Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API ..... *Application Programming Inferface*
B2B ..... *Business-to-Business*
CRM ..... *Customer-Relationship-Management*
CSS ..... *Cascading Style Sheets*
HTML ..... *HyperText Markup Language*
LWC ..... *Lightning Web Component*
LWR ..... *Lightning Web Runtime*
SEO ..... *Search Engine Optimization*
SLDS ..... *Salesforce Lightning Design System*
URL ..... *Uniform Resource Locator*

1 Einleitung

Die IT-Branche befindet sich im stetigen Wandel. Regelmäßig werden neue Technologien vorgestellt und viele, bereits vorhandene, werden den aktuellen Anforderungen entsprechend weiterentwickelt. Ein wichtiger Schritt eines modernen Unternehmens ist es, mit diesem Wandel mitzugehen und die eigenen Produkte kontinuierlich an diese neuen Technologien anzupassen. Dieser Prozess kann jedoch unter Umständen sehr zeit-, sowie kostenaufwändig sein.

Ein aktuelles Beispiel für so einen Technologiewandel ist die Weiterentwicklung der Salesforce Aura Runtime zur Lightning Web Runtime (LWR) für Salesforce Digital Experiences. Diese neue Runtime verspricht einige Vorteile gegenüber ihrem Vorgänger, weshalb ein Kunde der dotSource GmbH die Entscheidung traf, ihren bestehenden Business-to-Business (B2B) Shop auf die LWR zu migrieren.

Das Ziel dieser Arbeit besteht darin, ein Migrationskonzept zu erstellen, mit dessen Hilfe es möglich sein soll, eine bestehende Salesforce Storefront von der Aura Runtime auf die Lightning Web Runtime umzubauen. Dieser Punkt ist erfüllt, wenn die Komponenten der Storefront vollständig auf der LWR laufen und auch funktionell sind. Im Konzept soll nicht auf die Migration der Daten und Konfigurationen eingegangen werden, da dies sehr Shop spezifisch ist und den Ramen der Arbeit übersteigt. Schlussendlich wird das Migrationskonzept an verschiedenen Teilkomponenten des Kundenshops angewendet und dabei auf seine Praxistauglichkeit überprüft.

Hierfür wird zunächst ein Überblick über Salesforce, die Aura Runtime, sowie die Lightning Web Runtime und deren Vorteile gegeben. Dabei wird außerdem auf die Limitierungen der LWR im Vergleich zur Aura Runtime eingegangen. Nachdem nun die technischen Grundlagen erläutert sind, wird der aktuelle Stand des Kundenshops erfasst und das Konzept entwickelt, um die Storefront des Shops auf die neue Runtime zu migrieren. Anschließend wird dieses Konzept an einer Komponente des Shops angewendet. Letzten Endes folgt die Auswertung des Prozesses, die Zusammenfassung der Arbeit und das Fazit.

2 Grundlagen

2.1 Salesforce

Salesforce ist ein 1999 in San Francisco gegründetes Unternehmen, welches sich insbesondere auf das Gebiet Customer-Relationship-Management (CRM) spezialisiert und in diesem Bereich auch ihre eigene, sehr umfangreiche, Cloud-basierte Software entwickelt. Diese CRM-Software basiert auf dem „Software as a service“-Model und ist demnach abhängig vom Bedarf des Kundens skalierbar.

Mithilfe der Salesforce CRM-Software ist es möglich Onlineshops für den Business-to-Business, sowie den Business-to-Consumer Bereich zu entwickeln. Die Benutzeroberfläche dieser Shops werden mit Einsatz von Salesforce Aura-Komponenten oder Lightning-Web-Components (LWC) aufgebaut. Solche Komponenten sind Teilbausteine der Website, welche schlussendlich beliebig kombiniert werden können, um den fertigen Shop zu entwickeln.

Die Bereitstellung dieses Onlineshops zum tatsächlichen Nutzer kann über verschiedene Wege geschehen. Den genauen Ablauf dieses Prozesses bestimmt schlussendlich die Runtime, welche für die jeweilige Seite verwendet wