind nicht alle einfach auffindbar, ohne Brute-Force zu verwenden. Beispiel eines Links, über welchen eine Kreditkarte aufgerufen werden kann: https://localhost/#!/creditcard/5404000000000003.

**Unprotected REST API**
Dieses Problem taucht auf, wenn ein API Aufruf nicht genügend oder gar nicht geschützt ist, obwohl dieses nur für autorisierte Benutzer zur Verfügung stehen sollte.[4]

**Szenario** Die Lücke ist kombiniert mit der *IDOR* Schwachstelle. Ein User kann mehrere Kreditkarten besitzen und diese ändern. IDOR ist dafür verantwortlich, dass Kreditkarten direkt über die Kreditkartennummer aufgerufen werden können. Unprotected REST API hingegen ist dafür verantwortlich, dass beim Aufruf einer Kreditkarten keine Prüfung geschieht, ob der Aufrufer auch Besitzer der Kreditkarte ist.

**Scalable Vector Graphic (SVG) Injection**
Die SVG-Injection basiert auf skalierbaren Vektorgrafiken, welche mit JavaScript Code ergänzt werden können. Falls eine solche SVG-Datei auf einem System eingebettet werden kann und dabei der JavaScript Code nicht unterbunden wird, ermöglicht es vollen Zugriff auf den Inhalt der Seite.[4]

**Szenario** Im Shop gibt es einen Community Bereich, in welchem die Nutzer Bilder hochladen und kommentieren können. Die Uploadfunktion nimmt unter anderem SVG Dateien entgegen und diese können JavaScript Code beinhalten. Sobald ein anderer User die Home Seite aufruft, welche für die Darstellung aller Beiträge zuständig ist, wird im Hintergrund der Schadcode ausgeführt. Dabei kann zum Beispiel das JWT Token ausgelesen werden.
Gleichzeitig hat der Angreifer einen Webserver gestartet, welcher auf Anfragen vom bösartigen JavaScript Code wartet. Bei jedem Aufruf der Seite schickt der Browser des Opfers so unbemerkt Daten an den Webserver des Hackers.

**Stored XSS**
Als XSS Angriff wird das Einschleusen von JavaScript in eine andere Webseite bezeichnet. Diese Lücke ist seit Jahren in der OWASP Top 10 Liste vertreten. Das Ziel dabei ist meist die Session oder das Benutzerkonto eines anderen Benutzers zu übernehmen. Häufig passiert dies durch einbetten von bösartigem Code in Webformularen, die beim Laden der Seite wiederum an andere Mitbenutzer ausgeliefert werden.[11]

**Szenario** Im Shop kann ein registrierte Nutzer ein Produkt mittels Sternen und Kommentar bewerten. Das bestbewertete Produkt wird in einer Seitenleiste ständig eingeblendet und ist somit für alle Shopbesucher sichtbar.
Ein Angreifer tippt seinen bösartigen JavaScript Code in das Kommentar Feld und speichert den Kommentar. Sobald ein Benutzer eine Seite aufruft, wird im Hintergrund der Schadcode ausgeführt. Dabei kann zum Beispiel das JWT Token ausgelesen werden. Weil das bestbewertete Produkt überall eingeblendet wird, spielt es dabei keine Rolle, welche Seite das Opfer öffnet.
Gleichzeitig hat der Angreifer einen Webserver gestartet, welcher auf Anfragen vom bösartigen JavaScript Code wartet. Bei jedem Aufruf der Seite schickt der Browser des Opfers so unbemerkt Daten an den Webserver des Hackers.

### 4.8.2 Zu implementierende Schwachstellen

In diesem Abschnitt werden alle während der Arbeit neu zu entwickelnden Schwachstellen behandelt. Auf mögliche Gegenmassnahmen wird nicht eingegangen, dies wird in den Musterlösungen im Kapitel **??** *Aufgabenstellungen mit Lösungen* genauer beschrieben.

**JSON Web Token (JWT) Time Checking**
JWT Token sind eine Möglichkeit, User auf einer Plattform zu authentifizieren. Normalerweise werden diese im "Local Storage" des Browsers gespeichert. Anders als Cookies ist dieser Speicher nur von der selben Domain aus erreichbar.
Ein solches Token sollte nur für einen gewissen Zeitraum gültig sein und dies sollte auch bei jedem Request geprüft werden. Wird die Gültigkeitsdauer nicht überprüft und ein Token gestohlen, kann dieses beliebig lange missbraucht werden.[12]

**Szenario** Wie schon beschrieben gibt es Retailer Benutzer. Damit diese auch von externen Applikationen Bestellungen aufgeben können, gibt es eine API, über welches Rabatte bezogen werden können. Im Webshop gibt es dafür einen Beispiel zu sehen, welcher zugleich ein JWT Token beinhaltet. Dieses ist ein echtes Retailer Token und weil die Gültigkeitsdauer nicht geprüft wird, kann sich jeder Benutzer nachträglich Rabatte auf bestehende Buchungen verschaffen.

**CSWSH**
Eine WebSocket Verbindung wird zwischen Client und Server über einen Handshake mittels HTTP Protokoll aufgebaut. Dabei fordert der Client den Server in einem *Upgrade Request* auf, eine WebSocket Verbindung zu öffnen. Sobald die Verbindung steht, kommunizieren beide Parteien über einen symmetrischen Kanal miteinander. Gemäss Standard[8] schreibt WebSocket dabei nicht vor, wie die Authentifizierung für einen erfolgreichen Verbindungsaufbau stattzufinden hat. Dies kann also während dem WebSocket Handshake auf klassische Art mittels Cookie oder auch HTTP Authentifizierung

geschehen.
Ein Angreifer präpariert nun eine Webseite so, dass diese beim laden im Hintergrund eine WebSocket Verbindung auf den gewünschten Dienst öffnet. Hat sich ein Opfer zuvor schon einmal beim legitimen Dienst angemeldet, hat dieses ein Cookie erhalten. Steuert jetzt das Opfer die bösartige Seite an, sendet der Browser im Hintergrund automatisch das Cookie zum Dienst mit und die Verbindung wird erfolgreich aufgebaut. Dies ist der Tatsache geschuldet, dass WebSockets nicht der *Same-Origin Policy* (SOP) unterliegen.

**Szenario** In der Applikation ist ein Chat integriert, womit Nachrichten zwischen authentifizierten Benutzern ausgetauscht werden können. Ein Cookie dient im Hintergrund dazu, einen Chatuser zu identifizieren.
Da der Chat anfällig für CSWSH ist, kann sich ein Angreifer eine eigene Seite zusammenbauen, welche im Source Code ein JavaScript integriert hat, welches eine WebSocket Verbindung zum eigentlichen Shop aufbaut. Der Angreifer lockt das bereits im Shop eingeloggte Opfer auf diese Seite und der Browser öffnet daraufhin einen Kanal. Der Angreifer kann nun die Konversationen des Opfers abhören und auf seinem Server loggen.

**Node.js RCE**
Eine Lücke auf einer Webseite kann dazu führen, dass ein Angreifer beliebigen Code auf dem Server ausführen kann. Dieser Code wird im Kontext des Serverprozesses ausgeführt und kann dafür genutzt werden, weitere Prozesse zu starten oder andere Kommandos direkt auf dem Server auszuführen und somit beliebige Daten zu exfiltrieren. Je nachdem mit welchen Rechten der Serverprozess gestartet, wird ist die Übernahme des gesamte Servers möglich.
Damit dies möglich ist, braucht es eine Funktion auf der Seite, welche die vom User eingegebenen Daten zu wenig prüft. Häufig wird in diesem Zusammenhang die Funktion eval() genannt, welche bei unsachgemässer Handhabung dafür ausgenutzt werden kann.[15]

**Szenario** Ein Benutzer hat die Möglichkeit, alle seine vergangenen Bestellungen als PDF herunterzuladen. Dabei kann er mit einem "Von-Bis" Bereich entscheiden, welche Bestellungen exportiert werden sollen. Auf dem Server werden diese Inputparameter in einem eval() ausgewertet und dabei nicht auf deren Wert hin geprüft.
Der Angreifer hat so freien Zugriff auf das Zielsystem.

**Server Side Request Forgery (SSRF)**
Typischerweise tritt diese Lücke dann auf, wenn der Angreifer die Möglichkeit bekommt, einen vom Server ausgelösten Request zu beeinflussen. Nimmt eine Seite, welche einen Request auslöst, zum Beispiel als Parameter für den Request eine URL an, kann ein Hacker auf interne Dienste zugreifen, welche eigentlich nicht gegen aussen erreichbar sein sollten. Dies funktioniert, da die Anfrage vom Server selbst und somit von intern gemacht wird. Unter Umständen können so Konfigurationen oder sogar Dateien nach aussen exfiltriert werden. [18]

**Szenario** Im Community Bereich können Benutzer Beiträge mit Bildern erstellen. Ein Bild kann von der Festplatte hochgeladen oder über eine URL von extern eingebunden werden. Bilder von einer URL werden vom Server mit einer HTTP GET Abfrage heruntergeladen.

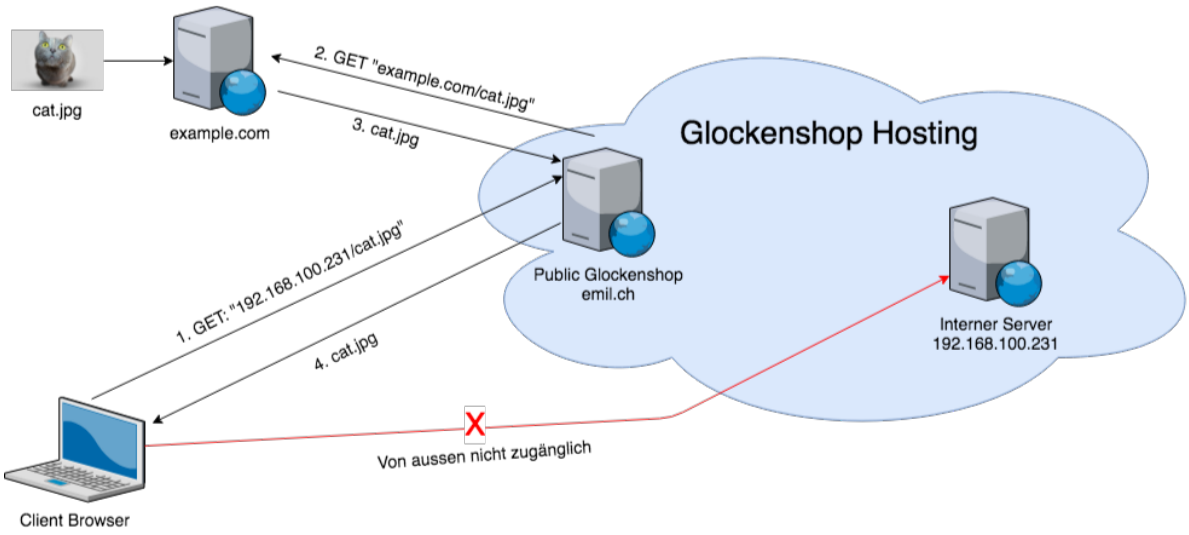


Abbildung 4.6: Vom Entwickler vorgesehene Nutzung der Funktion

Abbildung *4.6* beschreibt den Normalfall. Wird ein Bild von einer externen Ressource angefragt, entspricht dies dem vom Entwickler vorgesehenen Szenario. Das Bild wird heruntergeladen und danach angezeigt.

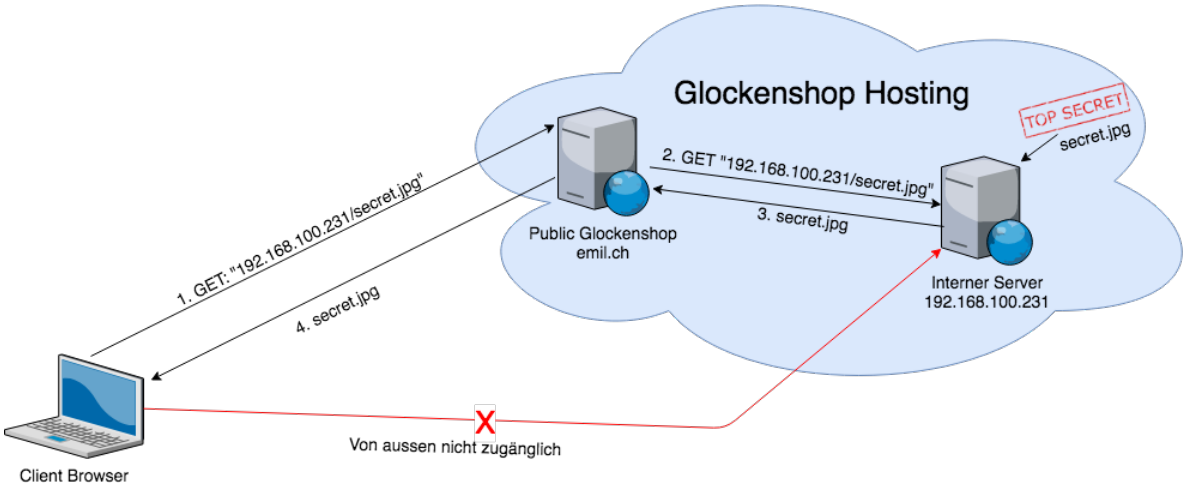


Abbildung 4.7: Aufbau eines SSRF Angriffes

Abbildung *4.7* beschreibt das Szenario im Glockenshop. Nebst dem normalen Webserver wird ein versteckter und von aussen nicht zugänglicher Server betrieben. Kennt ein Angreifer sowohl die Adresse und auch die Ressourcen des verstecken Servers, kann dies für einen Angriff ausgenutzt werden. Diese Angaben findet der Hacker über einen Hinweis auf der Startseite. Darauf ist ein Bild eingebettet, welches nicht korrekt angezeigt wird. Wird das Bild im Quelltext untersucht, ist dort sowohl die Adresse, wie auch ein Hinweis auf die Dateinamen der versteckten Ressourcen zu finden. Wird nun ein neuer Beitrag per URL und diesen Angaben erstellt, kann der Hacker so indirekt auf den versteckten Server und dessen Ressourcen zugegriffen.

**De-/Serialization Bugs**
Setzt ein Dienst in einer Funktion auf De-/Serialisierung von externen Daten, kann als Entwickler den Daten, welche beim deserialisieren entstehen, nicht einfach vertraut werden. Ein Angreifer könnte gezielt Code in den Daten verstecken, welcher auf dem Zielsystem ausgeführt wird und somit wieder eine Remote Code Execution Lücke darstellt.[16]

**Szenario** Die Suche wird so umgebaut, sodass die Anfragen serialisiert und serverseitig wieder deserialisiert werden. Dabei wird keine Inputvalidierung gemacht. Somit kann der Angreifer diese Lücke für Remote Code Exection ausnutzen. Dabei gelten die selben Grenzen und Möglichkeiten wie bei der RCE Lücke.

**Document Object Model (DOM) Based XSS**
Der DOM Tree kann durch JavaScript Code nachträglich manipuliert werden. Besitzt die Applikation zum Beispiel ein Script, welches Dropdown Felder anhand von URL Parametern nachträglich bearbeitet, so besteht die Möglichkeit, dass die nicht validierten Parameter JavaScript Code beinhalten, welche direkt ausgeführt werden. [13]

**Szenario** Im Shop befindet sich beim Produkt ein Dropdown Feld, welches ermöglicht ein Produkt mehrfach in den Warenkorb zu legen. In der Shop-URL kann der Parameter "selectedQuantity" mitgegeben werden, welcher vorgibt, welcher Wert standarmässig im Dropdown Feld steht. Somit besteht die Möglichkeit JavaScript Code im Parameter zu übergeben, welcher auf der Seite ausgeführt wird. Ein so vom Hacker präparierte Link wird danach an ein Opfer versendet, was die Lücke aktiviert. [13]

Normales Beispiel: localhost/#!/shop?selectedQuantity=1
Bösartiges Beispiel: localhost/#!/shop?selectedQuantity=<script>alert(1)</script>

**JSON Response with text/html**
Üblicherweise findet der Datenaustausch zwischen Back- und Frontend im JSON Format statt. Diese Daten werden im Normalfall als "application/json" gekennzeichnet, was die Ausführung von Scripts in modernen Browser unterbindet. Falls der Response Header fälschlicherweise auf "content-type: text/html" gesetzt ist und die Daten JavaScript Code beinhalten, führt dies dazu, dass der Code direkt im Browser ausgeführt wird. [6]

**Szenario** Auf der Detailseite eines Produkts können Fragen dazu gestellt werden. Das Produkt wird vom Server mit dem Header "content-type: text/html" geliefert und beinhaltet ebenfalls die Fragen. Für die Lieferung der Daten ist die interne API zuständig, welche die Daten als JSON liefert und auch direkt, nicht nur über die Seite, angefragt werden kann. Öffnet das Opfer einen Link mit dem API Aufruf zu dem manipulierten Produkt, ladet der Browser die JSON Daten samt JavaScript Code und führt diesen aus.

**Server-Side JavaScript (SSJS)**
Besteht in einem System die Möglichkeit, JavaScript Code an den Server zu übermitteln, welcher daraufhin ausgeführt wird, so wird von einer SSJS Attacke gesprochen. MongoDB kennt wenige Funktionen, die in einer Abfrage auch JavaScript Code als Ausdruck erlauben. Somit kann der Server auf verschiedene Arten manipuliert werden. [9]

**Szenario** Es wird eine FAQ Seite angeboten, welche die Fragen und Antworten aus der Datenbank lädt. Der FAQ ist zudem durchsuchbar, was im Hintergrund eine Datenbankabfrage mit "$where" ausführt. "$where" ist dabei eine dieser wenigen MongoDB Funktionen, welche JavaScript zur Filterung des Ergebnisses zulässt. Die Eingabewerte im Suchfeld werden nicht überprüft und somit wird der Schadcode direkt auf dem Datenbankserver ausgeführt.

**CORS Access-Control-Allow-Origin: ***
Da Webseiten der SOP unterliegen, ist es nicht ohne weiteres möglich, innerhalb einer Seite auf die Ressourcen einer anderen zuzugreifen, welche die SOP nicht erfüllt. Cross-Origin Resource Sharing (CORS) bietet die Möglichkeit solche Zugriffe zu erlauben.
Als Beispiel dient eine Wetterseite (http://www.wetter.ch). Der Entwickler dieser Seite möchte die Wetterdaten auch auf seiner zweiten Webseite
(http://www.glockenshop.ch) einbinden, was die SOP aber nicht zulässt. Der Entwickler kann den Server von "http://www.wetter.ch" so konfigurieren, dass Zugriffe von der Seite "http://www.glockenshop.ch" erlaubt werden, indem der CORS Header in der HTTP Response auf "Access-Control-Allow-Origin: http://www.glockenshop.ch" eingestellt wird.
Es gibt auch die Möglichkeit den Header mit "Access-Control-Allow-Origin: *" so zu konfigurieren, dass alle Zugriffe, egal von wo, erlaubt werden. Dies kann ein unerwünschtes Verhalten darstellen, insbesondere wenn es sich um eine nicht öffentliche Schnittstelle handelt. [14]

**Szenario** Das Webshop Frontend erhält seine Daten über eine Schnittstelle, welche die Daten im JSON Format liefert. Diese API sollte eigentlich nur vom Glockenshop erreichbar sein, da der gepflegte Produktkatalog nicht so einfach für dritte zugänglich sein soll. Weil die API aber fälschlicherweise mit "Access-Control-Allow-Origin: *" konfiguriert ist, kann von überall darauf zugegriffen werden.

**Template Injection**
Eine Webanwendung kann Nutzern die Option bieten, eine Seite für sich zu personalisieren und diese nach seinen Wünschen anzupassen. Wird der im Template beinhaltete JavaScript Code nicht validiert, so kann bösartiger Code integriert werden. Somit können zum Beispiel Benutzerdaten gestohlen werden. [17]

**Szenario** Auf der Webanwendung kann ein persönliches Profil hochgeladen werden. Hierbei handelt es sich um eine HTML-Datei, welche mit AngularJS Expressions abgefüllt werden kann. Deise Expressions werden üblicherweise durch die AngularJS Sandbox eingeschränkt und somit sollte nur die Möglichkeit bestehen JavaScript Code nur eingeschränkt zu nutzen. Doch für fast jede AngularJS Version existiert ein Sandbox Escape, welcher es ermöglicht diese Sandbox zu umgehen. Somit kann jeglicher JavaScript Code in den Expressions eingebunden werden und somit ist zum Beispiel der Zugriff auf das LocalStorage möglich. Diese Daten können nun an den vom Hacker bereits gestarteten Webserver gesendet werden, welcher diese Daten loggt.

### 4.8.3 Übersicht der Lücken

Folgende Tabelle bietet eine Übersicht über alle im Shop geplanten Lücken. Dazu sagt sie aus, ob diese neu oder bereits in der Vorarbeit angedacht waren. Des Weiteren ist ersichtlich, ob sich die Szenarien der bereits bestehenden Lücken verändert haben und was sowohl bei den neuen, als auch bei den bestehenden Lücken erledigt werden muss.

| # | **Lücke** | **Neu/Bestehend** | **Vorgehen** |
|---|---|---|---|
| 1 | NoSQL Injection | Bestehend | Nichts. Lücke ist fertig. |
| 2 | SVG-Injection | Bestehend | Nichts. Lücke ist fertig. |
| 3 | Stored XSS | Bestehend | Nichts. Lücke ist fertig. |
| 4 | Hidden Functionality | Bestehend | Leicht angepasstes Szenario. Wenig Aufwand nötig im Code. |
| 5 | IDOR | Bestehend | Leicht angepasstes Szenario. Wenig Aufwand nötig im Code. |
| 6 | Unprotected REST API | Bestehend | Komplett neues Szenario ausgedacht. Implementation als Code. |
| 7 | CSWSH | Neu | Szenario ausgedacht. Implementation als Code. |
| 8 | De-/Serialization Bugs | Neu | Szenario ausgedacht. Implementation als Code. |
| 9 | RCE | Neu | Szenario ausgedacht. Implementation als Code. |
| 10 | SSRF | Neu | Szenario ausgedacht. Implementation als Code. |
| 11 | DOM Based XSS | Neu | Szenario ausgedacht. Implementation als Code. |
| 12 | JSON Response with text/html | Neu | Szenario ausgedacht. Implementation als Code. |
| 13 | SSJS | Neu | Szenario ausgedacht. Implementation als Code. |
| 14 | Template Injection | Neu | Szenario ausgedacht. Implementation als Code. |
| 15 | CORS: Allow: * | Neu | Szenario ausgedacht. Implementation als Code. |
| 16 | JWT Time Checking | Neu | Szenario ausgedacht. Implementation als Code. |

Tabelle 4.1: Übersicht über alle vorhandenen Schwachstellen

## 4.9 Aufgabenstellungen mit Lösungen

Zu jeder Schwachstelle muss eine Aufgabenstellung mit passender Lösungsanleitung formuliert werden. Diese wird danach im Hacking-Lab veröffentlicht und daher ist als Sprache Englisch vorgegeben.

### 4.9.1 Aufgabentypen

Eine Aufgabe kann zwei Typen entsprechen, welche beide dem User sowohl das Ziel, den Kontext der Attacke, die Voraussetzungen sowie die vom Teilnehmer erwarteten Antworten vorgeben:

**Step-by-Step** Das Ziel ist, die Aufgabe unter Berücksichtigung aller Vorgaben zu lösen. Sollte der Benutzer während der Aufgabe feststecken, kann er auf eine "Step-by-Step" Anleitung zurückgreifen, um so weiter voran zu kommen.

**Wargame** Dieser Aufgabentyp ist schwieriger, denn anders als bei "Step-by-Step" gibt es keine geführte Lösung, auf welche zurückgegriffen werden kann. Gleich bleiben die abzuliefernden Informationen. Ein Wargame ist also die "Step-by-Step" Aufgabe, ohne die Lösung dazu gleich mitzuliefern.

Nach erfolgreichem Abschluss muss der Nutzer seine Lösung und einige Fragen zu der Aufgabe für die Validierung an einen Prüfer senden, damit ihm die Punkte gutgeschrieben werden.
Ob eine Aufgabe als gelöst gilt, entscheidet der Prüfer anhand der Antworten des Benutzer auf folgende Fragen:

- Explain the security problem
- Explain your attack. (exploit, screenshot, hacking journal)
- Explain mitigation (remedy)

Um es dem Prüfer möglichst einfach zu machen sind pro Aufgabe, nebst der Lösungsanleitung für den Benutzer, auch noch Musterantworten für diese Fragen bereitzustellen

### 4.9.2 Usability-Tests

Damit die geschriebenen Musterlösungen klar verständlich sind, werden die Aufgaben von Personen ohne Bezug zum Glockenshop getestet. Die Tester sollen dafür im Bereich Websicherheit nicht allzu versiert sein.
Verbesserungsvorschläge fliessen danach in eine korrigierte Fassung ein.

## 4.10 Zusammenfassung

Die ausführliche Analyse der Arbeit, unter Berücksichtigung der geforderten Resultate aus dem Kapitel **??** *Aufgabenstellung*, ergibt zusammengefasst folgende Punkte, welche während der Arbeit erledigt werden müssen:

- Die Vorgängerarbeit muss getestet und allfällige Probleme behoben werden
- Die bestehenden Schwachstellen sind den überarbeiteten Szenarien anzupassen
- Die neuen Schwachstellen müssen samt den Szenarien implementiert werden
- Der Webshop und alle Lücken müssen automatisiert getestet werden
- Für die Applikation muss ein passendes Docker Konzept erarbeitet werden
- Für alle Schwachstellen müssen Aufgaben und Musterlösungen erstellt werden
- Die geschriebenen Aufgabenstellungen müssen mit Testpersonen überprüft werden

# 5 Umsetzung

In diesem Kapitel wird auf die Umsetzung, der in Abschnitt *4.10 Zusammenfassung* definierten Arbeiten, eingegangen.

## 5.1 Bugfixing der Vorgängerarbeit

Die Vorgängerarbeit enthielt Bugs, welche vor der Implementation neuer Funktionen und Schwachstellen bereinigt werden mussten. Dieser Abschnitt handelt von deren Verbesserung.

### 5.1.1 Feedback bei Benutzerinteraktion

Der Webshop gab dem Benutzer nicht bei allen Funktionen eine Rückmeldung, sodass eine erfolgreiche oder fehlgeschlagenen Aktion nicht unterschieden werden konnten.

**Umsetzung** Bei der Bearbeitung der Lieferadresse, der Kreditkarte, der Buchung, des Warenkorbs und des Beitrags wurde die HTTP-Response des Servers um eine Meldung ergänzt. Diese Meldung wird nun auf Views nach Erhalt des HTTP-Response angezeigt, sodass der Benutzer über den Status des HTTP-Requests immer informiert ist.

### 5.1.2 Bower durch Yarn ersetzt

Um die abhängigen Packages zu laden wurde Bower eingesetzt, welches mittlerweile veraltet und demnächst nicht mehr unterstützt wird.

**Umsetzung** Die Packages werden nun durch Yarn geladen. Der Einsatz von Yarn führt dazu, dass die Packages gecached werden. Die Sicherheit steig an, da die Packages mittels Checksumme vor der Ausführung geprüft werden. Zudem steigt die Zuverlässigkeit, da die Packages unabhängig vom System installiert werden können.

### 5.1.3 Datenerfassung mittels Batch-Datei

Die Mongo-Datenbank wurde mittels Mongoose Package beim Starten des Node.js Servers befüllt. Die Daten wurden in Javascript-Dateien gehalten, welche durch abhängige Callbacks aufgerufen wurden. Dies hat zur Folge, dass die Objekte nach jedem Neustart des Node.js Servers unterschiedliche ID's erhalten haben. Dies führte dazu, dass das Schreiben von Tests schwieriger war. Zudem war die Bearbeitung der Daten mühsamer.

**Umsetzung** Die Mongo-Datenbank wird neu mittels Batch-Datei mit Daten gefüllt. Die Batch-Datei lädt die Datenbank mit Hilfe des Befehls "mongoimport", welcher die JSON-Dateien importiert. Die Objekte erhalten nun immer die gleiche ID und auch die Bearbeitung der Daten ist im JSON-Format einfacher.

### 5.1.4 Optimierung der abhängigen Packages

Die Datei Package.json, in welcher die zu ladenden Abhängigkeiten aufgeführt sind, beinhaltete überflüssige Packages.

**Umsetzung** Die überflüssigen Packages wurden mittels NPM-Modul deep-check identifiziert und bereinigt. Das Package material-design-icons war sehr gross und führte zu einer sehr langen Buildzeit der Applikation. Daher wurden die benötigen Icons direkt in die lokalen Assets gespeichert, was die Nutzung des Packages überflüssig machte.

### 5.1.5 AngularJS Trusted Ressource Bereinigung

Zu Beginn der Entwicklung der alten Webshop Applikation wurde die AngularJS-Applikation und der Node.js Server einzeln gehostet. Während der Studienarbeit wurden die Applikation so umgebaut, dass der Node.js die AngularJS-Applikation hostet. Aufgrund dieser Änderung laufen die beiden Applikationen nun unter Same-Origin und die Trusted Ressources vom Typ RESOURCE_URL wird hinfällig. Beim Erstellen eines neuen Beitrags wird diese immer noch verwendet und kann entfernt werden.

**Umsetzung** Im AngularJS wurde die Trusted Ressource vom Typ RESOURCE_URL entfernt.

### 5.1.6 Erfassung der Kreditkarte

Ein Benutzer konnte keine Kreditkarten hinzufügen, da der ein Unique Key im Kreditkarten Modell falsch gesetzt war.

**Umsetzung** Im Modell der Kreditkarte wurde der Unique Key vom Typ auf die Nummer abgeändert. Somit ist nun die Erfassung von Kreditkarten möglich.

### 5.1.7 Kreditkarten Felder erweitert

Die Kreditkarte hatte bis an hin nur das Feld Nummer. Im realen Leben werden Kreditkarten aber zusätzlich immer mit einem Typ, dem CVV Wert und Ablaufdatum erfasst.

**Umsetzung** Das Model der Kreditkarte ist um die Felder Typ, CVV Wert und Ablaufdatum ergänzt worden. Die View zur Erfassung und Bearbeitung wurden dementsprechend auch angepasst.


Abbildung 5.1: Kreditkarten mit Felderweiterungen

### 5.1.8 Buchungsprozess

Der Buchungsprozess wies den Benutzer nicht auf eine fehlende Lieferadresse oder Kreditkarte hin.

**Umsetzung** Sollte im Buchungsprozess bei der Auswahl der Lieferadresse und Kreditkarte keine Daten vorhanden sein, wird neu ein Link auf die jeweilige Maske zur Erfassung eingeblendet.

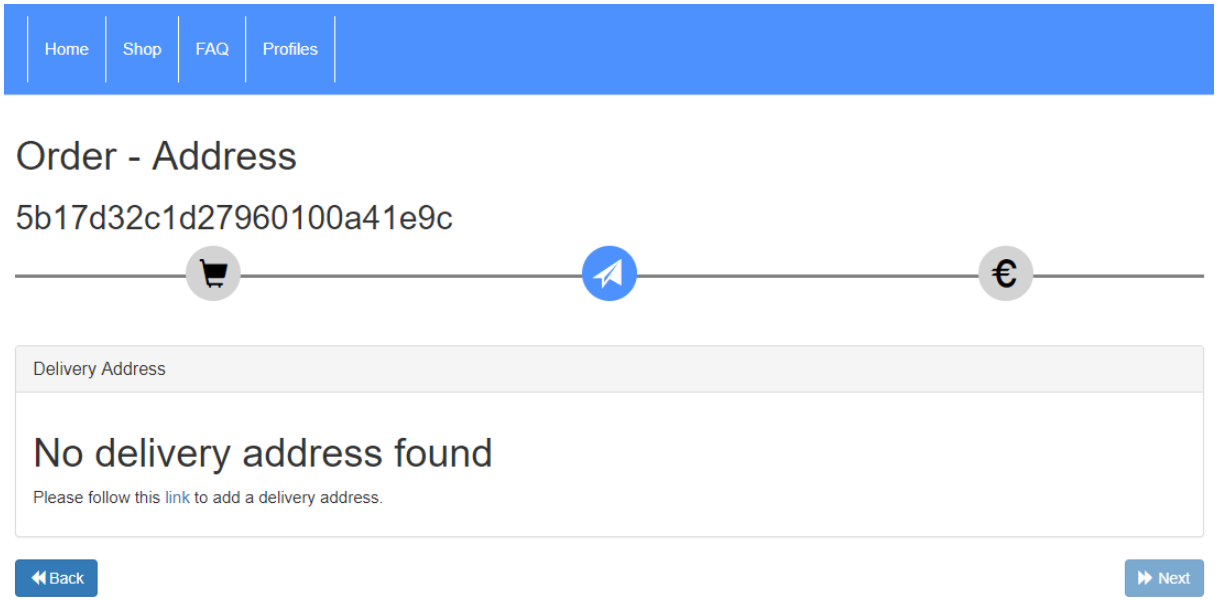


Abbildung 5.2: Buchungsprozesses mit Hinweis auf fehlende Lieferadresse

### 5.1.9 Produktbewertung

Nach der Erfassung einer Produktbewertung war nur der Wert in der Erfassungsmaske ersichtlich, nicht jedoch der Kommentar.

**Umsetzung** Der Kommentar wird nach der Erfassung einer Produktbewertung korrekt eingebunden.

### 5.1.10 Durchschnitt der Produktbewertung

Der Durchschnittswert der Bewertungen eines Produkts ist nicht korrekt berechnet worden.

**Umsetzung** Die Berechnung des Durchschnitts wurde überarbeitet, sodass er nun immer korrekt ermittelt wird.

### 5.1.11 Beitrag löschen

Nach der Erstellung eines Beitrag konnte dieser nicht gelöscht werden.

**Umsetzung** Der Beitrag kann nun vom Ersteller entfernt werden. Dies hat den Vorteil, dass SVG Bilder, welche beim Anzeigen JavaScript Code ausführen und dabei einen Alert werfen wieder gelöscht werden können und die Usabilty der Applikation nicht mehr beeinträchtigt wird.

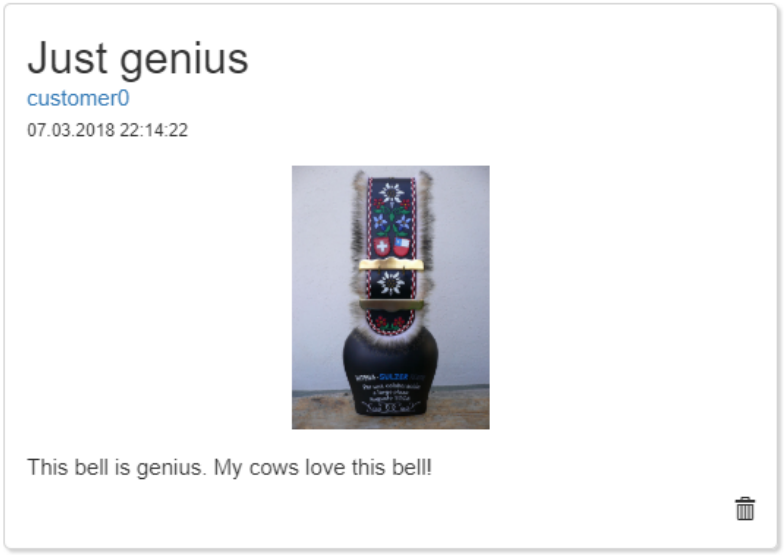


Abbildung 5.3: Beitrag mit Löschfunktionalität

### 5.1.12 Responsive Design

Die Website ist bereits mit Bootstrap aufgesetzt worden. Jedoch wurden die Responsive-Elemente nicht durchgehend verwendet. Beim Hacken wird die Seite oft verkleinert, weil auch anderer Applikation genutzt werden müssen. Dies führt dazu, dass nicht alle Elemente der Seite vollständig angezeigt werden.

**Umsetzung** Die Bootstrap-Elemente wurden wo nötig eingesetzt, damit bei verkleinertem Browserfenster noch alles ersichtlich ist.

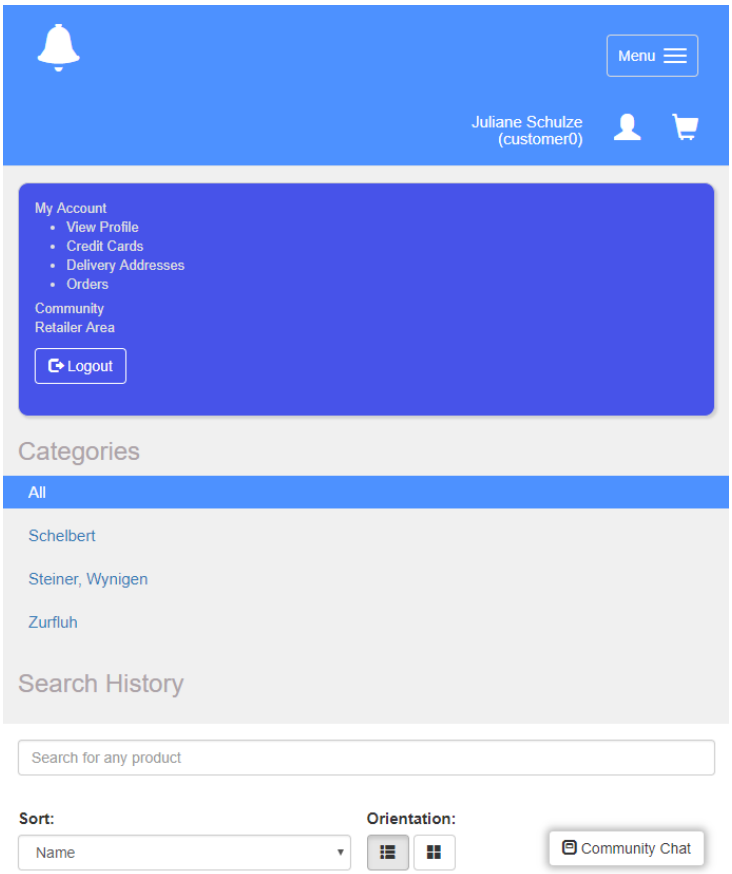


Abbildung 5.4: Responsive Design

### 5.1.13 Lightbox

Für gewisse Schwachstellen werden Bilder als Resultat einer Challenge geliefert. Diese werden zum Teil sehr klein dargestellt und sind nicht lesbar.

**Umsetzung** Es wurde das Modul "lity" eingebunden, dieses kann dann bei einem Image Tag als Attribut angegeben werden und ermöglicht es, ein Bild in einer Lightbox darzustellen.

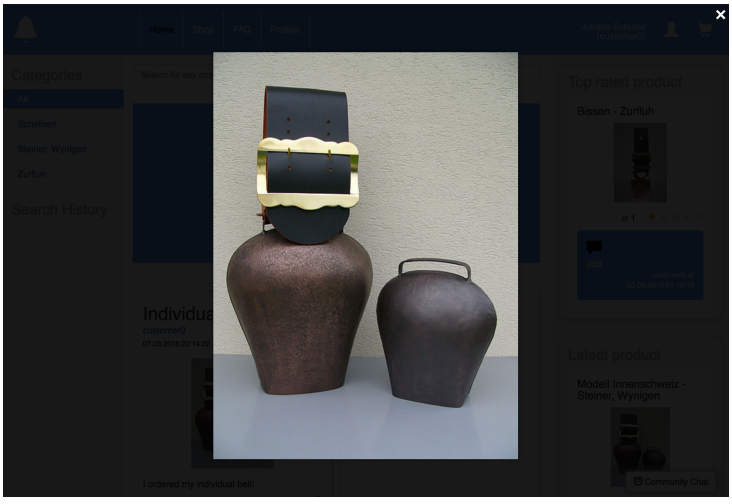

Abbildung 5.5: Lightbox Ansicht eines Bildes

## 5.2 Implementation

In diesem Abschnitt wird beschrieben, wie die neuen Funktionen mit ihrer Verwundbarkeit implementiert werden.

### 5.2.1 Architektur

Durch die vielen neuen Schwachstellen, wie auch neuen Funktionen ändert sich auch das Deployment des Systems.

#### Deployment

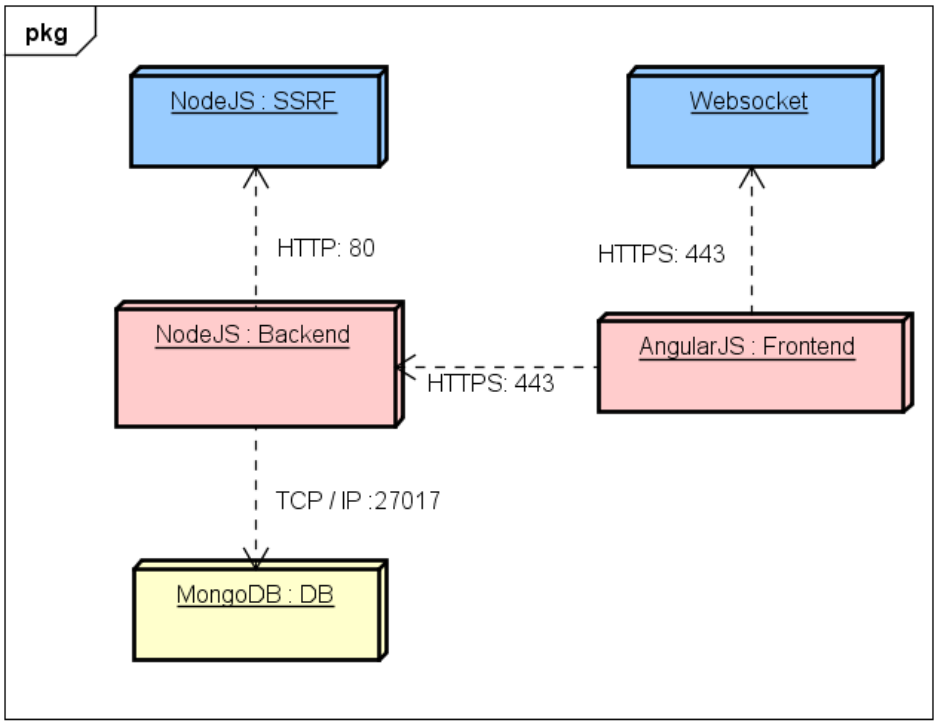


Abbildung 5.6: Deployment Diagramm

Durch die TLS Verbindung zwischen dem Client und Server wird der Port von ursprünglich 3000 auf 443 geändert und funktioniert nun über HTTPS. Zudem ist für die SSRF Schwachstelle ein neuer Server hinzugefügt worden, welcher mit dem Backend über das HTTP Protokoll mittels Port 80 kommuniziert.

**Domain Model** Das Domain Modell ist bezüglich den bestehenden Klassen der Vorgängerarbeit nur geringfügig Angepasst worden. Die meisten Änderungen des System führten dazu, dass einige neue Klassen hinzugefügt werden mussten.

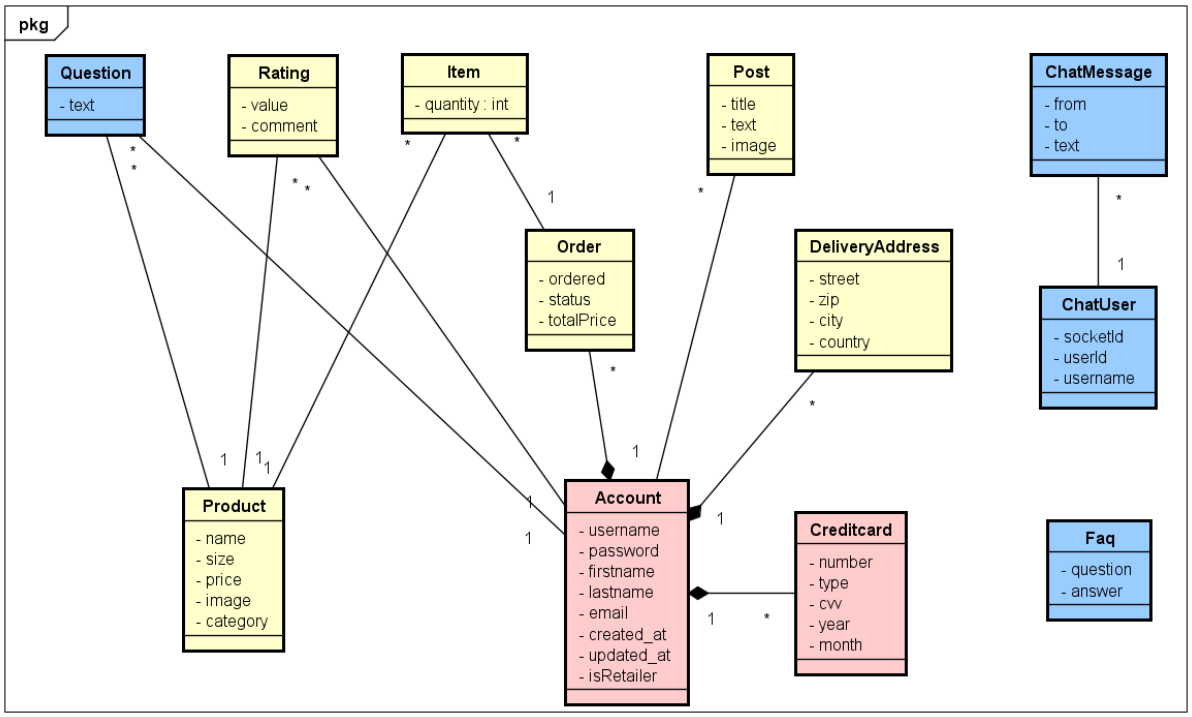


Abbildung 5.7: Domain Modell

Für die Erstellung von Fragen für ein Produkt musste das Domain Modell um die Klasse "Question" ergänzt werden. Die Funktionalität zur Bereitstellung von FAQ Fragen an die Benutzer führt dazu, dass das Modell um die Klasse "FAQ" erweitert wird. Der Community Chat setzt voraus, dass die Benutzer und die Messages nun persistent gespeichert werden. Dies soll über die zwei Klassen "ChatMessage" und "ChatUser" geschehen.

### 5.2.2 Direct Object Reference

Im Webshop soll die Kreditkarte Funktionalität durch Direct Object Reference angreifbar sein. Um dies zu gewährleisten, wurden im Code keinerlei Validierungen gemacht, welche überprüfen, ob der Benutzer auch der Inhaber der Kreditkarte ist. Desweiteren wird die Kreditkarte nicht wie bei den anderen Objekten über eine ObjectId aufgerufen, sondern über die Kreditkartennummer selbst.

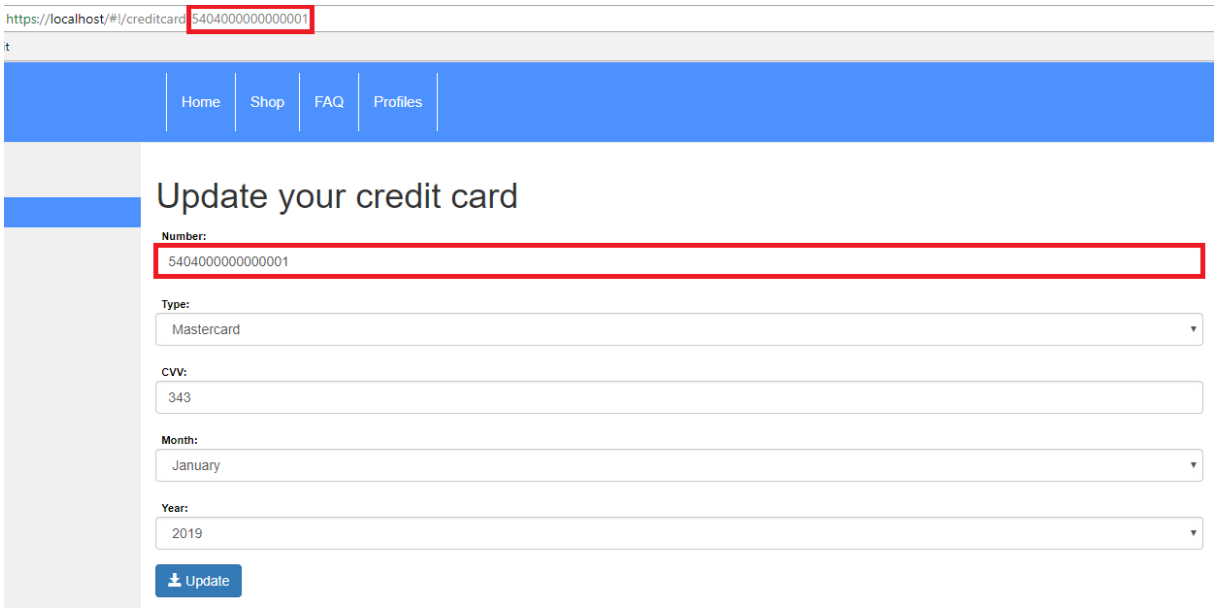


Abbildung 5.8: Direct Object Reference

Dies führt dazu, dass mittels Brute Force jegliche Kreditkarte, auch solche von anderen Benutzern, angezeigt werden kann.

### 5.2.3 CSWSH

Der Webshop soll mittels CSWSH angreifbar sein. Um dies bewerkstelligen zu können, wurde der Webshop um eine Chat Funktionalität erweitert, welche mittels WebSockets kommuniziert.

**Server Modul** Das WebSocket Modul auf dem Node.js Server nutzt die Erweiterung "socket.io", welche es ermöglicht, auf WebSocket basierend, Nachrichten vom Client an den Server und umgekehrt zu übermitteln.

```
1  'use strict';
2  const cookie = require('cookie');
3  module.exports = function(server) {
4     const io = require('socket.io')(server, {
5        transports: ['websocket']
6     });
7     let users = [];
8     io.on('connection', function(socket) {
9        let room = null;
10       if(socket.request.headers.cookie) {
11          let cookies = cookie.parse(socket.request.headers.cookie);
12          room = cookies.room;
13          if(room) {
14             socket.join(room);
15          }
16       }
17       socket.on('join', function (user) {
18          join(user);
19          socket.join(user._id);
20       });
```

```
21        socket.on('reJoin', function (user) {
22           let updatedUser = updateUser(user);
23           if(updatedUser) {
24              socket.emit('join', updatedUser);
25              io.emit('userList', users);
26              socket.join(updatedUser._id);
27           } else {
28              join(user);
29              socket.join(user._id);
30           }
31        });
32        socket.on('leave', function (data) {
33           deleteUser(data);
34           socket.emit('leave');
35           io.emit('userList', users);
36        });
37        socket.on('getMsg', function (data) {
38           socket.to(data.to).emit('sendMsg', { msg: data.msg, from: data.from });
39        });
40        io.emit('userList', users);
41        function join(user) {
42           let newUser = {
43              _id: user._id,
44              username: user.username
45           };
46           if(!existsUser(newUser)) {
47              users.push(newUser);
48           }
49           socket.emit('join', newUser);
50           io.emit('userList', users);
51        }
52        function existsUser(user) {
53           for(let i = 0; i < users.length; i++) {
54              if(users[i]._id === user._id) {
55                 return true;
56              }
57           }
58           return false;
59        }
60        function updateUser(user) {
61           for(let i = 0; i < users.length; i++) {
62              if(users[i]._id === user._id) {
63                 users[i].username = user.username;
64                 return users[i];
65              }
66           }
67           return null;
68        }
69        function deleteUser(user) {
70           for(let i = 0; i < users.length; i++) {
71              if(users[i]._id === user._id) {
72                 users.splice(i, 1);
73              }
74           }
75        }
76     });
77  };
```

Das Modul hält in der Variable "users" die Lister der Benutzer, welche bereits mit dem Socket verbunden sind.
Der Server besitzt die Event-Listener "join", "reJoin", "leave" und "getMsg", welche vom Client ausgelöst werden können.
Der Event-Listener "join" wird dazu verwendet, einen neuen Benutzer in die Benutzer-Liste einzufügen, dabei wird mittels der Hifsmethode "existsUser" geprüft, ob der Benutzer nicht bereits vorhanden ist, sodass jeder Benutzer nur einmalig vorhanden ist.
Der Event-Listner "reJoin" wird benötigt, falls ein Benutzer sich wieder mit dem Socket verbinden will. Dabei wird mittels der Hilfsmethode "updateUser" der Benutzername des Benutzers in der Benutzer-Liste aktualisiert, falls dieser sich in der Zwischenzeit verändert hat.
Der Event-Listner "leave" wird ausgelöst, wenn ein Benutzer die Verbindung mit dem WebSocket schliessen will.
Bei jeder Aktualisierung der Benutzer-Liste wird diese wieder an jeden Client gesendet.
Um Nachrichten an einen Raum eines anderen Benutzers zu senden, wird der Event-Listner "getMsg" verwendet.

**Client Controller** Im AngularJs existiert der Controller "WebSocketController", welcher auf die Interaktionen des Benutzers reagiert.

```
1  'use strict';
2
3  appControllers.controller('WebSocketController', ['$scope', 'AuthService',
   ↪  'WebSocketService', function ($scope, authService, webSocketService) {
4     const self = this;
5
6     self.user = null;
7     self.message = null;
8     self.messages = [];
9     self.userList = [];
10
11    self.selectedUser = null;
12
13    self.selectUser = function (selectedUser) {
14       self.selectedUser = selectedUser;
15    };
16
17    self.sendMsg = function () {
18       webSocketService.emit('getMsg',{
19          to : self.selectedUser._id,
20          msg : self.message,
21          from : self.user.username
22       });
23       self.message = null;
24    };
25
26    if(authService.isAuthenticated()) {
27       let authUser = {
28          _id: authService.getUser()._id,
29          username: authService.getUser().username
30       };
31
32       webSocketService.reJoin(authUser);
```

```
33    }
34
35    webSocketService.on('join', function (user) {
36       self.user = user;
37    });
38
39    webSocketService.on('leave', function () {
40       self.user = null;
41    });
42
43    webSocketService.on('userList', function (userList) {
44       self.userList = userList;
45    });
46
47    webSocketService.on('sendMsg', function (data) {
48       self.messages.push(data);
49    });
50 }]);
```

Wie auch der WebSocket Server sind auch hier die Events-Listener definiert, sodass die Applikation auf die vom Server erhaltenen Rückmeldungen reagieren kann. Der Event-Listener "join" speichert den vom Server erhaltenen Benutzer. Falls ein Benutzer bereits authentifiziert ist, wird der Event-Emitter "reJoin" ausgelöst. Der Event-Listener "leave" löscht den im Controller gespeicherten Benutzer. Der Event-Listener "userList" aktualisiert die im Controller vorhandene Benutzer-Liste. Der Event-Listener "sendMsg" aktualisiert die im Controller die erhaltenen Nachrichten von anderen Benutzern. Die Hilfsmethode "sendMsg" sendet mittels Event-Emitter an den Event-Listener des Servers.

**Client Service** Im Frontend wird zudem ein WebSocket Service bereit gestellt, welche die im Controller verwendeten Event-Listner und Event-Emitter an den Server weiterleiten.

```
1  'use strict';
2
3  appServices.factory('WebSocketService', ['$rootScope', function($rootScope) {
4     let self = this;
5     self.socket = io.connect({
6        transports: ['websocket']
7     });
8
9     return {
10       on: function (eventName, callback) {
11          self.socket.on(eventName, function () {
12             let args = arguments;
13             $rootScope.$apply(function () {
14                callback.apply(self.socket, args);
15             });
16          });
17       },
18       emit: function (eventName, data, callback) {
19          self.socket.emit(eventName, data, function () {
20          let args = arguments;
21             $rootScope.$apply(function () {
22                if (callback) {
```

```
23                     callback.apply(self.socket, args);
24                 }
25             });
26         })
27     },
28     reJoin: function(user) {
29         this.emit('reJoin', user);
30     },
31     join: function(user) {
32         this.emit('join', user);
33     },
34     leave: function(user) {
35         this.emit('leave', user);
36     }
37   };
38 }]);
```

**GUI** Das GUI wurde im authentifizierten Zustand um einen Button "Chat" ergänzt, welcher das Chat-Fenster ein- oder ausblendet. In der linken Spalte sind die Benutzer der Benutzer-Liste ersichtlich. In der rechten Spalte werden die erhaltenen Nachrichten angezeigt. Um eine Nachricht senden zu können, muss ein Benutzer ausgewählt werden, welcher die Nachricht erhalten soll und eine beliebige Nachricht im Eingabefeld eingefügt werden. Durch das klicken des Buttons "Send..." wird die Nachricht abgeschickt.

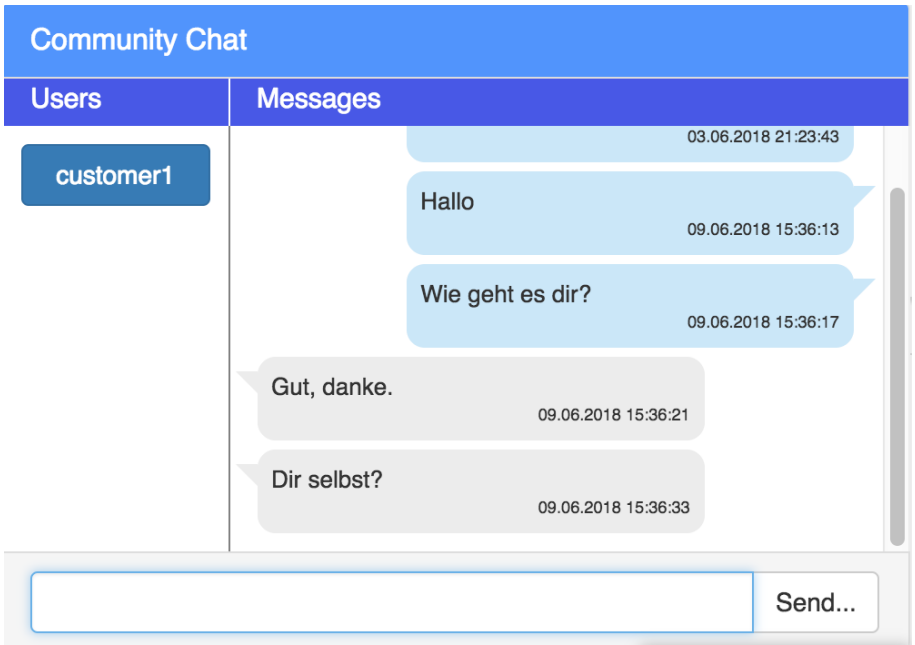


Abbildung 5.9: GUI des Chats

### 5.2.4 RCE

Der Glockenshop soll eine RCE Verwundbarkeit aufweisen, hierfür wurde der Webshop mittels einer PDF-Export Funktionalität erweitert. Diese Erweiterung ermöglicht es, die Buchungen unter "My Orders" von einem gewünschten Index bis zu einer frei wählbaren Anzahl an Einträgen als PDF zu exportieren.

**Server Controller** Auf dem Server werden die Request Parameter "From" und "Range" entgegengenommen und an den Service weitergereicht.

```
function getFromTo(req, res) {
   OrderService.getFromTo(req.params.from, req.params.range, (result) => {
      return res.json(result);
   });
}
```

**Server Service** Der Service auf dem Server nimmt die Parameter "From" und "Range" entgegen und berechnet den Wert "toResult". Dieser Wert wird abhängig von der Umgebungsvariable "NODE_RCE_EVAL" unterschiedlich berechnet.

```
function getFromTo(from, range, callback) {
   let fromResult = from;
   let toResult;

   if(process.env.NODE_RCE_EVAL.toLowerCase() === 'on') {
      toResult = eval(from + " + " + range);
   } else {
      toResult = parseInt(from) + parseInt(range);
   }

   let result = { from: fromResult, to: toResult };
   return callback(null, ResponseUtil.createSuccessResponse(result, 'Parameters
   ↪  accepted.'));
}
```

Standarmässig ist die Umgebungsvariable deaktiviert also mit dem Wert "OFF" definiert. Dies führt dazu, dass die Berechnung des Werts "toResult" nicht über die EVAL-Methode geschieht und somit die Verwundbarkeit nicht funktioniert.
Ist die Umgebungsvariable mit dem Wert "ON" definiert wird die Schwachstelle aktiviert und die Berechnung des Werts "toResult" geschieht über die EVAL-Methode, welche die Verwundbarkeit ermöglicht.
Konkret bedeutet dies, dass der Code, welcher über die Parameter übermittelt wird direkt auf dem System ausgeführt wird.

**Client Controller** Der Client Controller enthält die zwei Funktionen "downloadPDF" und "buildTableBody", welche für den Export der Tabelle zuständig sind. Die Funktion "downloadPDF" wird aufgerufen, wenn in der View der Button "Export pdf" angeklickt wird. Die Parameter "From" und "Range" werden daraufhin an den Client Service weitergereicht. Die vom Service erhaltenen Werte "From" und "To" werden in der Definition des PDF Layouts an die Funktion "buildTableBody" weitergereicht.

```
self.downloadPDF = function() {
   if(!self.export.from || !self.export.range) {
      $rootScope.messages = {};
      $rootScope.messages.warning = 'Parameter from or range is missing!';
   } else {
      ordersService.getFromTo(self.export.from, self.export.range, function (result)
  ↪  {
         let docDefinition = {
            pageOrientation: 'landscape',
            pageMargins: [ 40, 60, 40, 60 ],
            header: { text: 'Order output as pdf', margin: [25, 10, 25, 10] },
            footer: function(currentPage, pageCount) {
               return {
                  margin:10,
                  columns: [
                     { text: $filter('date')(new Date(), 'dd.MM.yyyy HH:mm:ss',
  ↪  '+0100'), alignment: 'left', margin: 25},
                     { text: currentPage.toString() + ' of ' + pageCount, alignment:
  ↪  'right', margin: 25 }
                  ]
               };
            },
            content: [
               { text: 'Orders', fontSize: 17, margin: [0, 0, 0, 25] },
               {
                  table: {
                     headerRows: 1,
                     widths: [ '*', '*', '*', '*' ],
                     body: buildTableBody(result.from, result.to)
                  }
               }
            ]
         };
         pdfMake.createPdf(docDefinition).download();
         self.export.from = null;
         self.export.range = null;
      });
   }
};
```

Die Funktion "buildTableBody" ist für die Erstellung des Tabelleninhalts zuständig und rendert nur die vom Start-Index (From) bis zum End-Index (To) Orders.

```
function buildTableBody(from, to) {
   let body = [];
   let header = [
      { text: 'Created', bold: true, fontSize: 12 },
      { text: 'Order', bold: true, fontSize: 12 },
      { text: 'Status', bold: true, fontSize: 12 },
      { text: 'Total price', bold: true, fontSize: 12 }
   ];
   body.push(header);
   let counter = 1;
   let row;
   self.data.orders.forEach(function(order) {
```

```
13        if(counter >= from && counter <= to) {
14           let createdAt = $filter('date')(order.createdAt, 'dd.MM.yyyy HH:mm:ss',
    ↪   '+0100');
15           let totalPrice = order.totalPrice.toFixed(2) + ' CHF';
16           row = [
17              { text: createdAt, bold: true, fontSize: 10 },
18              { text: order._id, bold: true, fontSize: 10 },
19              { text: order.status, bold: true, fontSize: 10 },
20              { text: totalPrice, bold: true, fontSize: 10 }
21           ];
22           body.push(row);
23           order.items.forEach(function(item) {
24              let itemName = item.quantity + ' x ' + item.product.name;
25              let itemPrice = (item.quantity * item.product.price).toFixed(2) + '
    ↪   CHF';
26              row = [
27                 { text: itemName, fontSize: 10, colSpan: 3, marginLeft: 15 },
28                 {},
29                 {},
30                 { text: itemPrice, fontSize: 10 }
31              ];
32              body.push(row);
33           });
34        }
35        counter++;
36     });
37     return body;
38  }
```

**Client Service** Der Client Service übermittelt die vom Client Controller erhaltenen Parameter "From" und "Range" mittels HTTP-Request an den Server.

```
1  getFromTo: function (from, range, callback) {
2     $http
3        .get('/api/order/from/' + from + '/range/' + range)
4        .then(function (response) {
5           let from = response.data.from;
6           let to = response.data.to;
7           let result = { "from": from, "to": to };
8           if (result) {
9              return callback(result);
10          } else {
11             return callback(false);
12          }
13       }, function (response) {
14          return callback(false);
15       });
16 }
```

**GUI** Unter "My Orders" ist nun ein Formular ersichtlich, welche den Export der Bestellungen als PDF steuert. Das Formular beinhaltet die Felder "From" und "Range", welches beides Pflichtfelder sind. Das Wert "From" setzt den Startindex der Orders und der Wert "Range" stellt die Anzahl an Einträgen dar, welche ab dem Startindex exportiert werden sollen. Über den Button "Export pdf" kann der Export gestartet werden.

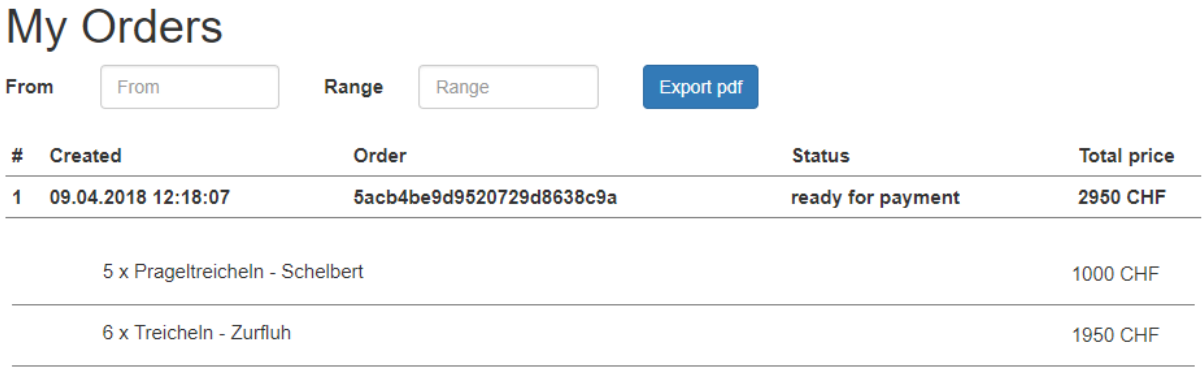


Abbildung 5.10: RCE zu GUI

**Output** Die Bestellungen werden mit der ExpressJS Erweiterung "pdfmake" in ein PDF exportiert und sieht wie folgt aus.

Orders

| Created | Order | Status | Total price |
|---|---|---|---|
| 09.04.2018 12:18:07 | 5acb4be9d9520729d8638c9a | ready for payment | 2950.00 CHF |
| 5 x Prageltreicheln | | | 1000.00 CHF |
| 6 x Treicheln | | | 1950.00 CHF |
| 09.04.2018 12:17:49 | 5acb4bd7d9520729d8638c69 | ready for delivery | 125.25 CHF |
| 1 x Berner Treicheln | | | 125.25 CHF |
| 09.04.2018 12:17:30 | 5acb4bc1d9520729d8638c33 | ready for delivery | 1485.90 CHF |
| 3 x Bissen | | | 705.00 CHF |
| 1 x Gotthardtreicheln | | | 180.90 CHF |
| 1 x Prageltreicheln | | | 200.00 CHF |
| 1 x Modell Innenschweiz | | | 400.00 CHF |

Abbildung 5.11: RCE zu Output

### 5.2.5 JWT Time Checking

Damit der Glockenshop auch mittels abgelaufenem JWT-Tokens aufrufbar ist mussten die Validierungsoptionen auf dem Server für das Token angepasst werden.

```
jwt: {
   secret: authSecret,
},
auth: {
   signOptions: {
      audience: 'self',
      issuer: 'webshop',
   },
   validateOptions: {
      secret: authSecret,
      audience: 'self',
      issuer: 'webshop'
   }
},
```

In der Konfigurationsdatei des Servers, welcher die konfigurierbaren Werte im JSON-Format enthält, wurde der Wert "Exp", welcher für die Ablaufzeit des Tokens steht, entfernt. Dies führt dazu, dass jedes JWT-Token unendlich lange gültig ist.

### 5.2.6 De-/Serialization Bug

Die Webanwendung soll durch den De-/Serialization Bug angreifbar sein. Hierfür wird der Controller auf dem Server soweit angepasst, dass die übermittelten Daten im HTTP-Request mittels der Node.js Erweiterung "node-serialize" deserialisiert werden.

```
function getBySearchValue(req, res) {
   let searchValueObj;
   if (process.env.NODE_RCE_SERIALIZATION.toLowerCase() === 'on') {
      searchValueObj = serialize.unserialize(req.body);
   } else {
      searchValueObj = req.body;
   }

   ProductService.getBySearchValue(searchValueObj, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}
```

Diese Verwundbarkeit kann über die Umgebungsvariable "NODE_RCE_SERIALIZATION" aktiviert oder deaktiviert werden.
Standarmässig ist diese Umgebungsvariable mit dem Wert "OFF" definiert und die Schwachstelle ist nicht vorhanden.
Ist die Umgebungsvariable jedoch mit dem Wert "ON" definiert, so wird die Verwundbarkeit aktiviert und das der Funktion übergebene Objekt wird mittels dem Modul "node-serialize" deserialisiert.
Dies führt dazu, dass eine IIFE direkt auf dem System ausgeführt werden kann.

### 5.2.7 Hidden Functionality

Der Glockenshop wird mit einer versteckten Retailer API ausgerüstet. Die API bietet die Funktion, einen 50% Rabatt auf eine bereits existierende Buchung zu erhalten. Die Funktionalität kann jedoch nur von einem Account, welcher als Retailer markiert ist, ausgeführt werden.

**Model und Daten** Um diese Funktionalität nutzen zu können, ist das Modell "Account" um das boolean Feld "isRetailer" erweitert worden.

```
let accountSchema = new Schema({
   username: {type: String, required: [true, 'Username is required'], unique: true},
   password: {type: String, required: [true, 'Password is required'], validate:
 ↪  passwordValidator},
   firstname: {type: String, required: [true, 'Firstname is required']},
   lastname: {type: String, required: [true, 'Lastname is required']},
   email: {type: String, required: [true, 'Email is required'], validate:
 ↪  emailValidator, unique: true},
   isRetailer: { type: Boolean, required: true, default: false }
}, {
   timestamps: {}
});
```

Das Feld wird standardmässig mit dem Wert "false" gefüllt. Dies führt dazu, dass die neu angelegten Benutzer im System nicht als Retailer definiert werden. Zudem wird ein vordefinierter Retailer angelegt, welcher die Berechtigungen hat, diese Funktion zu nutzen.

```
{
   "_id": {
      "$oid": "5acc851fc8bc262214c01ee5"
   },
   "username": "retailer0",
   "password": "5ca00d6802d80442a0f978673b68b0f212dafae915e2f1c43fc06046e569afd8",
   "firstname": "Jackob",
   "lastname": "Müller",
   "email": "Jackob.Mueller@gmail.com",
   "isRetailer": true,
   "createdAt": {
      "$date": "2018-03-07T20:14:22.568Z"
   },
   "updatedAt": {
      "$date": "2018-03-07T20:14:22.568Z"
   },
   "__v": 0
}
```

**Route** Für die Umsetzung ist eine Retailer Route angelegt worden.

```
'use strict';

const express = require('express');
const router = express.Router();
const retailerController = require('../controllers/retailer');
const GlobalConfig = require('../configs/index');
const jwt = require('express-jwt');

router.get('/order/:orderId/change/', jwt(GlobalConfig.auth.validateOptions),
  ↪  retailerController.change);

module.exports = router;
```

Die Route prüft, ob ein gültiges Token vorhanden ist und ruft die Funktion "change" des Retailer Controllers auf.

**Controller** Der Controller überprüft, ob der Benutzer gesetzt ist und zusätzlich, ob es sich um einen Retailer handelt. Falls dies der Fall ist, wird der Retailer Service aufgerufen.

```
'use strict';

const RetailerService = require('../services/retailer');

function change(req, res) {
   if(req.user && req.user.isRetailer) {
      let orderId = req.params.orderId;
      RetailerService.change(orderId, function (error, result) {
         if (error) return res.render('feedback', { feedback: error });
         return res.render('feedback', {feedback: result });
      });
   } else {
      return res.render('feedback', { feedback: { statusCode: 500, data: null,
  ↪  message: 'User is not authenticated for this    operation!'} });
   }
}

module.exports = {
   change,
};
```

**Service** Der Service sucht sich die Buchung über die ID und prüft, ob es sich bei der übermittelten Buchung um eine mit Rechnung zu bezahlende Buchung handelt. Ist dies der Fall, wird die Buchung mit einem 50% Rabatt versehen. Dabei wird der Preis jedes Produktes, sowie auch die Summe in der Buchung mit 0.5 Multipliziert.

```
1  'use strict';
2
3  const Order = require('../models/order');
4  const ResponseUtil = require('../utils/response');
5
6  function change(orderId, callback) {
7     Order.findById(orderId, function (error, result) {
8        if (error) return callback(ResponseUtil.createErrorResponse(error));
9        if (!result) return callback(ResponseUtil.createNotFoundResponse());
10       if (result.payment.type === 'bill') {
11          result.totalPrice = (result.totalPrice * 0.5).toFixed(2);
12          result.items.forEach(function (item) {
13             item.product.price = (item.product.price * 0.5).toFixed(2);
14          });
15          result.save();
16       } else {
17          return callback(ResponseUtil.createErrorResponse('Expected payment type:
    ↪  bill'));
18       }
19       result = {'order': result};
20       return callback(null, ResponseUtil.createSuccessResponse(result));
21    });
22 }
23
24 module.exports = {
25    change
26 };
```

### 5.2.8 Template Injection

Für die Szenarion der Template Injection benötigt es einer Möglichkeit, ein Template so zu bearbeiten, dass mittels JavaScript code, welcher über eine Expression im Template ausgeführt wird, aus der Angular Sandbox ausgebrochen werden kann. Hierfür ist eine Profil Seite erstellt worden, welche für die jeweiligen anderen Benutzer im System ersichtlich ist.

**My Account Ansicht** Die Account Ansicht musste so angepasst werden, dass ein Template hochgeladen werden kann, welches dann unter "My Profile" ersichtlich ist. Ein Beispiel-Template ist unter dem Bereich "Profile example" zu finden. Dieses kann heruntergeladen werden und lokal editiert werden. Anschliessend wird dieses im Bereich "Upload your profile" angehängt, um es dann zu speichern. Im Bereich "Share your profile" ist der eigene Link zum Profil ersichtlich, welcher mit anderen Benutzern geteilt werden kann.

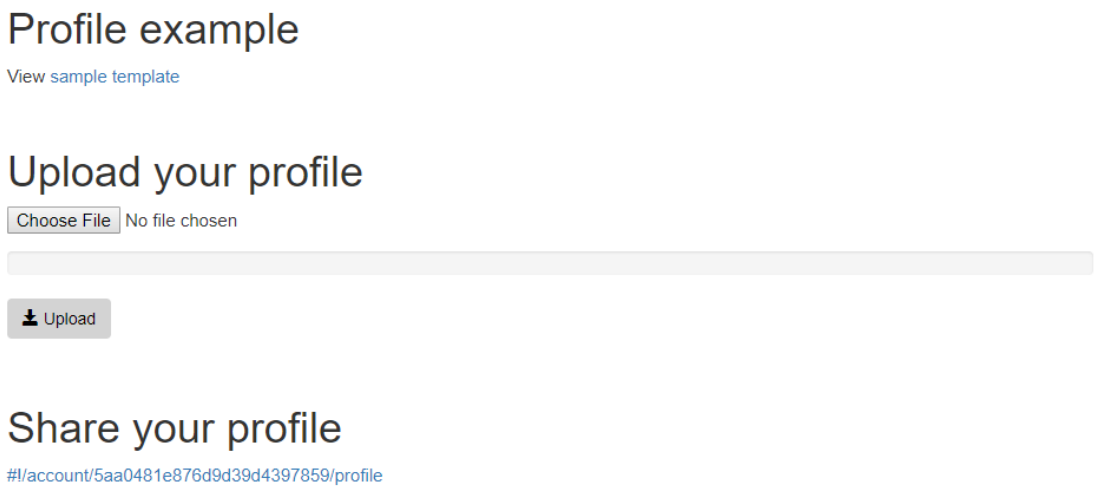


Abbildung 5.12: Template Injection zu Account

Das Formular akzeptiert eine HTML Datei, welche dem Backend übergeben wird.

**My Profile Ansicht** Das eigene Profil kann über das Menü unter "My Profile" aufgerufen werden und lädt dann die hochbeladene HTML Seite.

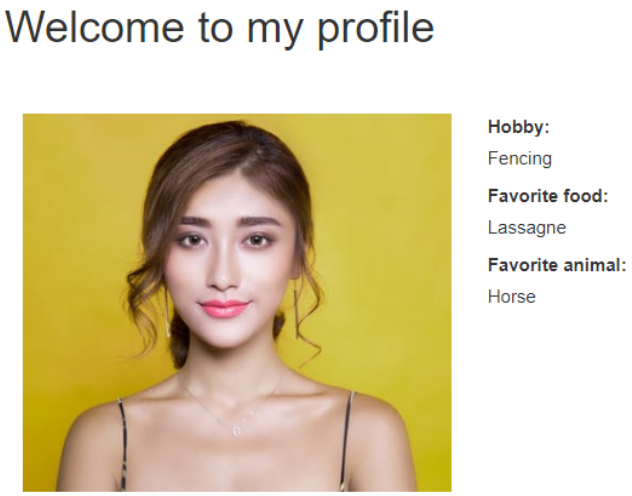


Abbildung 5.13: Template Injection zu Profile

**Clientseitiger Controller** Der Client Controller hat die Aufgabe, den Upload auf der Account Seite zu handhaben.

```
self.upload = function () {
   let accountId = AuthService.getUser()._id;
   $rootScope.messages = {};
   AccountService.upload(accountId, self.data.profile, function (error, data,
    ↪  message, validations) {
      if (error) $rootScope.messages.error = error;
      if (validations) $rootScope.messages.validations = validations;
      if (!data) $rootScope.messages.warning = message;
      if (data) {
         self.data.account = data.user;
         $rootScope.messages.success = message;
      }
   }, function (progress) {
      self.data.progress = progress;
   });
};
```

Die Methode "upload" ruft den Client Service mit der Methode "upload" auf und übergibt ihm den angemeldeten Benutzer und die angefügte HTML Datei als Parameter.

**Clientseitiger Service** Der Client Service hat die Aufgabe, den Upload zu handhaben und die Daten an das Backend weiterzureichen.

```
upload: function (accountId, profile, callback, callbackEvent) {
   let data = {accountId: accountId, profile: profile};
   Upload.upload({
      url: '/api/account/profile',
      data: data,
   }).then(function (response) {
      let statusCode = response.data.statusCode;
      let data = response.data.data;
      let message = response.data.message;
      let validations = response.data.validations;
      if (statusCode === 200) {
      let responseData = data;
         return callback(null, responseData, message, null);
      } else if (statusCode === 405) {
         return callback(null, null, null, validations);
      }
      return callback(null, null, message, null);
   }, function (error) {
      return callback(error);
   }, function (event) {
      let progressPercentage = parseInt(100 * event.loaded / event.total);
      return callbackEvent(progressPercentage);
   });
},
```

Die Methode "upload" nimmt den Benutzer und die hochzulandende Datei entgegen und übermittelt diese via HTTP-Request an das Backend.

**Serverseitiger Controller** Der Server Controller nimmt die Anfrage des Clients entgegen und übergibt diese dem Server Service.

```
function upload(req, res) {
   let accountId = req.body.accountId;
   let profileFile = req.file;
   let profile = profileFile.buffer.toString().replace(/<script.*>/g,
    ↪  '&ltscript&gt').replace(/<\/script>/g, '&lt/script&gt');
   let fileName = accountId + '.' + mime.getExtension(profileFile.mimetype);
   let filePath = GlobalConfig.accountProfile.directory + fileName;

   fs.writeFile(filePath, profile, function(error) {
      if(error) {
         return res.status(500).json(ResponseUtil.createErrorResponse('Upload
    ↪  failed'));
      }
      AccountService.upload(accountId, profile, (error, result) => {
         if (error) return res.json(error);
         return res.json(result);
      });
   });
}
```

Die HTML Datei wird zuvor jedoch mittels der Replace Methode escapet, dass heisst die Script-Tags werden entfernt, sodass kein JavaScript Code eingefügt werden kann und somit auch nicht auf der Seite ausgeführt wird.

**Serverseitiger Service** Der Server Service sucht einen Benutzer mit der übermittelten "accountId" und fügt das Profil dem Account hinzu.

```
function upload(accountId, profile, callback) {
   Account.findOne({_id: accountId}, function(error, result) {
      if (error) return callback(ResponseUtil.createErrorResponse(error));
      if (!result) return callback(ResponseUtil.createNotFoundResponse());
      result.profile = profile;
      result.save(function (error, user) {
         if (error) return callback(ResponseUtil.createErrorResponse(error));
         if (!result) return callback(ResponseUtil.createNotFoundResponse());
         result = {'user': user};
         return callback(null, ResponseUtil.createSuccessResponse(result, 'Profile
    ↪  successfully uploaded.'));
      });
   });
}
```

### 5.2.9 SSRF

Für das SSRF Szenario wird ein zusätzlicher Node.js Server benötigt. Dieser ist sehr simpel. Er kümmert sich nur um das Errorhandling und liefert ansonsten statisch eine einzige HTML Datei aus. Des weiteren muss die Community Funktion, welche es ermöglicht Beiträge zu speichern, so erweitert werden, dass auch Beiträge mittels URL erstellt werden können.
Nebst der HTML Datei befinden sich für die Aufgabe zwei Bilder auf dem Server, welche vom Benutzer gefunden werden müssen.

```
1  'use strict';
2
3  const express = require('express');
4  const ssrf = express();
5
6  ssrf.use(express.static(__dirname + '/public/ssrf'));
7
8  // catch 404 and forward to error handler
9  ssrf.use(function (req, res, next) {
10    let err = new Error('Not Found');
11    err.status = 404;
12
13    if(process.env.NODE_ENV === 'production') {
14       res.redirect('/index.hmtl');
15    }
16    next(err);
17 });
18
19 // error handler
20 ssrf.use(function (err, req, res) {
21    if(process.env.NODE_ENV === 'development') {
22       res.locals.message = err.message;
23       res.status(err.status || 500);
24       res.send(err.message);
25    }
26    res.status(500);
27    res.send('An error occurred!');
28 });
29
30 module.exports = ssrf;
```

**Clientseitiger Controller** Der Controller wird um die Methode "insertUrl" erweitert, welche den Upload vom Beiträgen mittels URL ermöglicht.

```
1  'use strict';
2
3  appControllers.controller('PostController', ['$rootScope', '$scope', '$location',
   ↪  'PostService', 'AuthService', 'AlertsService',
4  function ($rootScope, $scope, $location, PostService, AuthService, AlertsService) {
5     const self = this;
6     self.data = {};
7     self.data.post = {};
8     self.data.url = "";
9     self.data.postImage = {};
```

```
10    self.data.progress = 0;
11    self.error = false;
12    self.type = 0;
13
14    self.insert = () => {
15       switch (self.type) {
16          case 1:
17             self.insertUpload();
18             break;
19          case 2:
20             self.insertURL();
21             break;
22          default:
23             AlertsService.addWarning('No Type selected');
24             break;
25       }
26    };
27
28    self.insertUpload = () => {
29       self.data.post._account = AuthService.getUser()._id;
30       PostService.insertUpload(self.data.post, self.data.postImage, self.updatePost,
   ↪  (progress) => {
31          self.data.progress = progress;
32       });
33    };
34
35    self.insertURL = () => {
36       self.data.post._account = AuthService.getUser()._id;
37       PostService.insertURL(self.data.post, self.data.url, self.updatePost);
38    };
39
40    self.updatePost = (error, data) => {
41       if(data) {
42          self.data.post = {};
43          $location.path('/home');
44       }
45    };
46 }]);
```

Diese Methode ergänzt den Beitrag um den aktuellen Benutzer und ruft die Methode "insertUrl" des Services auf.

**Clientseitiger Service** Der Service ruft via HTTP Request die Backend Route "/api/-post/url" auf und übergibt als Daten den erfassten Beitrag.

```
1  insertURL(post, url, callback) {
2     let data = {url: url, post: post};
3     $http
4        .post('/api/post/url', data)
5        .then(
6           (response) => ResponseService.successCallback(response, callback),
7           (error) => ResponseService.errorCallback(error, callback)
8     );
9  },
```

Diese Daten werden vom Backend Controller entgegengenommen.

**Serverseitiger Controller** Die Methode "insertUrl" im Controller ist für die Entgegennahme der Beiträge, welche mit einer URL gespeichert werden, zuständig.

```
function insertURL(req, res) {
   let post = req.body.post;
   let url = req.body.url;
   PostService.insertURL(post, url, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}
```

Die Daten werden danach an den Backend Service weitergereicht.

**Serverseitiger Service** Im Service wird die Methode "insertUrl" aufgerufen, welche den Input validiert. Die Validierung umfasst die Prüfung, ob es sich um eine gültige URL handelt. Des weiteren wird geprüft, ob die Dateiendung der Datei in der URL einer Bilddatei entspricht. Zu guter Letzt wird das Bild von der URL heruntergeladen und gespeichert.

```
function insertURL(post, url, callback) {
   let parsedFile = "";
   try {
      parsedFile = path.parse(urlParse.parse(url).pathname);
   } catch (err) {
      return callback(ResponseUtil.createErrorResponse('Not a valid URL'));
   }
   let extension = parsedFile.ext;
   if (extension !== '.jpg' && extension !== '.jpeg' && extension !== '.gif' &&
   ↪  extension !== '.png') {
      return callback(ResponseUtil.createErrorResponse('Only jpg, jpeg, gif and png
   ↪  are allowed'));
   }
   let scrambledFileName = crypto.randomBytes(16).toString('hex');
   post.image = "";
   const options = {
      url: url,
      dest: GlobalConfig.postImages.directory + scrambledFileName + extension
   };
   download.image(options)
      .then(() => {
         post.image = scrambledFileName + extension;
         let postObj = new Post(post);
         handleInsertPost(postObj, callback);
      }).catch((error) => {
         return callback(ResponseUtil.createErrorResponse(error, 'Could not fetch
   ↪  image from given URL'));
      });
}
```

Nach der fehlgeschlagenen Validierung wird der Client über die nicht erfolgreiche Erstellung des Beitrags informiert. Bei erfolgreichem Download der Datei wird der neu erstellte Post an den Client gesendet.

**Route** Weil es nun 2 verschiedene Methoden für die Bildspeicherung gibt mussten auch die Routen verändert werden, sodass jede Methode eine eigene Route bekommt:

```
1 router.get('/', PostController.getAll);
2 router.post('/upload', jwt(GlobalConfig.auth.validateOptions), upload,
  ↪  PostController.insertUpload);
3 router.post('/url', jwt(GlobalConfig.auth.validateOptions),
  ↪  PostController.insertURL);
4 router.delete('/:postId', jwt(GlobalConfig.auth.validateOptions),
  ↪  PostController.remove);
```

**GUI** Vorher war nur der Bildupload möglich. Mit der URL als zweite Option benötigt die View eine Auswahl der Option und die Möglichkeit eine URL zu hinterlegen.

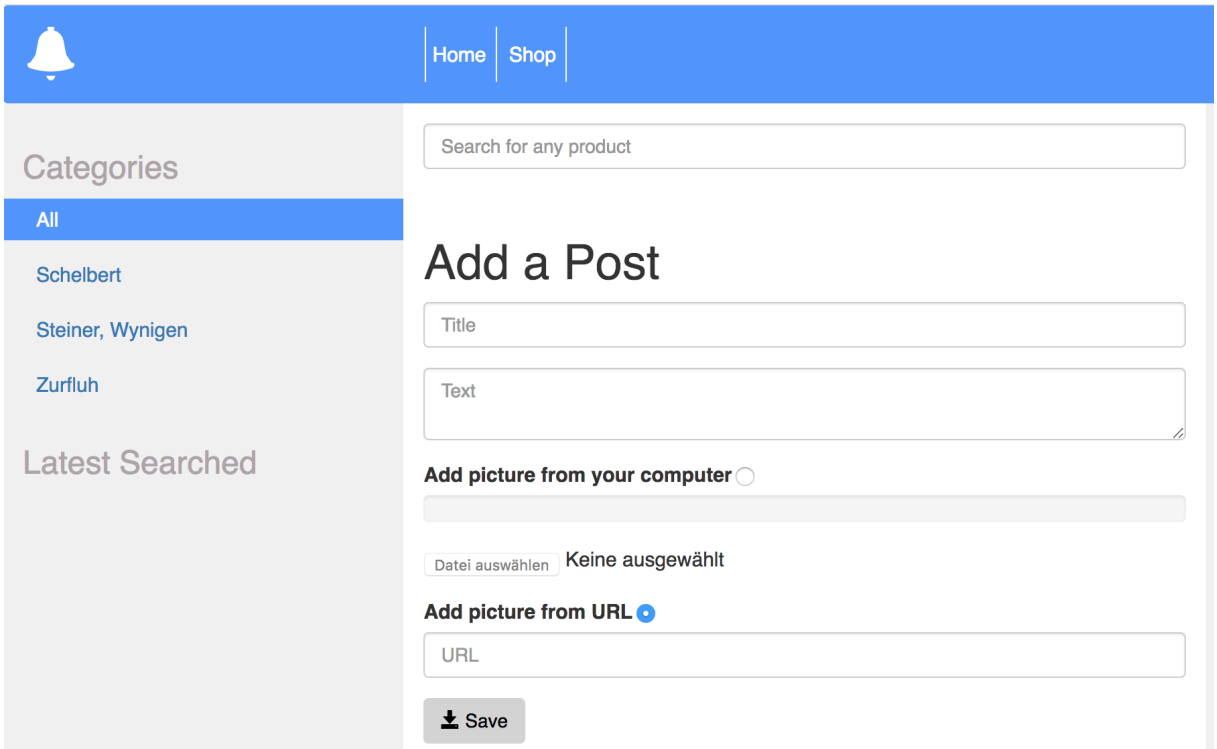


Abbildung 5.14: Neue Ansicht der Bilderupload Funktion

### 5.2.10 CORS Access-Control-Allow-Origin *

Die API des Glockenshops soll für jegliche Anfragen von fremden Seiten offen sein. Dazu wird die SOP gänzlich ausgeschaltet. Mit der Ergänzung des Node.js CORS Modules im "app.js" ist dies schnell umgesetzt.

```
1  'use strict';
2  const express = require('express');
3  const logger = require('morgan');
4  const bodyParser = require('body-parser');
5  const forceSSL = require('express-force-ssl');
6  var cors = require('cors')
7
8  require('./utils/mongo');
9
10 const api = require('./routes/api');
11 const auth = require('./routes/auth');
12 const account = require('./routes/account');
13 const creditCard = require('./routes/creditCard');
14 const deliveryAddress = require('./routes/deliveryAddress');
15 const order = require('./routes/order');
16 const product = require('./routes/product');
17 const post = require('./routes/post');
18 const retailer = require('./routes/retailer');
19 const app = express();
20
21 app.set('view engine', 'pug');
22
23 app.use(cors());
24 app.use(forceSSL);
25 app.use(logger('dev'));
```

Ab jetzt schickt der Server im Header auf alle Anfragen: "Access-Control-Allow-Origin: *", was einem Drittbenutzer erlaubt, beliebigen Inhalt auf seiner eigenen Seite einzubinden.

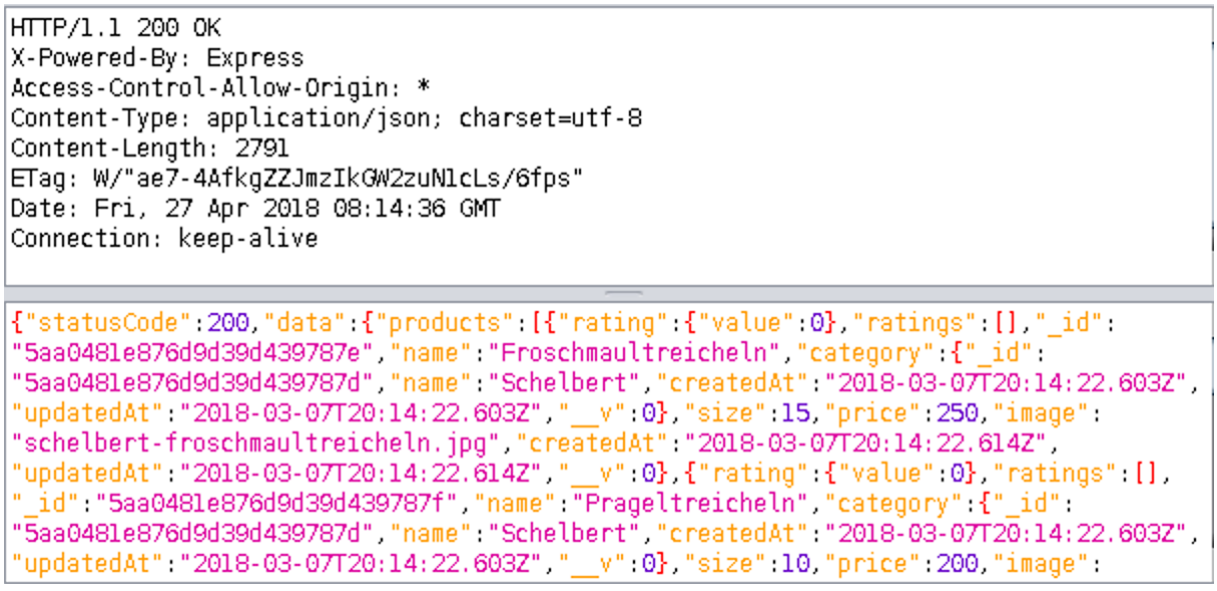


Abbildung 5.15: CORS Header für die API Abfrage aller Produkte

### 5.2.11 DOM Based XSS

Der Webshop soll für DOM Based XSS Attacken angreifbar sein. Hierfür ist ein Parameter notwendig, welcher mit der URL mitgeliefert wird und schlussendlich in den DOM Tree eingefügt wird.

**Clientseitiger Controller** Der Client Controller ist für die Aktualisierung des Parameters "selectedQuantity" zuständig. Falls sich dieser ändert, wird dieser auch auf dem Client Service angepasst.

```
1 self.selectedQuantity = $routeParams.selectedQuantity;
2
3 $scope.$watch(() => {
4    return $routeParams.selectedQuantity;
5 }, (selectedQuantity) => {
6    ShopService.selectedQuantity = selectedQuantity;
7 }, true);
```

**Clientseitiger Service** Bei jeder Aktualisierung der Produkte wird den Produkten der Wert "selectedQuantity" hinzugefügt. Dies ist notwendig, damit bei der Änderung der Anzahl auf einem Produkt nicht bei allen Produkten die gleiche Anzahl ausgewählt wird, sonder pro Produkt das Dropdown-Feld unterschiedlich gesetzt werden kann.

**Clientseitige View** Die Shop View besitzt ein Dropdown-Feld, welches das Model "selectedQuantity" nutzt und zugleich auch der Standartwert dieses Dropdown-Feldes ist.

```
1 <select add-options class="quantity form-control"
  ↪  ng-model="product.selectedQuantity">
2    <option value="1">1</option>
3    <option value="2">2</option>
4    <option value="3">3</option>
5    <option value="4">4</option>
6    <option value="5">5</option>
7 </select>
```

**GUI** Beim laden der Shop Seite sieht dies wie folgt aus.

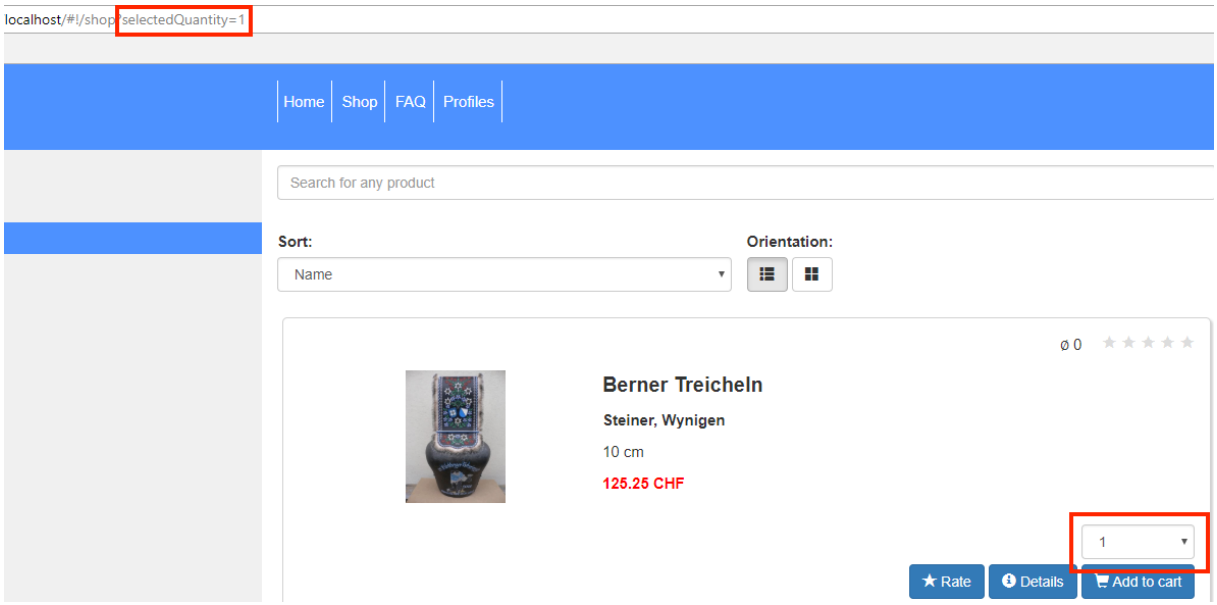


Abbildung 5.16: DOM Based XSS mit URL-Parameter

Der Standardwert wird bei allen auf 1 gesetzt und kann nachträglich bei jedem Produkt verändert werden.

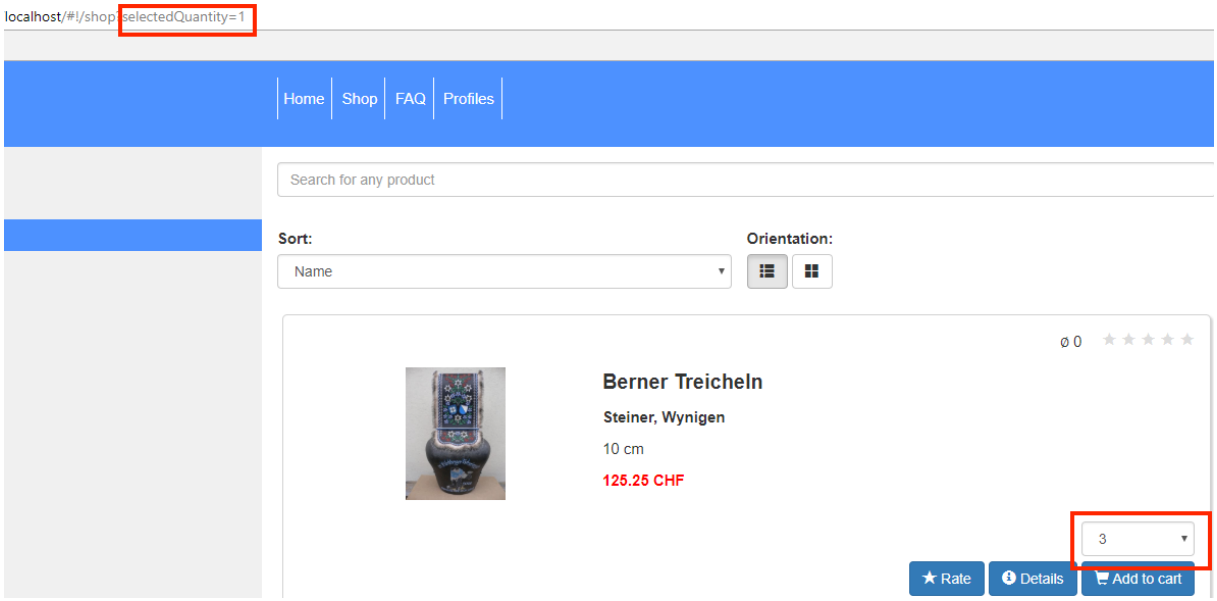


Abbildung 5.17: DOM Based XSS mit URL-Parameter, jedoch mit manueller Änderung auf den Produkten

Nun kann ein Produkt nicht nur einmal, sondern mehrmals dem Warenkorb hinzugefügt werden.

### 5.2.12 JSON Response text/html

Für dieses Szenario muss der Webshop um eine weitere Funktionalität erweitert werden. Auf der Shop Seite soll jedes Produkt eine Detail Ansicht erhalten, welche es authentifizierten Benutzern erlaubt, Fragen zu einem Produkt zu stellen.

**Clientseitiger Controller** Der Controller übernimmt die Aufgabe der Weiterleitung der Suche eines einzelnen Produkts und die Erstellung einer Frage zu einem Produkt an den Service.

```
1  'use strict';
2
3  appControllers.controller('ProductController', ['$scope', '$routeParams',
   ↪  'AuthService', 'ProductService', function ($scope, $routeParams, AuthService,
   ↪  ProductService) {
4    const self = this;
5
6    self.data = {};
7    self.data.product = {};
8    self.data.question = {};
9
10   self.init = () => {
11      self.get($routeParams.productId);
12   };
13
14   self.get = (productId) => {
15      ProductService.get(productId, (error, data) => {
16         if (data) {
17            let product = data.product;
18            self.data.product = product;
19         }
20      });
21   };
22
23   self.saveQuestion = () => {
24      self.data.question._account = AuthService.getUser()._id;
25      ProductService.saveQuestion(self.data.product._id, self.data.question, (error,
   ↪  data) => {
26         if (data) {
27            let product = data.product;
28            self.data.question = {};
29            self.data.product = product;
30         }
31      });
32   };
33
34   self.init();
35 }]);
```

Die Methode "get" liest die aus der URL die Produkt ID und gibt diese an den Service weiter. Die Speicherung der Fragen eines Produkt übernimmt die Methode "saveQuestion". Hier wird der aktuell registrierte Benutzer der Frage hinzugefügt und an den Service weitergereicht.

**Clientseitiger Service** Der Service ruft die entsprechenden Backend Routen für die zwei Funktionen auf.

```
'use strict';

appServices.factory('ProductService', ['$http', 'ResponseService', function ($http,
  ↪  ResponseService) {
   return {
      get(productId, callback) {
         $http
            .get('/api/product/' + productId)
            .then(
               (response) => ResponseService.successCallback(response, callback),
               (error) => ResponseService.errorCallback(error, callback)
            );
      },
      saveQuestion(productId, question, callback) {
         let data = { productId: productId, question: question };
         $http
            .post('/api/product/questions', data)
            .then(
               (response) => ResponseService.successCallback(response, callback),
               (error) => ResponseService.errorCallback(error, callback)
         );
      }
   };
}]);
```

Die Suche eines bestimmten Produkts wird über die Methode "get" gehandhabt, welche im Backend die Route "/api/produkt/:produktId" aufruft. Die Speicherung der Frage wird über die Methode "saveQuestion" vollzogen. Diese ruft im Backend die Route "/api/product/questions" auf und übergibt als Daten die erfasste Frage.

**Serverseitiger Controller** Auf dem Server nimmt der Controller die Anfragen entgegen und ruft auf dem Service die entsprechende Funktion auf.

```
function getById(req, res) {
   let productId = req.params.productId;
   ProductService.getById(productId, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      res.setHeader('content-type', 'text/html');
      return res.send(result);
   });
}

function insertQuestion(req, res) {
   let productId = req.body.productId;
   let question = req.body.question;
   ProductService.insertQuestion(productId, question, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}
```

Die Methode "getById" ruft auf dem Backend Service die gleichnamige Methode auf und fordert das Produkt mit der gesuchten ID an. Das Speichern der Frage wird über die Methode "insertQuestion" geregelt. Diese ruft wiederum die gleichnamige Methode auf dem Service auf.

**Serverseitiger Service** Der Service hat die Aufgabe, Fragen zu einem bestimmten Produkt hinzuzufügen, oder ein spezifisches Produkt über dessen ID zu finden.

```
function getById(productId, callback) {
   // Find product by id and populate questions
   Product.findById(productId)
      .populate({
         path: 'questions._account',
         select: '_id username'
      }).exec((error, product) => {
         if (error) return callback(ResponseUtil.createErrorResponse(error,
  ↪  'Something went wrong.'));
         if (!product) return callback(ResponseUtil.createNotFoundResponse('No
  ↪    product found.'));
         let data = {'product': product};
         return callback(null, ResponseUtil.createSuccessResponse(data));
      });
}


function insertQuestion(productId, question, callback) {
   let questionObj = new Question(question);
   // Find product and update question and populate account
   Product.findOneAndUpdate({
      _id: productId
   }, {
      $push: {questions: questionObj}
   }, {
      new: true,
      setDefaultsOnInsert: true,
      runValidators: true,
      context: 'query'
   }).populate({
      path: 'questions._account',
      select: '_id username'
   }).exec((error, product) => {
      if (error && error.hasOwnProperty('errors')) return
  ↪  callback(ResponseUtil.createValidationResponse(error.errors));
      if (error) return callback(ResponseUtil.createErrorResponse(error, 'Something
  ↪  went wrong.'));
      if (!product) return callback(ResponseUtil.createNotFoundResponse('Question
  ↪  failed to save.'));
      const data = {'product': product};
      return callback(null, ResponseUtil.createSuccessResponse(data, 'Question
  ↪    saved successfully.'));
   });
}
```

**GUI** Das Produkt wird um einen Button "Details" erweitert, welcher das Produkt in einer Detailansicht anzeigt.

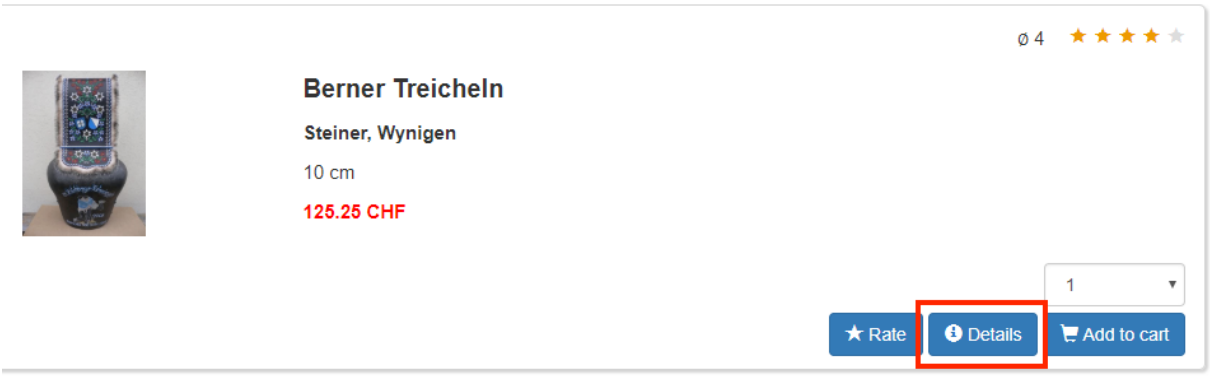


Abbildung 5.18: JSON Response text/html Shop Ansicht

Die Detailansicht sieht wie folgt aus.

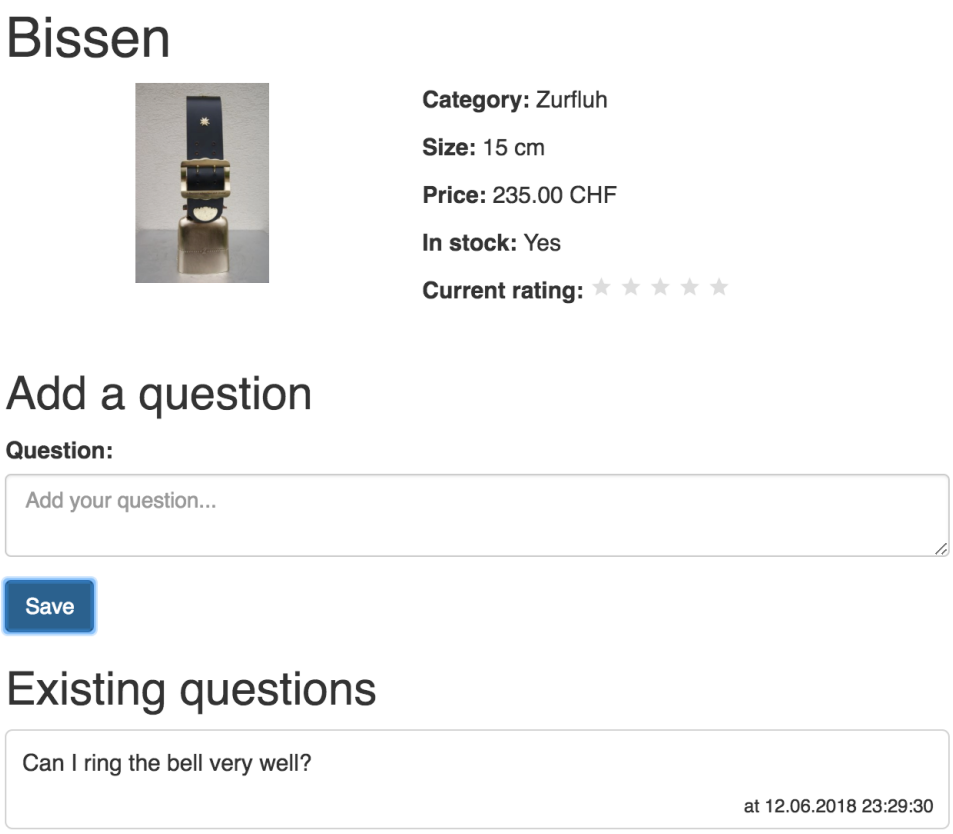


Abbildung 5.19: JSON Response text/html Produkt Details

Das Produkt wird auf einer eigenen Seite mit den selben Informationen wie auf der Shop Ansicht dargestellt. Die Darstellung der Informationen weicht jedoch leicht von der in der Shop Ansicht ab. Des weiteren existiert ein Formular, welches es ermöglicht, eine Frage zu erfassen und die erstellen Fragen einzusehen.

### 5.2.13 SSJS

Der Webshop soll für eine SSJS Attacke angreifbar sein. Hierfür wird eine FAQ Seite bereitgestellt, welche die Benutzer über bereits aufgetauchte Fragen betreffend der Bedienbarkeit des Shops informiert.

**Clientseitiger Controller** Dieser Controller ist für das Laden und Filtern der FAQ Fragen zuständig.

```
1  'use strict';
2
3  appControllers.controller('FaqController', ['$scope', '$routeParams', 'FaqService',
   ↪  function ($scope, $routeParams, FaqService) {
4     const self = this;
5     self.data = {};
6     self.data.searchValue = null;
7     self.spinner = false;
8
9     self.init = () => {
10       self.getFaq();
11    };
12
13    self.getFaq = () => {
14       FaqService.getFaq(self.updateFaq);
15    };
16
17    self.getFaqBySearchValue = () => {
18       self.spinner = true;
19       if (self.data.searchValue) {
20          FaqService.getFaqBySearchValue(self.data.searchValue, self.updateFaq);
21       } else {
22          self.getFaq();
23       }
24       self.data.searchValue = '';
25    };
26
27    self.updateFaq = (error, data) => {
28       if (data) {
29          self.data.FaqQuestions = data.faq;
30       } else {
31          self.data = {};
32       }
33       self.spinner = false;
34    };
35
36    self.init();
37 }]);
```

Die Methode "getFaq" wird beim Initialisieren direkt aufgerufen und lädt alle FAQ Fragen. Die Methode "getFaqBySearchValue" ist für das Filter der Fragen zuständig und wird ausgeführt, wenn der Filter angepasst wird.

**Clientseitiger Service** Der Service regelt die Kommunikation zwischen den Client und dem Server und aktualisiert die FAQs des Controllers.

```
'use strict';

appServices.factory('FaqService', ['$http', 'ResponseService', function ($http,
  ↪  ResponseService) {
   return {
      getFaq(callback) {
         $http
            .get('/api/faq')
            .then(
               (response) => ResponseService.successCallback(response, callback),
               (error) => ResponseService.errorCallback(error, callback)
            );
      },
      getFaqBySearchValue(searchValue, callback)  {
         let data = {searchValue: searchValue};
         $http
            .post('/api/faq/searchValue/', data)
            .then(
               (response) => ResponseService.successCallback(response, callback),
               (error) => ResponseService.errorCallback(error, callback)
            );
      },
   };
}]);
```

**Serverseitiger Controller** Der Controller nimmt die Anfragen des Clients entgegen und ruft die entsprechende Funktion im Service auf.

```
'use strict';

const FaqService = require('../services/faq');

function get(req, res) {
   FaqService.get((error, result) => {
   if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}

function getFaqBySearchValue(req, res) {
   let searchValueObj = req.body;
   FaqService.getFaqBySearchValue(searchValueObj, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}

module.exports = {
   get,
   getFaqBySearchValue
};
```

**Serverseitiger Controller** Der Controller nimmt die Anfragen des Clients entgegen und ruft die entsprechende Funktion im Service auf.

```
'use strict';

const FaqService = require('../services/faq');

function get(req, res) {
   FaqService.get((error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}

function getFaqBySearchValue(req, res) {
   let searchValueObj = req.body;
   FaqService.getFaqBySearchValue(searchValueObj, (error, result) => {
      if (error) return res.status(error.statusCode).json(error);
      return res.status(result.statusCode).json(result);
   });
}

module.exports = {
   get,
   getFaqBySearchValue
};
```

**Serverseitiger Service** Der Service auf dem Server führt die Datenbank anfragen aus. Die Daten werden nachfolgend an den Client zurückgeschickt.

```
'use strict';

const Faq = require('../models/faqQuestion').Faq;
const ResponseUtil = require('../utils/response');

function get(callback) {
// Get Faqs
   Faq.find({})
      .then((faq) => handleGetFaq(faq, callback))
      .catch((error) => {
      return callback(ResponseUtil.createErrorResponse(error, 'Something went
   ↪  wrong.'));
   });
}

function getFaqBySearchValue(searchValueObj, callback) {
   let param = searchValueObj.searchValue;
   // Get Faqs by searchValue
   Faq.find({ $where: "'"+ param +"'; /" + param + "/i.test(this.question) || /" +
   ↪  param + "/i.test(this.answer);" })
      .then((faq) => handleGetFaq(faq, callback))
      .catch((error) => {
         return callback(ResponseUtil.createErrorResponse(error, 'Could not process
   ↪  your input.'));
      });
```

```
23 }
24
25 function handleGetFaq(faq, callback) {
26    if (!faq.length) return callback(ResponseUtil.createNotFoundResponse('No faqs
   ↪  found.'));
27    const data = {'faq': faq};
28    return callback(null, ResponseUtil.createSuccessResponse(data));
29 }
30
31 module.exports = {
32    get,
33    getFaqBySearchValue
34 };
```

Hier ist speziell zu erwähnen, dass die Methode "getFaqBySearchValue" das Model FAQ durchsucht und mittels $where mit dem übergebenen Suchparameter "searchValueObj" filtert. Hierbei kann JavaScript Code der Methode übergeben werden, welche zugleich auch auf der Datenbank ausgeführt wird.

**GUI** Der Shop wird um eine FAQ Seite erweitert. Diese Seite soll Benutzern die Möglichkeit geben, sich über bereits aufgetauchte Fragen bezüglich der Applikation zu informieren.

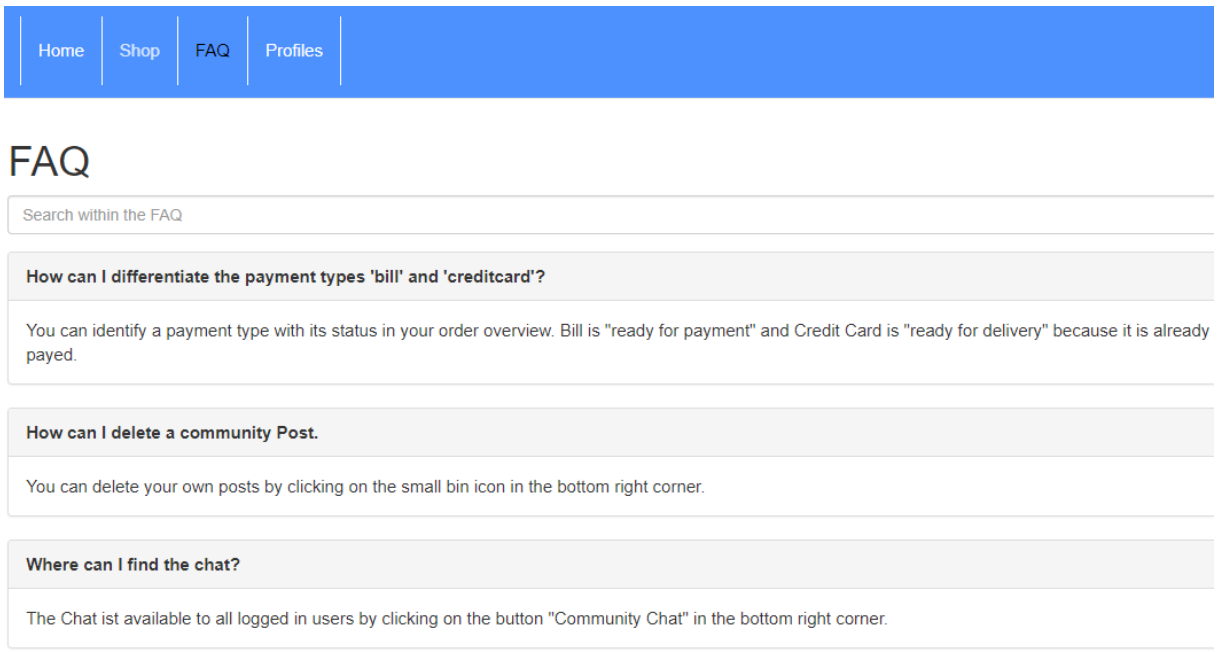


Abbildung 5.20: SSJS FAQ

Das Filtern der Fragen wird über das Feld "Search within the FAQ" umgesetzt. Nach Eingabe des gewünschten Suchbegriffs werden alle Fragen angezeigt, welche mit dem gesuchten Begriff übereinstimmen.

### 5.2.14 Transport Layer Security (TLS)

Der Glockenshop soll nur via TLS erreichbar sein und dafür muss der Webserver HTTPS:// Verbindungen entgegen nehmen können. Dabei nicht vergessen werden dürfen Benutzer, welche eine Seite mittels HTTP:// oder ganz ohne Protokoll aufrufen, diese müssen entsprechend weitergeleitet werden.
Aus diesen Anforderungen ergibt sich, dass zwei "Listener" benötigt werden:

- Der Webserver hinter Port 443, über welchen die ganze Applikation laufen soll.
- Listener hinter Port 80, welcher alle Aufrufe an Port 443 weiterleitet.

**Weiterleitung** Die Weiterleitung geschieht mit dem HTTP Status Code 301, welcher für "Moved Permanently" steht und für das Vorhaben der passende darstellt. Im folgenden Diagramm ist der Ablauf genauer dargestellt:

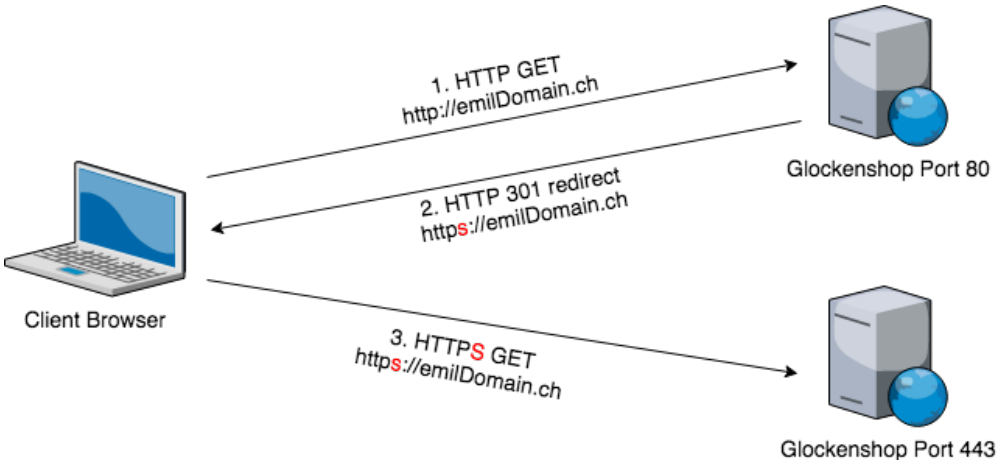


Abbildung 5.21: HTTP zu HTTPS Weiterleitung

**Load Balancer** Eine solche Weiterleitung funktioniert in diesem Beispiel nur, weil die TLS Verbindung auf dem Webshop selbst terminiert. Würde ein Load Balancer oder eine andere Netzwerkkomponente davorgestellt, welche die TLS Verbindung terminiert, würde diese Konfiguration unter Umständen in einer Endlosschleife enden.
Die Weiterleitung müsste dann ausgeschaltet werden und auf dem Load Balancer geschehen.

**Zertifikat** Die TLS Verbindung benötigt ein Zertifikat. Während der Entwicklung wird ein "Self-signed" Zertifikat verwendet. Später wird vom Hacking-Lab Personal auf ein gültiges Wildcard Zertifikat gewechselt. Ein solches Zertifikat inklusive Private Key kann mittels OpenSSL und folgendem Befehl erzeugt werden:

```
1 openssl req -x509 -newkey rsa:2048 -sha256 -keyout key.pem -out cert.pem -days 3650
```

**Event Listener Funktionen** Die Applikation wurde zu Beginn nur für einen Server ausgelegt. Beim starten des Prozesses und binden des Ports und IP an den Prozess werden mehrere Events ausgelöst. Für zwei dieser Events gibt es "Event Listener", welche aber so gebaut sind, dass nur mit einem Webserver korrekt umgegangen werden kann. Der

Code benötigte also Refactoring, sodass beliebige Instanzen eines Listeners diese Events korrekt auslösen.
Dafür wurden die "Event Listener" Funktionen "onError" und "onListenings" so angepasst, dass der Port der Serverinstanz oder die Instanz selbst mitgegeben werden kann.

```
/**
* Event listener for HTTP server "error" event.
*/
function onError(error, port) {
   if (error.syscall !== 'listen') {
      throw error;
   }
   let bind = typeof port === 'string'
      ? 'Pipe ' + port
      : 'Port ' + port;

   // handle specific listen errors with friendly messages
   switch (error.code) {
      case 'EACCES':
         console.error(bind + ' requires elevated privileges');
         process.exit(1);
         break;
      case 'EADDRINUSE':
         console.error(bind + ' is already in use');
         process.exit(1);
         break;
      default:
         throw error;
   }
}

/**
* Event listener for HTTP server "listening" event.
*/
function onListening(|\colorbox{yellow}{server}|) {
   let addr = server.address();
   let bind = typeof addr === 'string'
      ? 'pipe ' + addr
      : 'port ' + addr.port;
   debug('Listening on ' + bind);
}
```

Die Serverinstanzen registrieren entsprechenden Funktionen neu mit ihrem Port und der eigenen Instanz:

```
server.listen(port, host, function() {
   let address = server.address();
   let host = address.address;
   let port = address.port;
   console.log('HTTP Server started: http://' + host + ':' + port + ' in ' +
   ↪  app.get('env') + ' mode');
});
server.on('error', (error) => onError(error, port));
server.on('listening', () => onListening(server));

```

```
10 secureServer.listen(sslPort, host, function() {
11     let address = secureServer.address();
12     let host = address.address;
13     let port = address.port;
14     console.log('HTTPS Server started: https://' + host + ':' + port + ' in ' +
    ↪  app.get('env') + ' mode');
15 });
16 secureServer.on('error', (error) => onError(error, sslPort));
17 secureServer.on('listening', () => onListening(secureServer));
```

### 5.2.15 Testing

Beim Testing wurde der Fokus auf E2E Tests gelegt. Die Test sind mit Protractor, einem für Angular entwickelten Testframework, erstellt worden.
Protractor ermöglicht in Zusammenarbeit mit dem Selenium Web Driver das Testen der grafischen Benutzeroberfläche. Der Selenium Web Driver imitiert im Browser die von Usern ausgeführten Aktionen. Die erstellten Testfälle decken aktuell alle implementierten Funktionen, sowie alle Schwachstellen ab.

## 5.3 Docker

Im folgenden Abschnitt wird darauf eingegangen, welche Überlegungen für das Deployment der Applikation mit Docker gemacht wurden und die daraus resultierte Umsetzung.

### 5.3.1 Voraussetzungen

Der Glockenshop besteht zur Laufzeit aus zwei Komponenten: Webserver und Datenbank. Als Faustregel gilt, dass sich ein Docker Container um eine Angelegenheit kümmert und somit zwei Container benötigt werden. Dies bringt Vorteile wie Skalierbarkeit, Sicherheit und Fehlerbehandlung. Für Applikationen mit mehreren Komponenten würde daher normalerweise mit Docker Compose gearbeitet, um alle Abhängigkeiten zu definieren und die Applikation möglichst simpel zu starten. Hacking-Lab kennt in der jetzigen Version kein Docker Compose und daher kann nur ein Docker Container hinterlegt werden.
Darum bleibt als einzige Lösung, die Datenbank und den Webserver im gleichen Container unterzubringen. Somit muss zum Teil auf die genannten Vorteile verzichtet werden. Für dieses Projekt fällt dies nicht ins Gewicht, da es sich um einen Webshop mit gezielten Schwachstellen handelt und es keiner Skalierung benötigt.

### 5.3.2 Deployment Diagramm

Aus den Voraussetzungen lässt sich folgendes Deployment Diagramm für die Docker Umgebung ableiten.

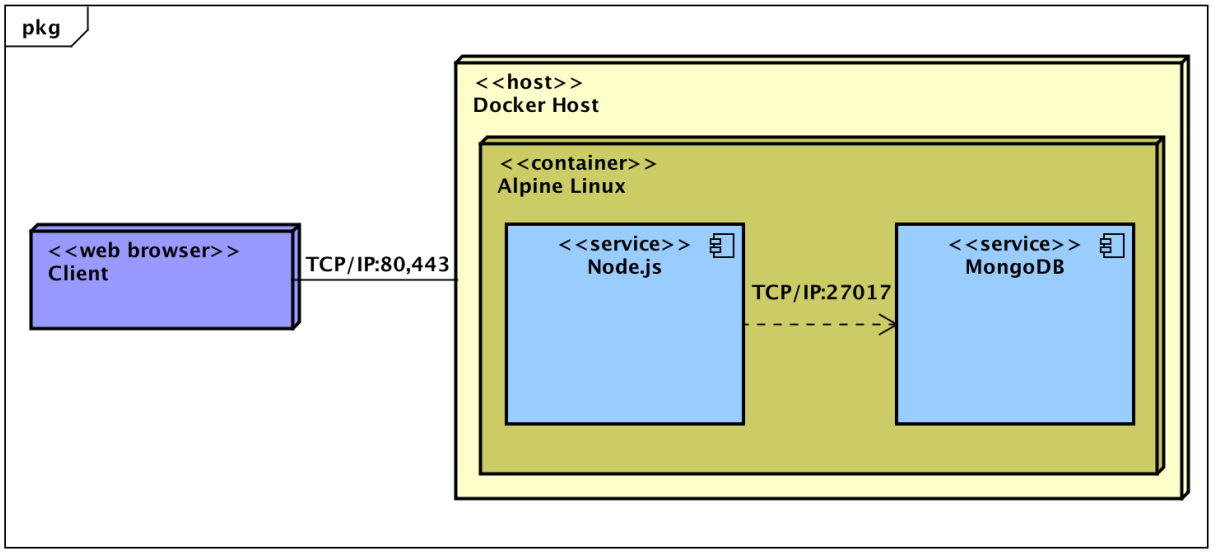


Abbildung 5.22: Docker Deployment Diagramm

### 5.3.3 Vergleich verschiedener Docker Varianten

Für einen möglichst realistischen Webshop werden Testdaten wie User oder Bestellungen benötigt. Überlegungen, wie diese in die Datenbank gelangen, haben 3 verschiedene Varianten ergeben:

**Variante 1** Das Docker Image wird erstellt. Beim jedem Start des Containers werden die Testdaten per Skript in die Datenbank importiert. Das Tool für den Import ist nicht vorinstalliert und muss daher im Image integriert sein. Dies vergrössert das Image um ~100MB.

**Vorteile**

- Simples Dockerfile
- Kurze Build-Zeit

**Nachteile**

- Mittelgrosses Image
- Längere Startzeit

**Variante 2** Bereits während dem Build-Prozess werden die Testdaten in die Datenbank importiert. Beim Starten gibt es somit keine Verzögerung. Das Tool für den Import ist nicht vorinstalliert und muss daher im Image integriert sein. Dies vergrössert das Image um ~100MB.

**Vorteile**

- Simples Dockerfile
- Kürzere Startzeit

**Nachteile**

- Grosses Image
- Längere Build-Zeit

**Variante 3** Die Daten werden während dem Build-Prozess in die Datenbank importiert. Es gibt somit keine Verzögerung beim Starten. Das Tool für den Import ist nicht vorinstalliert. Es werden aber Docker Multi-Stage Builds verwendet um dies auszugleichen.

**Vorteile**

- Kleines Image
- Kürzere Startzeit

**Nachteile**

- Längere Build-Zeit
- Komplexeres Dockerfile

**Gemessene Werte** Die Auswirkungen der Vor- und Nachteile als Zahlen ausgedrückt:

| Variante | Image Grösse in MB | Build-Zeit in s | Startzeit in s |
|---|---|---|---|
| 1 | 335 | 43 | 4 |
| 2 | 546 | 49 | 2 |
| 3 | 267 | 116 | 2 |

Tabelle 5.1: Docker: Vergleich der Varianten

**Analyse der Ergebnisse** Build-Zeiten können bei diesen Differenzen über alle Varianten generell vernachlässigt werden, da ein Image nur selten neu erstellt wird.

Variante 1 und 2 unterscheiden sich in Grösse und Startzeit. Variante 2 startet schneller, da die Daten beim Start des Containers bereits importiert sind. Der Grössenunterschied ist darauf zurückzuführen, dass bei Variante 2 die Daten bereits im fertigen Image enthalten sind. Dafür musste die Datenbank für den Import gestartet werden. Dies wiederum löst seitens der MongoDB die Erstellung von zwei Transaktions-Dateien für die Datenbankkonsistenz aus, welche jeweils ca. 100MB gross sind und ebenfalls im finalen Image enthalten sind.

Variante 3 ist die aufwändigste, bringt dafür aber die Vorteile der kurzen Startzeiten inklusive reduzierter Image Grösse gegenüber den anderen Varianten mit sich. Dank Multi-Stage Builds können in das finale Image nur die benötigten Daten übernommen werden. Das MongoDB Import Tool und die Transaktions-Dateien entfallen somit, was Platz spart.

**Fazit** Da stets ein kleines Docker Image angestrebt werden sollte und die Startzeit für die User möglichst klein sein soll, geht Variante 3 als beste Lösung hervor.

### 5.3.4 Konfiguration

**Base Image** Für Node.js stehen verschiedene Docker Images als Basis zur Verfügung. [20] Darin sind bereits viele Abhängigkeiten installiert sie sind daher ideal für den Aufbau eines eigenen Images. Die Entscheidung fiel auf das "Alpine Linux" basierte Image, mit welchem sehr kleine Images möglich sind.

**Node.js als Root** Es ist ratsam, einen Webserver mit einem eigenen Benutzer zu starten, welcher nur die nötigen Berechtigungen besitzt. Für eine von Haus aus "Defekte" Applikation wird dies vernachlässigt. Im Gegenteil, die Aufgabenstellungen können sich noch ausweiten, zum Beispiel bis zur Übernahme des Servers.

**Entrypoint** Ein Container besitzt immer einen "Entrypoint". Dies ist der Prozess, welcher beim Starten des Containers ausgeführt wird, zum Beispiel der Webserver. Da nun mehrere Prozesse in einem Container ausgeführt werden, muss das Prozesshandling selbst gemacht werden. Der einfachste Ansatz dafür ist ein Bash Skript, welches als Entrypoint dient und dann die einzelnen Prozesse separat startet.
Die nötige Filestruktur und Skripte werden während dem Docker Build-Prozess angelegt.

```
1  #!/bin/ash
2  #set pipefail so piped commands exit with errorcode of previous command
3  set -o pipefail
4  LOG="/var/log/all"
5  touch $LOG
6
7  for a in /utils/services/*
8  do
9     printf "---------\nstarting $a \n" 2>&1 | tee -a $LOG
10    $a 2>&1 | tee -a $LOG
```

```
11    status=$?
12    wait
13    if [ $status -ne 0 ]; then
14       printf  "Failed to start $a with code: $status\n" 2>&1 | tee -a $LOG
15       exit $status
16    fi
17 done
```

Im Ordner "/utils/services" innerhalb des Containers befinden sich die einzelnen Skripte, pro Prozess eines. Auf dieser Art könnten beliebig weitere Prozesse im selben Container gestartet werden.

**Startskript für Node.js** Dieses Skript führt den Webshop aus.

```
1 #!/bin/ash
2 exec node /app/bin/www
```

**Startskript für MongoDB** Dieses Skript führt die Datenbank aus, verlangt von einer Verbindung eine Authentifizierung und setzt dabei den Log- und Datenpfad.

```
1 #!/bin/ash
2 DBPath="/data/db"
3 MongoLogPath="/var/log/mongodb/mongo.log"
4
5 mongod --dbpath $DBPath --fork --logpath $MongoLogPath --auth
```

**MongoDB Import Skript** Die Daten werden während des Build-Prozesses importiert. Das Skript startet MongoDB, erstellt die Datenbank mit Benutzer und Passwort und importiert alle vorhandene JSON Daten aus dem Ordner "data" in die Datenbank.

```
1 #!/bin/ash
2 DataFilePath="/app/data"
3 DBPath="/data/db"
4 MongoLogPath="/var/log/mongodb/mongo.log"
5 DBName="webshop"
6
7 #Create MongoDB User and Roles
8 mongod --dbpath $DBPath --fork --logpath $MongoLogPath --noauth && \
9 mongo < /app/configure.js  && \
10
11 #Import all JSON Files
12 ls $DataFilePath/*.json | xargs -n1 basename | sed 's/.json$//' | while read col; do
13    mongoimport -d $DBName -c $col --file $DataFilePath/$col.json --jsonArray;
14 done && \
15
16 #Shutdown clean
17 mongod --shutdown
```

### 5.3.5 Dockerfile

Aus den vorhergegangen Überlegungen und Skripts resultiert folgendes Dockerfile. Mögliche Optionen werden im Anhang *B Benutzerhandbuch* erklärt.

```
# ---- Base Stage ----
FROM node:alpine as base
#Define where our app lives
WORKDIR /app

# Install node_modules
COPY package.json yarn.*lock ./
RUN yarn install --production=true --non-interactive

#Install MongoDB, create Data Directory and set Permissions for User Node
RUN apk --no-cache add \
mongodb \
mongodb-tools && \
mkdir -p /data/db

# Add temporary necessary docker assets and import DB
COPY data ./data
COPY docker_config/configure.js .
COPY docker_config/run_all /utils/scripts/
COPY docker_config/mongodb_import /utils/scripts/
COPY docker_config/mongodb /utils/services/
COPY docker_config/node /utils/services/
RUN chmod -R 755 /utils && /utils/scripts/mongodb_import
RUN rm -rf /data/db/journal/

# ---- Release Stage ----
FROM node:alpine AS final
#Define where our app lives
WORKDIR /app

# Set variables for production mode and turning on/off RCE and Serialization Bugs
ENV NODE_ENV production
ENV NODE_RCE_EVAL off
ENV NODE_RCE_SERIALIZATION off

# Install MongoDB
RUN apk --no-cache add mongodb

# Copy node_modules, db and scripts to final image
COPY --from=base /app/node_modules ./node_modules
COPY --from=base /data/db /data/db
COPY --from=base /utils /utils

# Bundle app source code and delete not needed docker assets
COPY . .
RUN rm -rf ./dockerfile  ./docker_config ./data

# Expose this Docker on following ports to the outside world
EXPOSE 80 443

# Set our run_all script as entrypoint
ENTRYPOINT ["/utils/scripts/run_all"]
```

**Multi-Stage Builds** Das Dockerfile ist in zwei Teile aufgeteilt. In der "Base Stage" werden alle Abhängigkeiten geladen und die Datenbank mit Daten abgefüllt. In der "Release Stage" werden die Datenbank und vorher erstellten Abhängigkeiten in ein sauberes Image kopiert, um das finale Image möglichst klein zu halten. Zum Schluss wird der komplette Source Code des Webshops in das Image kopiert.
Ein Vorteil der Trennung der Erstellung von Abhängigkeiten und kopieren des Source Codes ist das Caching von Docker. Solange nur der Source Code verändert wird und keine Änderungen an den Abhängigkeiten vorgenommen werden, bleibt das Caching intakt. Somit wird nur der Code neu kopiert und die Erstellung des Images ist in wenigen Sekunden erledigt.

**Mögliche Erweiterungen** Während der Arbeit wurde nebst der "Base Stage" und "Release Stage" eine weitere Build-Stage angedacht, welche richtig eingesetzt nützlich ist. Dabei werden in einem Zwischenschritt (Test Stage) zuerst alle Tests ausgeführt und nur wenn alle ohne Probleme ausgeführt worden sind, wird am Schluss die "Release Stage" gestartet. Dies ist vor allem nützlich, wenn die Applikation ohne ein Continuous Integration/Delivery (CI) System entwickelt wird. Somit wird der Entwickler gezwungen, die Tests auszuführen, bevor das Image erstellt wird. Der einzige Nachteil ist, dass sich die Build Zeit natürlich entsprechend der Laufzeit der Tests verlängert, wobei auch kein Caching nützt.
Aufgrund dieses Nachteils und weil im Zuge dieser Arbeit ein CI verwendet wurde, welches das Testing übernimmt, wurde dies ausgelassen.

## 5.4 Usability-Tests der Aufgabenstellungen

Aus den Tests mit den Testpersonen sind hauptsächlich folgenden Punkte aufgetaucht:

- Für die Aufgaben notwendige Programme konnten nicht gefunden werden.
- Die Musterlösung war zum Teil ungenügend beschrieben und führte zu Verwirrung.

Aufgrund des Feedbacks sind die Musterlösungen noch detaillierter und zusätzlich mit Notizen ergänzt worden, sodass zum Beispiel ersichtlich ist, wo sich die zu benutzenden Programme befinden. Die Notizen wurden zudem verwendet, um den Benutzer darauf hinzuweisen, dass gewisse auszuführende Befehle kein Feedback an den Benutzer zurückliefern und somit nicht auf diese gewartet werden muss.

# 6 Ergebnisdiskussion und Ausblick

In diesem Kapitel wird auf die Ergebnisse der Arbeit und die aufgetauchten Probleme eingegangen. Zudem werden potentielle Punkte für die Weiterentwicklung aufgeführt.

## 6.1 Abweichungen vom Projektplan

Während des Projektes kam es zu wenigen Problem und der wöchentliche Fortschritt lag jeweils im Plansoll oder darüber. Aus diesen Gründen konnte der Projektplan nicht nur eingehalten, sondern in einigen Bereichen gar vorgezogen werden. So wurde der Meilenstein 4 "Verwundbarkeiten", bei welchem geplant war, 60% aller Schwachstellen implementiert zu haben, auf 80% erweitert. Die zweite Änderung betraf den Meilenstein 5 "Code Freeze", dieser wurde um eine Woche vor verschoben. Ansonsten stellte sich die Planung des Projektes als realistisch heraus.

## 6.2 Probleme

Es folgen die während des Projektes aufgetauchten Probleme, sowie die gefundenen Lösungen dazu.

### 6.2.1 Unterschätzen des Aufwandes für Schwachstellen

Während der Projektdurchführung hat sich herauskristallisiert, dass sich die normale Softwareentwicklung von der Implementation absichtlicher Schwachstellen unterscheidet. Das Thema Sicherheit wurde in den letzten Jahren auch im Bereich Web immer wichtiger und somit wurde auch bei der Entwicklung der Frameworks besonderes Augenmerk darauf gelegt. Daher kam es beim Entwicklungsprozess vor, dass Software, welche schon im Auslieferungszustand gewisse Sicherheitsprobleme beseitigt, bewusst manipuliert werden musste, um eine Schwachstelle überhaupt ausnutzen zu können. Dies hat zum Teil dazu geführt, dass bei ausgedachten Szenarien mehrfach nachgebessert werden musste.

### 6.2.2 Hacking-Lab LiveCD: TLS Listener

Alle Aufgaben lassen sich mit der Hacking-Lab LiveCD lösen. Ein Teil dieser LiveCD besteht aus einem Apache Webserver, welcher ein Angreifer als seinen Landing-Page Server benutzen kann, um Daten des Opfers zu exfiltrieren.
Während der Entwicklung wurde der Glockenshop auf TLS umgerüstet. Aufgrund der Konfiguration des "Pre-Authentication" Moduls des Landing-Page Servers, kam es danach zu Problemen. Das Problem hätte mit der Änderung von Konfigurationsdateien gelöst werden können. Weil dies zur Folge hätte, dass die LiveCD aktualisiert werden muss, wurde diese Idee verworfen. Alternativ wurde ein Weg gefunden einen TLS Listener mittels "openSSL" einzurichten, was für die Aufgaben genügt.

### 6.2.3 Zertifikate des Angreifers

Damit zum Beispiel ein JWT Token eines Benutzers mittels einer XSS Attacke exfiltriert werden kann, muss ein Remotehost als Ziel vorhanden sein. Da der Glockenshop neu nur über TLS erreichbar ist, muss auch der Remotehost des Hackers über ein gültiges TLS Zertifikat verfügen. Wird versucht die Daten an einen Webserver ohne TLS zu senden, wird diese Verbindung wegen "Mixed Active Content" vom Browser blockiert.
Dies ist nicht weiter problematisch für Benutzer der Challenge, welche über einen eigenen Webserver und gültige Zertifikate verfügen. Aber sobald ein Nutzer die Aufgabe mittels LiveCD löst, hat dieser das Problem ein gültiges Zertifikat besitzen zu müssen um die Musterlösungen durchzuspielen.
Dieses Problem kann auf zwei Arten umgangen werden:
Entweder erstellt der Nutzer mithilfe der LiveCD ein eigenes Zertifikat für den TLS Listener und importiert dieses anschliessend in den Trusted Store, oder er benutzt als Zertifikat das auf der LiveCD vorinstallierte. Dieses ist zwar für den Landing-Page Server gedacht, kann aber für diesen Zweck genutzt werden. Weil dieses bereits im Trusted Store des Browsers importiert ist, kommt es zu keinem Fehler.

## 6.3 Vergleich mit der bisherigen Lösung

Im Vergleich mit der ursprünglichen Lösung des Glockenshops hat sich vieles verändert. Der Shop erhielt ein visuelles Facelifting und wurde mithilfe moderner Webtechnologien von Grund auf neu entwickelt. Nicht nur die Bedienung wurde rundum erneuert, auch wurde mithilfe verschiedener Technologien Schwachstellen eingebaut, welche zuvor noch nicht im Glockenshop vorhanden waren.
Die Codebasis wurde sauber aufgeteilt und es wurde möglichst viel Wert auf wiederverwendbaren Code gelegt. So wurde mit dem finalen Produkt die Basis für einen modernen Glockenshop geschaffen, der leicht weiterzuentwickeln ist. Diese Wartbarkeit war einer der grossen Nachteile der bestehenden Lösung.

## 6.4 Veränderung der Codequalität

Während des gesamten Projektes wurde die Codequalität automatisch gemäss den Vorgaben im Kapitel **??** *Projektplan* geprüft. Die Entwicklung der Qualität während des Projektes wird in diesem Abschnitt diskutiert.

### 6.4.1 Auswertung zum Projektstart

Zu Beginn des Projektes wurde der Code der Vorgängerarbeit analysiert, um mögliche Mängel aufzudecken.

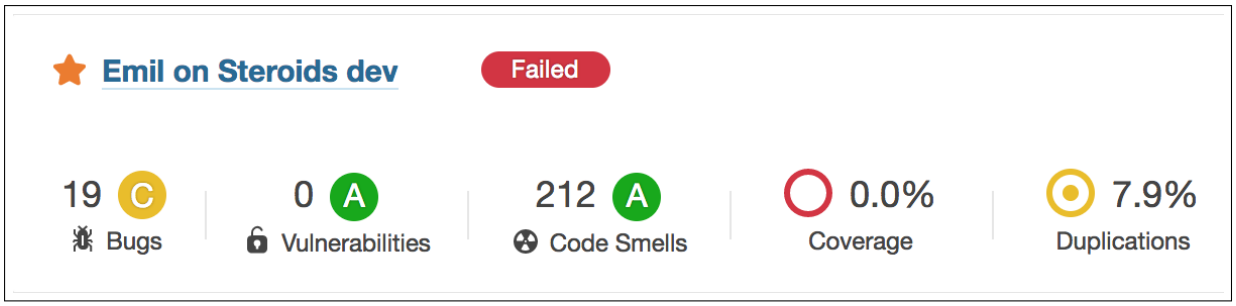


Abbildung 6.1: SonarCloud Auswertung zum Projektbeginn

Nebst der Codeduplikation waren die meisten Probleme mit relativ kleinem Aufwand zu beheben und vor allem im weiteren Entwicklungsprozess zu vermeiden.

### 6.4.2 Auswertung zum Projektende

Der folgende Stand der Codequalität reflektiert den Stand zum Projektende.

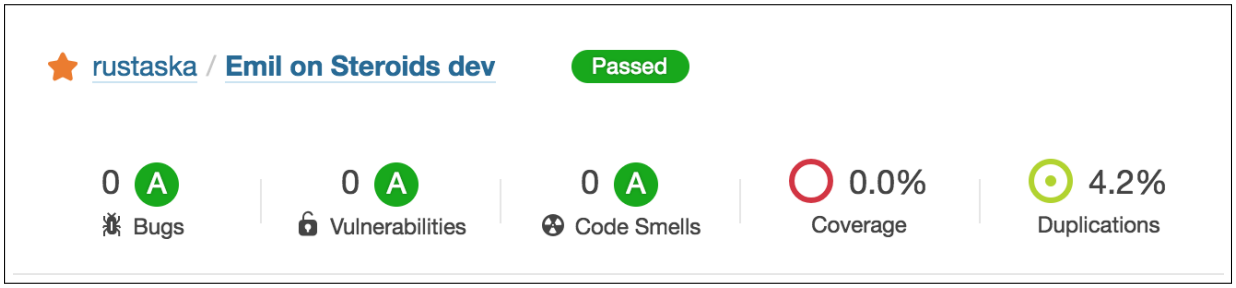


Abbildung 6.2: SonarCloud Auswertung zum Projektendes

Wie zu sehen ist, wurden nicht nur alle Probleme, welche die statische Codeanalyse im Vorfeld aufgedeckt hat behoben, sondern auch der restliche Code möglichst sauber entwickelt. Eine statische Codeanalyse ist natürlich kein Garant für Fehlerfreiheit, aber als Instrument für die Überprüfung der Codequalität durchaus nützlich.

## 6.5 Ausblick

In diesem Abschnitt wird behandelt, in welchen Bereich die Applikation in Zukunft weiterentwickelt werden kann.

### 6.5.1 Design

Das Design könnte überarbeitet werden und sich dabei je nachdem noch mehr Responsive verhalten. Der Glockenshop ist zwar nicht für mobile Geräte gedacht, wäre aber einfach damit zu ergänzen.
Entwickelt wurde das Design hauptsächlich während der Vorgängerarbeit. Zu dieser Zeit war Bootstrap in der Version 3.3.7 aktuell. Mittlerweile ist Version 4 verfügbar, aber leider nicht ohne weiteres aktualisierbar. Dafür benötigt es an diversen Stellen Anpassungen.

### 6.5.2 AngularJS

Als Vorgabe der Vorgängerarbeit galt AngularJS, dessen letzte Version 1.6.7 im Jahre 2017 erschienen ist. Bereits im Jahre 2016 erschien Angular 2 und wird seither stetig weiterentwickelt. Damit der Glockenshop aktuell bleibt, könnte Angular auf die neuste Version aktualisiert werden. Dies würde allerdings bedeuten, dass im Frontend viele Anpassungen nötig wären, um den Code nach den neuen Richtlinien umzuschreiben.

### 6.5.3 Mehr Schwachstellen

Viele Schwachstellen wurden bereits eingebaut. Sollen neue Schwachstellen eingebaut werden, können diese dank der aufgeräumten Codestruktur einfach hinzugefügt werden.

### 6.5.4 Weitere Technologien

Die Integration von weiteren Technologien in die Applikation ermöglicht es, technologiespezifische Lücken einzubauen.

### 6.5.5 Unit Tests

Wie bereits in der Analyse im Kapitel *4.7 Testing* beschrieben, wird der Shop im Moment nur mittels E2E Tests getestet. Diese haben gegenüber Unit Tests zwei Vorteile. Zum einen wird das Frontend getestet und die Funktionen so über allen Schichten hinweg überprüft, was mehr dem realen Umfeld entspricht. Zum anderen bietet dies die Möglichkeit, alle Schwachstellen automatisiert zu testen, was mit Unit Tests nicht möglich wäre.
Trotzdem würden Unit Tests im Backend dem Entwicklungsprozess helfen, denn deren Durchführung benötigt weniger Zeit. Des Weiteren wäre so die Ermittlung der Code-Coverage, einem Instrument zur Prüfung der Softwarequalität, möglich.

### 6.5.6 Mehr Testdaten

Der Webshop wird mit Testdaten geliefert, sodass beim Start des Containers der Eindruck entsteht, der Shop wäre in Benutzung. Der Realismus könnte weiter gesteigert werden, indem für alle Funktionen des Shops mehr Testdaten vorhanden wären.

### 6.5.7 Dritte Docker Build Stage

Wie im Abschnitt *5.3.5 Mögliche Erweiterungen* nachzulesen ist, wäre eine weitere Build-Stage für das Dockerfile möglich. Diese bietet vor allem Vorteile in einer Entwicklungsumgebung ohne CI System, indem alle Tests bei jedem Docker Build-Vorgang ausgeführt werden.

## 6.6 Erreichte Ziele und Fazit

In diesem Abschnitt werden die aus der Aufgabenstellung erwarteten Resultate mit den Ergebnissen der Arbeit abgeglichen. Ebenfalls wird auf die in dieser Arbeit zu erledigenden Arbeiten eingegangen, welche in der Analyse, im Kapitel *4.10 Zusammenfassung*, erarbeitet wurden.

### 6.6.1 Dockerfile

Es wurde ein Dockerfile mit Multi-Stage Builds erstellt, welches alle Abhängigkeiten des Webshops im Cache hält und somit äusserst effizient arbeitet. Dabei wurde sehr viel Wert auf Geschwindigkeit und Optimierung der Grösse des Docker Images gelegt.

### 6.6.2 Beschreibung von Aufgabenstellungen und Musterlösungen

Während der Arbeit sind 16 Schwachstellen in den Webshop eingebaut worden. Zwei dieser Schwachstellen flossen, den Szenarien geschuldet, in die restlichen 16 ein. Somit wurden für 14 Lücken Aufgabenstellungen und entsprechende Musterlösungen geschrieben. Zudem wurde pro Aufgabe dokumentiert, wieso diese Sicherheitslücke überhaupt möglich ist und wie diese zu vermeiden wäre.

### 6.6.3 Webanwendung

Die Webanwendung wurde so umgesetzt, dass sich die Bedienung wie ein normaler Shop anfühlt. Der Source Code der Vorgängerarbeit wurde an vielen Stellen überarbeitet, von Problemen befreit und um viele neue Funktionen ergänzt. Diese Funktionen dienen als Grundstein für die neu implementierten Schwachstellen. Als Technologien wurden Node.js, AngularJS, Express, Docker und MongoDB eingesetzt, was einem modernen Technologiestack entspricht.

### 6.6.4 Komplette Software Dokumentation

Die Arbeit wurde in allen Bereichen dokumentiert. Es sind sowohl Domain Modell, Use Cases, Deployment Diagramme und weitere Software Architektur Dokumente erstellt worden. Des Weiteren entstand ein Benutzerhandbuch, welches es Drittpersonen ermöglicht, den Webshop weiterzuentwickeln und neue Docker Images zu erstellen.

### 6.6.5 Testing der Schwachstellen

Nebst dem Webshop wurde ein Testing Framework integriert. Dieses ermöglicht es, nicht nur den Webshop, sondern auch alle Schwachstellen automatisiert zu testen. Dies wurde vom Auftraggeber gewünscht, um Breaking-Changes bei der Weiterentwicklung zu vermeiden.

### 6.6.6 Fazit

Im Bezug auf die Aufgabenstellung wurden alle Vorgaben umgesetzt. Die am Anfang des Projektes priorisierten Schwachstellen konnten früher als geplant fertig gestellt werden. Aus diesem Grund blieb Zeit für die Implementation weiterer Lücken und Verbesserungen an der Bedienbarkeit des Webshops, was dem ganzen Projekt zugute kam.

# 7 Glossar

| Abkürzung | Beschreibung |
|---|---|
| API | Das Application Programming Interface ist ein Programmteil, welcher Softwaresystemen anderer Programme zur Anbindung an das System zur Verfügung gestellt wird. |
| CD | Compact Disc ist die Bezeichnung für einen optischen Speicher. |
| CI | Continuous Integration wird in der Software Entwicklung genutzt und beschreibt den Prozess des fortlaufenden Zusammenfügens von Komponenten zu einer Anwendung. |
| CORS | Cross-Origin Resource Sharing ermöglicht Webclients Cross-Origin-Requests. Diese Zugriffe werden üblicherweise durch die Same-Origin-Policy unterbunden. |
| CSRF | Cross-Site-Request-Forgery ist ein Angriff auf eine Webanwendung, bei dem der Angreifer eine Transaktion im Namen eines anderen Benutzers ausführt. |
| CSWSH | Cross-Site WebSocket Hijacking ist ein Angriff, welcher über eine bösartige Webseite eine WebSocket Verbindung zum Server aufbaut. Dies ist möglich, da WebSockets standardmässig nicht der Same-Origin-Policy unterliegen. |
| CVV | Card Verification Value ist eine Sicherheitsmechanismus der Kreditkarte. Dieser soll die Nutzung von gefälschten oder gestohlenen Kreditkarten mühevoller machen. |
| DOM | Das Document Object Model ermöglicht die Darstellung von HTML- oder XML-Dokumenten als Baumstruktur. |
| E2E | Als End-to-End wird ein Produkt oder Dienstleistung bezeichnet, welche im Gesamten betrachtet wird. |
| ES | Hierbei handelt sich um die JavaScript Standarisierung mit dem Namen ECMAScript. |
| FAQ | Frequently asked questions sind aufgelistete Fragen mit dazugehörigen Antworten, welche an ein System immer wiederkehrend auftauchen. |
| GUI | Bei dem Graphical User Interface handelt sich um Benutzerinterface mit welchem Benutzer über grafische Symbole interaktiv agieren. |
| HTML | Bei der Hypertext Markup Language handelt sich um die Standard-Markup-Sprache für die Erstellung von Webseiten. |
| HTTP | Das Hypertext Transfer Protocol wird genutzt um Daten über die Anwendungsschicht zu übermitteln. Hierbei handelt sich um ein zustandloses Protokoll, welches hauptsächlich zum Laden von Webseiten im Webbrowser genutzt wird. |

| | |
|---|---|
| HTTPS | Das Kommunikationsprotokoll Hypertext Transfer Protocol Secure wird zur abhörsicheren Übertragung von Daten genutzt. |
| ID | Der Identifikator wird zur eindeutigen Bestimmung eines Objekts definiert. |
| IDOR | Die Insecure Direct Object Reference ist eine Schwachstelle, welche von Angreifern ausgenutzt werden kann. Hierbei wird auf ein Objekt über die ID zugegriffen. Die Applikation macht jedoch keine Prüfung, ob es sich um einen autorisierten Benutzer für dieses Objekt handelt. |
| IP | Das Internet Protokoll ist eine Netzwerkprotokoll und bildet die Grundlage des Internets. |
| JSON | Die JavaScript Object Notation ist ein Dateiformat zum Datenaustausch zwischen Anwendungen. Es besitzt die Eigenschaft leicht lesbar zu sein. |
| JWT Token | Steht für JSON Web Token und ist eine Möglichkeit der Nutzerauthentifizierung. |
| LDAP | Das Netzwerkprotokoll Lightweight Directory Access Protocol wird zur Abfrage und Änderung von Informationen in verteilten Verzeichnisdiensten genutzt. |
| LTS | Long-term Support ist eine Version einer Anwendung, welche einen längeren Zeitraum an Support besitzt. |
| MEAN Stack | Abkürzung für MongoDB, Express, AngularJS und Node.js. |
| NoSQL | NoSQL ist eine Bezeichnung für Datenbanken, welche keinen relationalen Ansatz nutzen. |
| NPM | Bei NPM handelt es sich um ein Packet Manager für JavaScript. |
| OWASP | Das Open Web Application Security Project ist eine Non-Profit-Organisation mit dem Ziel, die Sicherheit von Anwendungen und Diensten im World Wide Web zu verbessern. |
| OWASP Top 10 | Eine Liste mit den 10 grössten sicherheitskritischen Risiken für Webapplikationen. |
| PEM | Hierbei handelt es sich um das Standard-Format für OpenSSL und viele andere SSL Tools. |
| RCE | Die Remote Code Execution bietet Angreifer die Option auf Computer zuzugreifen. Das System kann somit von der Ferne manipuliert werden. |
| REST | Representational State Transefer soll die Kommunikation zwischen Computersystemen im Web erleichtern. Es handelt sich um einen Architekturstil, welcher Standars bereitstellt. |
| RUP | Der Rational Unified Process enthält Vorhersagemodelle zur Softwareentwicklung und die entsprechenden Programme. |

| | |
|---|---|
| SOP | Same-Origin-Policy ist ein Konzept, welches einer Seite untersagt auf Objekte zuzugreifen, welche von einem anderen Ort stammen als sie selbst. |
| SSJS | Bei dieser Verwundbarkeit wird darauf abgezielt, dass JavaScript Code direkt auf dem Server ausgeführt werden kann. |
| SSL | Bei SSL handelt es sich um einen standardisierte verschlüsselte Verbindung zwischen Webserver und Browser. |
| SSRF | Bei der Attacke Server-Side Request Forgery hat der Angreifer die Möglichkeit einen selbst erstellten Request vom Server aus zu senden. |
| SVG | Eine Scalable Vector Graphic ist eine Spezifikation, welche zur Beschreibung von zweidimensionalen Vektorgrafiken dient und basiert auf XML. |
| TLS | Die Transport Layer Security ist ein hybrides Verschlüsselungs-Protokoll zur Datenübertragung. |
| UC | In einem Use Case werden alle Szenarien, welche ein Akteur an einem System durchführen kann dargestellt. |
| URL | Als Uniform Resource Locator wird die Adresse einer Website bezeichnet. |
| VPN | Die Bezeichnung VPN wird für ein virtuelles privates Netzwerk genutzt. |
| XHR | XMLHttpRequest wird zur Übertragung von Daten über das HTTP-Protokoll mittels JavaScript genutzt. |
| XSS | Als Cross Site Scripting Verwundbarkeit wird die Sicherheitslücke bezeichnet, wenn eine dynamisch generierte Anwendung JavaScript Code entgegennimmt und bei Aufruf eines anderen Benutzers ausgeführt wird. |
| ZAP | Beim Zed Attack Proxy handelt sich um eine Penetrationtest-Anwendung für Webseiten. |

# A Use Cases

In diesem Kapitel sind die einzelnen Use Cases in der "Fully dressed" Notation inklusive Use Case Diagramme beschrieben.

## A.1 Webshop

In diesem Abschnitt wird der Webshop beschrieben aus Kundensicht beschrieben.

### A.1.1 Diagramm

Das Diagramm enthält die zwei Aktoren "unregistrierter Benutzer" und "registrierter Benutzer". Die während dieser Arbeit neu entstandenen Use Cases wurden blau markiert.

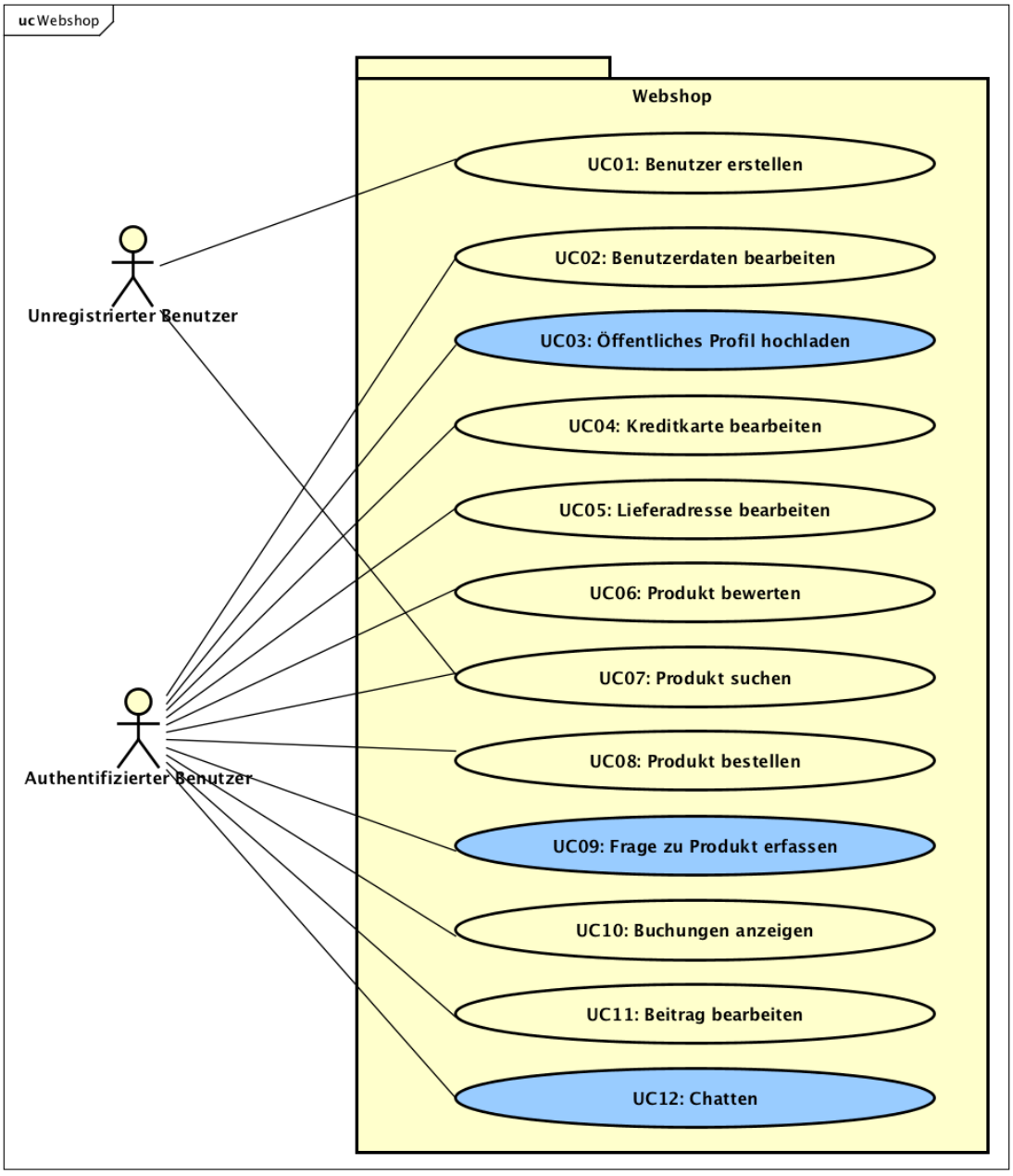


Abbildung A.1: Use Case Diagramm Webshop

### A.1.2 Use Cases Fully dressed

Nachfolgend sind die bereits erwähnten Uses Cases im Fully dressed Format aufgeführt.

**UC01: Benutzer erstellen**

| Primary Actor | Unregistrierter Benutzer |
|---|---|
| Beschreibung | Ein unregistrierter Benutzer kann mit einer Email Adresse, einem Passwort und seinen persönlichen Angaben einen neuen Benutzer erstellen. |
| Stakeholders | Unregistrierter Benutzer:<br>• Möchte einen Account erstellen. |
| Preconditions | • Der Benutzer darf nicht bereits existieren.<br>• Der Benutzer darf nicht eingeloggt sein. |
| Postconditions | • Der Benutzer hat ein Konto eröffnet und kann sich nun anmelden. |
| Main Success Story | 1. Der Benutzer gibt die geforderten Angaben ein.<br>2. Das System validiert die eingegebenen Angaben.<br>3. Die Daten für den User Account werden gespeichert und der Benutzer über den Erfolg informiert. |
| Alternative Flows | 2a. Die Validierung schlägt fehl.<br>1. System meldet ungültige Angaben mit Begründung.<br>2. Benutzer gibt Daten erneut ein und das System validiert.<br>*Wiederhole Schritt 1-2 bis die Angaben valide sind.* |

Tabelle A.1: Fully Dressed UC01

**UC02: Benutzerdaten bearbeiten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Ein authentifizierter Benutzer kann seine persönlichen Daten anpassen. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte seine persönlichen Benutzerdaten anpassen. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Persönliche Daten korrekt angepasst. |
| Main Success Story | 1. Vorname, Nachname anpassen.<br>2. Benutzerdaten speichern. |

Tabelle A.2: Fully Dressed UC02

### UC03: Öffentliches Profil hochladen

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Jeder authentifizierte Benutzer kann ein persönliches Profil hochladen, welches für die restlichen Benutzer einsehbar ist. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte sein öffentliches Profil erstellen. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Profil ist korrekt hochgeladen und für anderer Benutzer ersichtlich. |
| Main Success Story | 1. Beispiel Template herunterladen.<br>2. Template anpassen.<br>3. Template hochladen. |

Tabelle A.3: Fully Dressed UC03

**UC04: Kreditkarte bearbeiten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Ein authentifizierter Benutzer kann bis zu drei Kreditkarten hinzufügen. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte seine Kreditkarten bearbeiten. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Kreditkarten korrekt angepasst. |
| Main Success Story | 1. Kreditkarte Übersicht öffnen.<br>2. Kreditkarte hinzufügen.<br>3. Kreditkartennummer eintragen.<br>4. Kreditkarten Typ auswählen.<br>5. Kreditkarten CVV eintragen.<br>6. Kreditkarten Ablauf-Monat wählen.<br>7. Kreditkarten Ablauf-Jahr wählen.<br>8. Kreditkarte speichern.<br>*Schritt 1-8 wiederholen bis drei Kreditkarten erfasst sind.* |
| Alternative Flows | 2a. Kreditkarte anpassen.<br>1. Kreditkarte anpassen.<br>2. Kreditkartennummer ändern.<br>3. Kreditkarten Typ ändern.<br>4. Kreditkarten CVV ändern.<br>5. Kreditkarten Ablauf-Monat ändern.<br>6. Kreditkarten Ablauf-Jahr ändern.<br>*Gehe zu Main Success Story Schritt 1.*<br>2b. Kreditkarte löschen.<br>*Main Success Story endet.* |

Tabelle A.4: Fully Dressed UC04

**UC05: Lieferadresse bearbeiten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Ein authentifizierter Benutzer kann bis zu drei Lieferadressen hinzufügen. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte seine Lieferadressen bearbeiten. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Lieferadressen korrekt angepasst. |
| Main Success Story | 1. Lieferadressen Übersicht öffnen<br>2. Lieferadresse hinzufügen.<br>3. Strasse, Postleitzahl, Stadt und Land erfassen<br>4. Lieferadresse speichern.<br>*Schritt 1-4 wiederholen bis drei Lieferadressen erfasst.* |
| Alternative Flows | 2a. Lieferadresse anpassen.<br>1. Strasse, Postleitzahl, Stadt und Land ändern<br>2b. Lieferadresse löschen.<br>*Gehe zu Main Success Story Schritt 1.* |

Tabelle A.5: Fully Dressed UC05

**UC06: Produkt bewerten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Jeder authentifizierte Benutzer kann einem im Webshop enthaltenen Produkt eine Bewertung abgeben. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte ein Produkt bewerten. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Produkt ist korrekt bewertet. |
| Main Success Story | 1. Eine Bewertung von 1 bis 5 Sternen auswählen.<br>2. Kommentar zu der Bewertung abfüllen.<br>3. Bewertung speichern. |

Tabelle A.6: Fully Dressed UC06

**UC07: Produkt suchen**

| | |
|---|---|
| Primary Actor | Unregistrierter Benutzer |
| Secondary Actor | Authentifizierter Benutzer |
| Beschreibung | Der Webshop kann nach passenden Produkten durchsucht werden, dabei gibt es mehrere Möglichkeiten die Produkte zu selektieren. |
| Stakeholders | Unregistrierter Benutzer:<br>• Möchte Produktkatalog durchsuchen können.<br>Unregistrierter Benutzer:<br>• Möchte Produktkatalog durchsuchen können. |
| Preconditions | • keine |
| Postconditions | • Ansicht muss die gefilterten Produkte enthalten. |
| Main Success Story | 1. Benutzer navigiert zu Shop.<br>2. Benutzer durchsucht alle Produkte durch scrollen.<br>3. Benutzer sieht sich Produkt durch klicken an. |
| Alternative Flows | 2a. Benutzer klickt auf eine Kategorie.<br>2b. Benutzer nutzt Suchfunktion.<br>2c. Benutzer nutzt Sortierfunktion. |

Tabelle A.7: Fully Dressed UC07

**UC08: Produkt bestellen**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Ein Benutzer oder authentifizierter Benutzer kann im Webshop Produkte dem Warenkorb hinzufügen. Der authentifizierte Benutzer ist nun in der Lage, die Produkte des Warenkorbs zu bestellen. Der Bestellprozess beinhaltet eine Übersicht der bestellten Artikel, danach folgt die Auswahl der Lieferadresse und der Zahlungsmethode. Die Bestellung kann am Ende dieses Prozesses abgeschlossen werden. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte ein oder mehrere Produkte bestellen. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Bestellprozess erfolgreich abgeschlossen.<br>• Bestellung in der Bestellübersicht ersichtlich. |
| Main Success Story | 1. Produkt dem Warenkorb hinzufügen.<br>*Schritt 1 wiederholen bis gewünschte Produkte im Warenkorb.*<br>2. Warenkorb auschecken.<br>3. Bestellübersicht verifizieren.<br>4. Lieferadresse auswählen.<br>5. Zahlungsmethode wählen.<br>6. Bestellung abschliessen. |

Tabelle A.8: Fully Dressed UC08

**UC09: Frage zu Produkt erfassen**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Ein authentifizierter Benutzer kann im Webshop in der Detailansicht des Produkts Fragen zu diesem Produkt stellen. Diese Fragen sind auch für anderer Benutzer ersichtlich. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte eine Frage zu einem Produkt erfassen. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Frage zum Produkt korrekt gespeichert. |
| Main Success Story | 1. Detailansicht eines Produkts öffnen.<br>2. Frage erfassen.<br>3. Frage speichern. |

Tabelle A.9: Fully Dressed UC09

**UC10: Buchungen anzeigen**

| Primary Actor | Authentifizierter Benutzer |
|---|---|
| Beschreibung | Ein authentifizierter Benutzer kann seine bestellten Produkte einsehen und exportieren. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte seine Buchungen ansehen und exportieren. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Buchungen sind korrekt dargestellt.<br>• Buchungen werden korrekt exportiert. |
| Main Success Story | 1. Buchungsübersicht öffnen.<br>2. From und Range ausfüllen.<br>3. Buchungen exportieren. |

Tabelle A.10: Fully Dressed UC10

**UC11: Beitrag bearbeiten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Jeder authentifizierte Benutzer kann im Webshop einen neuen Beitrag erfassen. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte einen Beitrag veröffentlichen. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Beitrag erfolgreich erstellt.<br>• Beitrag ist auf der Home-Seite ersichtlich. |
| Main Success Story | 1. Im Menü den Punkt Community auswählen.<br>2. Titel und Text erfassen.<br>3. Ein Bild aus dem Filesystem auswählen.<br>4. Beitrag speichern. |
| Alternative Flows | 3a. Ein Bild von einer URL auswählen. |

Tabelle A.11: Fully Dressed UC11

**UC12: Chatten**

| | |
|---|---|
| Primary Actor | Authentifizierter Benutzer |
| Beschreibung | Jeder authentifizierte Benutzer kann mit anderen authentifizierten Benutzern Nachrichten austauschen. |
| Stakeholders | Authentifizierter Benutzer:<br>• Möchte mit einem anderen Benutzer chatten. |
| Preconditions | • Benutzer muss authentifiziert sein. |
| Postconditions | • Nachricht erfolgreich gesendet.<br>• Nachricht erfolgreich erhalten. |
| Main Success Story | 1. Chat öffnen.<br>2. Benutzer auswählen.<br>3. Nachricht erfassen.<br>4. Nachricht senden. |

Tabelle A.12: Fully Dressed UC12

## A.2 Hacking

In diesem Abschnitt werden die Use Cases des Hackers beschrieben.

### A.2.1 Diagramm

Die implementierten Schwachstellen sind gleichzeitig die Use Cases des Aktors "Hacker". Mit blau markiert wurden die während dieser Arbeit neu integrierten Schwachstellen.

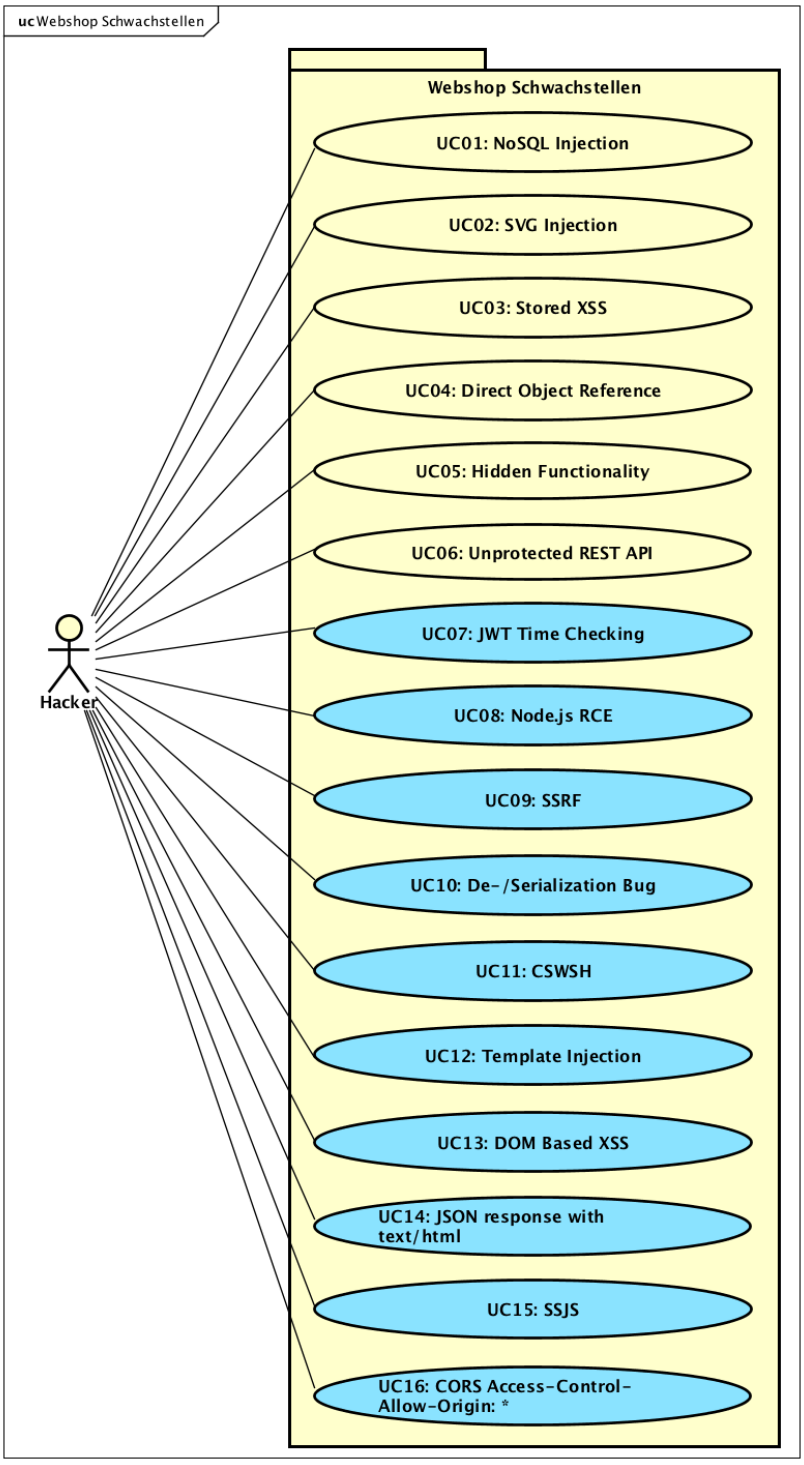


Abbildung A.2: Use Case Diagramm Hacking

### A.2.2 Use Cases Fully dressed

Da es bei den Use Cases des Hackers immer um darum geht, Sicherheitslücken zu suchen und auszunutzen, macht es keinen Sinn diese auszuschreiben. Im Kapitel **??** *Aufgabenstellungen und Musterlösungen* werden diese genau beschrieben.

# B Benutzerhandbuch

## B.1 Docker

Ist eine Änderung an der Konfiguration des Docker Containers oder ein Update des Webshop Source Codes nötig, muss das Docker Image neu erstellt werden. Diese Anleitung beinhaltet alle Schritte dafür. Um lediglich ein fertiges Image auszuführen, kann direkt bei Schritt *B.1.4 Docker Run* begonnen werden.

### B.1.1 Voraussetzungen

- Installierte Docker Version >= 17.05 (Für Multi-Stage Builds)

### B.1.2 Dockerfile Konfiguration

Im Dockerfile sollten, wenn überhaupt, nur zwei Konfiguration nötig sein:

**Veröffentlichte Ports anpassen**
Werden die Ports, auf welchen der Webshop läuft, in dessen Source Code verändert, müssen diese ebenfalls im Dockerfile angepasst werden. Dies dient der guten Dokumentation und ist mit der Anpassung des Docker Befehls EXPOSE möglich:

```
1 EXPOSE 80 443
```

**Standardmässiges Verhalten der RCE / Serialization Bugs**
Die zwei Lücken "RCE" und "Serialization Bug" ermöglichen es dem User, vollen Systemzugriff zu erlangen. Diese können per Umgebungsvariablen an- oder ausgeschaltet werden. Die Standardkonfiguration dieser Lücken ist "off" und ist so im Dockerfile hinterlegt. Dieses Verhalten kann später beim starten überschrieben werden. Siehe dazu Schritt *B.1.4 Docker Run*.
Soll das Standardverhalten angepasst werden, sodass die Lücken bei jedem Start "on" sind, können im Dockerfile die Umgebungsvariablen von "off" auf "on" geschaltet werden:

```
1 ENV NODE_RCE_EVAL on
2 ENV NODE_RCE_SERIALIZATION on
```

### B.1.3 Docker Build

Eine Änderung an der Docker- oder Node.js-Konfiguration zieht zwingend einen neuen Docker Build-Prozesses nach sich. Dazu wird im Ordner, in welchem das "Dockerfile" liegt, folgender Befehl im Terminal ausgeführt:

```
1 docker build . -t hackinglab/emil
```

Für alle Beispiele der Anleitung wurde als Image Name "hackinglab/emil" verwendet. Im Rest der Anleitung wird dieser Name immer wieder benutzt und ist den eigenen Wünschen anzupassen.
Wird während des Build-Prozesses kein Image Name mittels Parameter "-t" angegeben, wird das Image zwar erstellt, ist aber schwer wiederzufinden, da es keinen Namen trägt.

### B.1.4 Docker Run

**Starten des Containers**
Standardmässig läuft der Webshop für HTTP auf Port 80 und HTTPS auf Port 443. Um diese Ports gegenüber dem Host zu veröffentlichen, kann folgender Befehl genutzt werden, bei geänderten Ports verhält sich dies analog:

```
1 docker run -p 80:80 -p 443:443 hackinglab/emil
```

In diesem Beispiel werden die selben Ports für Host und Container verwendet (-p 443:443). Dies ist nicht zwingend, doch muss folgendes **unbedingt** beachtet werden:
Der Webserver leitet HTTP zu HTTPS um. Auf welchen Port umgeleitet wird, hängt von dem im Node.js konfigurierten HTTPS Port ab. Ist im Node.js als HTTPS Port 443 eingestellt und beim Starten des Containers wird dieser auf den Port 5000 veröffentlicht (-p 5000:443), funktioniert der Webshop nicht. Dies, weil Node.js jede Anfrage vom HTTP Port auf den Port 443 weiterleitet, vom Container jedoch nur der Port 5000 veröffentlicht wurde, von welchem der Webserver nichts weiss. Der HTTPS Port muss also auf dem Host und dem Container gleich sein (-p 443:443). Beim HTTP Port spielt dies keine Rolle.

**Starten des Containers mit RCE / Serialization=on**
Um das Standardverhalten der Lücken "RCE" und Serialization Bug" zu ändern, können die involvierten Umgebungsvariablen beim Start des Containers mit dem Parameter "-e" überschrieben werden:

```
1 docker run -p 80:80 -p 443:443 -e NODE_RCE_EVAL=on -e NODE_RCE_SERIALIZATION=on
  ↪ hackinglab/emil
```

Wird der Container nicht als Hintergrundprozess gestartet, ist die Änderung ersichtlich:

```
---------
starting /utils/services/mongodb
about to fork child process, waiting until server is ready for connections
forked process: 13
child process started successfully, parent exiting
---------
starting /utils/services/node
HTTP Server started: http://0.0.0.0:80 in production mode
HTTPS Server started: https://0.0.0.0:443 in production mode
Secret SSRF Server started: http://0.0.0.0:8765 in production mode
NODE_RCE_EVAL: on
NODE_RCE_SERIALIZATION: on
Date: Tue Jun 05 2018 16:14:05 GMT+0000 (UTC)    Type: Info      Message:

Date: Tue Jun 05 2018 16:14:05 GMT+0000 (UTC)    Type: Info      Message:
```

Abbildung B.1: Container mit eingeschalteten Lücken

**Starten als Hintergrundprozess**
Um den Container im Hintergrund zu starten, sodass er das Terminal Fenster nicht blockiert, kann die Option "-d" genutzt werden. Dabei erfolgt keine Ausgabe, ob die Datenbank oder der Webserver erfolgreich gestartet werden konnte.

```
1 docker run -d -p 80:80 -p 443:443 hackinglab/emil
```

**Hintergrundinformationen zu EXPOSE und Docker Run**
Der Befehl EXPOSE im Dockerfile bewirkt nicht, dass diese Ports automatisch gegenüber dem Host geöffnet werden. Dies zeigt einem Nutzer lediglich, auf welchen Ports die Applikation im Container hört und ist somit vor allem für Dokumentationszwecke gedacht. Wird der Container ohne Ports (ohne -p) gestartet, hat der Container keine Konnektivität. Es sind vom aktiven Container jedoch alle Ports, welche unter EXPOSE angegeben wurden, einsehbar:

```
1 docker ps
```

Ausgabe:

```
sh-3.2# docker ps
CONTAINER ID        IMAGE                       PORTS
db1c436b4091        psteinha/emilonsteroids     80/tcp, 443/tcp
```

Abbildung B.2: 'docker ps' ohne veröffentlichten Ports

In diesem Beispiel sind lediglich die Ports, auf welchen im Container gehört wird, aufgelistet, aber kein Mapping gegen den Host.

Um einen dieser Ports gegenüber dem Host zu veröffentlichen, müssen diese beim Starten angegeben werden. Dies ist mittels der Option "-p" im Format "Host Port:Container Port" möglich. So wird angegeben, auf welchem Host Port welcher Container Port erreichbar ist. Der Parameter "-p" kann mehrfach angegeben werden, um mehrere Ports zu veröffentlichen.

Wird nun der Container wieder normal (mit -p) gestartet, kann mit folgendem Befehl erneut die veröffentlichten Ports der Container eingesehen werden:

```
1 docker ps
```

Ausgabe:

```
sh-3.2# docker ps
CONTAINER ID        IMAGE                       PORTS
  NAMES
4d198783b7c0        psteinha/emilonsteroids     0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
```

Abbildung B.3: 'docker ps' mit veröffentlichten Ports

Anhand des Formates ist erkennbar, dass eine Veröffentlichung gegen den Host stattgefunden hat. (Erkennbar am Pfeil)

### B.1.5 Test des Images

Werden die Ports nicht verändert, ist der Webshop unter "http://localhost" oder mit TLS über "https://localhost" erreichbar, ansonsten über die Ports, welche konfiguriert wurden. Als vorinstallierte Benutzer können "customer0" bis "customer11" mit Passwort "compass0" bis "compass11" verwendet, oder ein neuer Benutzer erstellt werden.

### B.1.6 Debugging

Der Webserver loggt alle seine Antworten oder Fehler im Terminal Fenster und im Dateisystem des Containers. Wird der Container als Hintergrundprozess gestartet, ist der Output des Terminal Fensters nicht ersichtlich. Falls etwas nicht funktioniert wie geplant, gibt es somit keinen Anhaltspunkt. Es kann sich aber jederzeit mit einem laufenden Container verbunden werden, um diesen Output und weitere Logfiles trotzdem sehen zu können.

**Zugriff auf laufenden Docker**
Um auf einen laufenden Container zuzugreifen, muss zuerst die ID des gewünschten Containers ermittelt werden:

```
1 docker ps
```

Ausgabe:

```
sh-3.2# docker ps
CONTAINER ID        IMAGE                       COMMAND
9aac776796ca        psteinha/emilonsteroids     "/utils/scripts/run_…"
```

Abbildung B.4: Container ID ermitteln

Um sich in den Container zu verbinden reicht nun folgender Befehl, wobei ContainerID mit der vorher ermittelten ersetzt werden muss:

```
1 docker exec -it ContainerID ash
```

**Zugriff auf Container während des Starts**
Ebenfalls ist es möglich, sich direkt beim Start in einen Container zu verbinden.

```
1 docker run -it --entrypoint /bin/ash -p 80:80 -p 443:443 hackinglab/emil
```

Dieser Befehl überschreibt den ENTRYPOINT des Dockerfiles und daher wird weder Webserver noch Datenbank gestartet. Falls benötigt kann dies mit folgendem Befehl manuell nachgeholt werden:

```
1 /utils/scripts/run_all
```

**Logfiles**

- Beim starten des Containers werden Logfiles über das Startverhalten aller Prozesse im Ordner "/var/log/all" in einer Datei pro Prozess gespeichert.
- Zusätzlich speichert die Datenbank eigene Logfiles über Transaktionen innerhalb des Containers unter "/var/log/mongodb/".
- Der Webserver loggt im laufenden Betrieb die Fehler nebst dem Terminal Fenster ebenfalls unter "/app/logs/log.txt".

### B.1.7 Hilfsscripts

Im Ordner "docker_config" des Source Codes befinden sich Dateien, welche entweder während des Docker Build Vorgangs oder während der Ausführung des Containers benötigt werden:

| **Datei** | **Ausführungs-zeitpunkt** | **Zweck** |
|---|---|---|
| configure.js | Docker Build | Erstellt Datenbank "webshop" und konfiguriert User und Passwort dafür. |
| mongodb_import | Docker Build | Importiert alle Standarddaten vom Ordner "data" in die DB |
| run_all | Docker Run | ENTRYPOINT des Containers. Startet alle Dienste welche in den Container unter "/utils/services/" kopiert wurden. |
| mongodb | Docker Run | Startscript für den Dienst: MongoDB |
| node | Docker Run | Startscript für den Dienst: Node.js |

Tabelle B.1: Ordnerinhalt von "docker_config"

Diese Files werden an verschiedene Ordner im finalen Docker Image kopiert. Wo genau, ist im Dockerfile selbst ersichtlich.

## B.2 Weiterentwicklung des Node.js Webserver

Dieser Abschnitt beschäftigt sich damit, auf was geachtet werden muss, wenn der Webshop weiterentwickelt wird. Dabei gilt: Bei einer Änderung im Code muss das Docker Image anschliessend neu erstellt werden.

### B.2.1 Voraussetzungen

Folgende Software ist nötig, um den Webshop weiterzuentwickeln:

- MongoDB
- Node.js 8.9 (Höhere Versionen wurden nicht getestet)
- Yarn
- Chrome Webbrowser für Tests

### B.2.2 Code Übersicht

Es folgt einer Übersicht, welche Teile des Codes wo zu finden sind.

**Frontend**
Der gesamte CSS, HTML und AngularJS Frontend Code ist im Order "public/app" zu finden. Darin enthalten sind die beiden Ordner "src" und "dist". Beide enthalten von der Funktionsweise her betrachtet denselben Code. Der Unterschied liegt im Format. Entwickelt wird der Webshop **ausschliesslich** im Ordner "src" mit Javascript im ECMAScript 6 Format. Aufgrund von Kompatibilitätsgründen wird der komplette Frontend Code ins ältere ECMAScript 5 Format gebracht und schlussendlich auch vom Ordner "dist" her an den Client ausgeliefert. Im finalen Docker Image findet sich also kein Hinweis mehr auf den Ordner "src". Mehr dazu im Abschnitt *B.2.3 Transpile ES6 zu ES5*.

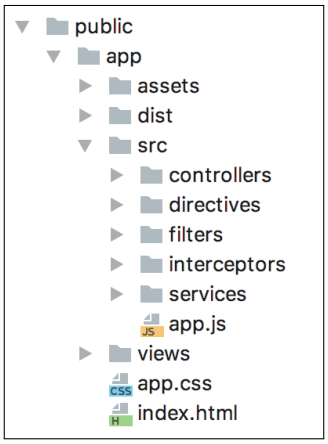


Abbildung B.5: Ordner Struktur im Frontend Bereich

### Backend

Der Backend Code befindet sich somit in allen anderen Ordner im Projekt:

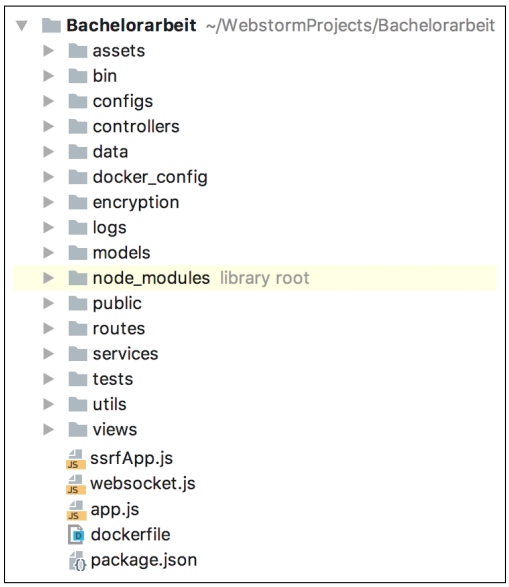


Abbildung B.6: Ordner Struktur im Backend Bereich

### Tests

Für das Testing relevante Dateien sind im Ordner "tests" abgelegt:

- **assets:** Demo Daten, welche während der Tests verwendet werden. (Datei Upload etc.)
- **e2e:** Beinhaltet alle Tests, welche die Webshop Funktionen testen.
- **vulnerabilities:** Der Unterordner beinhaltet Tests für die einzelnen Schwachstellen.

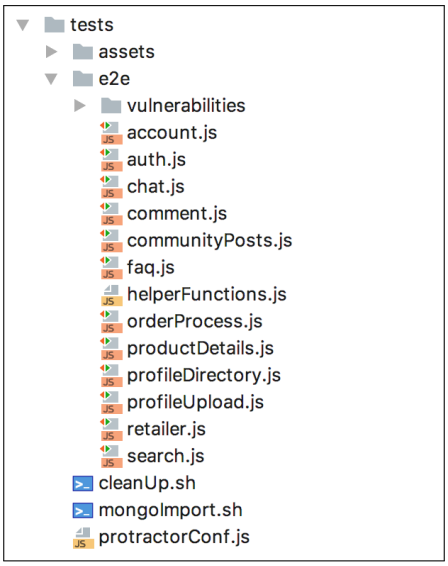


Abbildung B.7: Ordner Struktur der Tests

### B.2.3 Node.js Scripts

In Node.js wurde die Möglichkeit genutzt, mit diversen Start-Scripts zu arbeiten. Die meisten wurden auf macOS entwickelt und laufen daher nur auf Unix basierten Systemen. Gestartet werden können alle mit Yarn im Format: "yarn SCRIPTNAME".
Für Windows müssen die Scripts entweder so angepasst werden, dass sie universal für alle Betriebssysteme funktionieren, oder die entsprechenden Funktionen müssen von Hand gestartet werden. Ohne Anpassung funktionieren lediglich "prestart" und "start".

```
"scripts": {
  "prestart": "yarn install",
  "start": "node ./bin/www",
  "start:withMongo": "yarn run startmongo && sleep 5 && yarn start",
  "stop": "pkill -f mongod node",
  "startmongo": "yarn stop && sleep 5 && ./tests/mongoImport.sh",
  "pretest": "yarn run startmongo && webdriver-manager update && yarn run pretest:webdriver",
  "pretest:webdriver": "webdriver-manager start & sleep 10",
  "test": "yarn run build && yarn start 2>&1 > /dev/null 2>&1 & sleep 10 && protractor tests/protractorConf.js",
  "posttest": "./tests/cleanUp.sh && yarn stop",
  "clean-build": "rm -rf ./public/app/dist/*",
  "build": "yarn run clean-build && babel ./public/app/src -d ./public/app/dist"
},
```

Abbildung B.8: Vorhandene Node.js Scripts

Beschreibung der Scripts und ihrer Funktion:

| Script Name | Zweck |
|---|---|
| prestart | Installiert alle Dependencies, welche im "package.json" definiert sind |
| start | Startet automatisch "prestart" und danach den Node Server |
| start:withMongo | Startet MongoDB, importiert Testdaten und startet dann "start" |
| stop | Beendet alle laufenden MongoDB und Node Prozesse |
| pretest | Startet nur die MongoDB und importiert Testdaten |
| pretest:webdriver | Startet den Webdriver für die Tests |
| test | Startet automatisch "pretest" dann "pretest:webdriver" dann "clean-build" dann "build" dann "start" dann starten die Tests. Am Schluss startet noch automatisch "posttest" |
| posttest | Löscht alle temporären Testdaten und führt automatisch "stop" aus |
| clean-build | Löscht alle von ES6 zu ES5 übersetzten Javascript Dateien |
| build | Übersetzt alle Frontend Javascript Dateien von ES6 zu ES5 |

Tabelle B.2: Alle verfügbaren Node.js Scripts

Wie zu erkennen ist, sind viele der Scripts nur Helfer anderer und müssen meist nicht einzeln genutzt werden. Sie werden nur von einem anderen Script in einer Kette von Befehlen aufgerufen.

### Debugging der Tests

Da es sich bei den Tests um Frontend Tests handelt, werden diese in einem Chrome Browser ausgeführt. Standardmässig ist dieser unsichtbar. Reicht der Stacktrace der fehlerhaften Tests nicht aus, kann dieses Verhalten geändert und somit der Testablauf visuell mitverfolgt werden. Dazu muss im Source Code in der Datei "tests/protractorConf.js" unter "arg" die Option "–headless" entfernt werden.

```
capabilities: {
    'browserName': 'chrome',
    'loggingPrefs': {
        'driver': 'INFO',
        'server': 'INFO',
        'browser': 'INFO'
    },
    'acceptInsecureCerts': true,
    'chromeOptions': {
        args: ['--headless','--disable-gpu','--no-sandbox','--window-size=1500,1500'],
    },
},
```

Abbildung B.9: Auszug aus "protractorConf.js"

### Transpile ES6 zu ES5

Einige ältere Browser unterstützen kein ES6. Sowohl im Back- wie auch Frontend wurde mit verschiedenen ES6 Funktionen programmiert. Damit der Webshop auch für ältere Browser zugänglich ist, wurde mit dem Script "build" eine Möglichkeit geschaffen, im Frontend mit ES6 zu programmieren und den Source Code automatisch nach ES5 übersetzen zu lassen. Im Backend braucht es eine solche Konvertierung nicht.

Die Konvertierung wird automatisch beim ausführen der Tests mit:

```
1 yarn test
```

gestartet, was für die Integration in ein CI optimal ist. Alternativ kann der Prozess während der lokalen Entwicklungsphase auch ohne Tests ausgelöst werden:

```
1 yarn build
```

Da unter Windows "yarn build" nicht verwendet werden kann, ist der Übersetzungsprozess auch ohne Script ausführbar:

```
1 babel ./public/app/src -d ./public/app/dist
```

Eine erneute Übersetzung ist immer dann nötig, wenn Änderungen im Frontend Javascript Code gemacht wurden. Dieser Prozess ist automatisierbar, damit dies bei jeder Änderung im Code geschieht. Mehr unter: https://babeljs.io/docs/setup/#installation

**RCE / Serialization Bug**
Damit alles Tests erfolgreich sind und der Shop alle Funktionen bereitstellen kann, müssen die RCE und Serialization Schwachstellen per Umgebungsvariablen eingeschaltet werden, bevor die Tests gestartet werden.

Unix basierte Systeme:

```
export NODE_RCE_EVAL=on
export NODE_RCE_SERIALIZATION=on
```

Windows:

```
set NODE_RCE_EVAL=on
set NODE_RCE_SERIALIZATION=on
```

### B.2.4 Node.js Konfiguration

Die Konfigurationsdatei des Webservers ist im Source Code unter "configs/index.js" abgelegt. Darin sind alle Einstellungen definiert. Auf die wichtigsten wird nachfolgend eingegangen.

**Ports anpassen**
Die Ports, auf welche der Webserver hört, können hier angepasst werden.
"SSRFPort" ist für den versteckten Webserver für die "SSRF" Challenge gedacht und sollte weder geändert noch im Docker veröffentlicht werden, damit die Aufgabe weiterhin funktioniert.

```
server: {
    host: '0.0.0.0',
    port: '80',
    sslPort: '443',
    SSRFPort: '8765'
},
```

Abbildung B.10: Definition von IP und Ports des Webshops

### MongoDB Connection String

Hier gilt es zu beachten, dass falls "username", "password" oder "name" geändert wird, die gleichen Einstellungen auch in "docker_config/configure.js" geändert werden müssen. Diese Datei ist dafür verantwortlich, dass während des Docker Build-Prozesses die Datenbank korrekt konfiguriert wird.

```
mongo: {
    username: 'webshopEditor',
    password: '1234',
    host: 'localhost',
    port: 27017,
    name: 'webshop',
    connectionString() {
        return 'mongodb://' + this.username + ':' + this.password
            + '@' + this.host + ':' + this.port + '/' + this.name;
    }
},
```

Abbildung B.11: Connection String für die MongoDB

### Zertifikate austauschen

Der Webshop liefert die Daten nebst HTTP auch als HTTPS. Im Auslieferungszustand beinhaltet der Webshop dafür ein selbst signiertes Zertifikat inklusive Private Key.
Beide Dateien befinden sich im Ordner "encryption" und müssen im PEM Format vorliegen:

- **cert.pem:** Das Zertifikat
- **key.pem:** Der Private Key

### HTTP zu HTTPS Weiterleitung deaktivieren

Für gewisse Testzwecke kann es von Vorteil sein, die automatische Weiterleitung von HTTP zu HTTPS zu deaktivieren. Dafür wird im Basispfad des Webshops, in der Datei "app.js", die markierte Zeile auskommentiert.


Abbildung B.12: HTTP zu HTTPS Weiterleitung deaktivieren

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Literaturverzeichnis

[1] Hacking-Lab, 2018-05 (URL: https://www.hacking-lab.com)

[2] MEAN-Stack, 2018-05 (URL: http://mean.io)

[3] The OWASP Foundation *OWASP Top 10*, 2017-10 (URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)

[4] P. Steinhäusl *«Glockenemil on Steroids»*, Studienarbeit, HSR Hochschule für Technik Rapperswil, 2017

[5] Bootstrap , 2018-05 (URL: https://getbootstrap.com)

[6] C. Brunschwiler, Compass Security AG *Architektur*, 2018

[7] Docker *Docker*, 2018-04 (URL: https://www.docker.com)

[8] I. Fette, Google, Inc., A. Melnikov, Isode Ltd. *The WebSocket Protocol*, RFC 6455, 2011-12 (URL: https://tools.ietf.org/html/rfc6455)

[9] OWASP *Testing for NoSQL injection*, 2017-06 (URL: https://www.owasp.org/index.php/Testing_for_NoSQL_injection)

[10] OWASP *Top 10 2007-Insecure Direct Object Reference*, 2010-04 (URL: https://www.owasp.org/index.php/Top_10_2007-Insecure_Direct_Object_Reference)

[11] OWASP *Cross-site Scripting (XSS)*, 2018-06 (URL: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))

[12] OWASP *JSON Web Token (JWT) Cheat Sheet for Java*, 2018-04 (URL: https://www.owasp.org/index.php/JSON_Web_Token_(JWT)_Cheat_Sheet_for_Java)

[13] OWASP *DOM Based XSS*, 2015-06 (URL: https://www.owasp.org/index.php/DOM_Based_XSS)

[14] OWASP *Test Cross Origin Resource Sharing (OTG-CLIENT-007)*, 2013-12 (URL: https://www.owasp.org/index.php/Test_Cross_Origin_Resource_Sharing_(OTG-CLIENT-007))

[15] Michael Stepankin *Node.js code injection (RCE)*, 2016-08 (URL: http://artsploit.blogspot.com/2016/08/pprce2.html)

[16] *Exploiting Node.js deserialization bug for Remote Code Execution*, 2017-02 (URL: https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution)

[17] Gareth Heyes *XSS without HTML: Client-Side Template Injection with AngularJS*, 2016-01 (URL: https://portswigger.net/blog/xss-without-html-client-side-template-injection-with-angularjs)

[18] Docker *What is Server Side Request Forgery (SSRF)?*, 2017-03 (URL: https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability)

[19] Docker *Content-Types Matter More Than You Think*, 2018-01 (URL: https://textslashplain.com/2018/01/08/content-types-matter-more-than-you-think/)

[20] Node.js Community *Image Variants*, 2018-04 (URL: https://github.com/nodejs/docker-node/#image-variants)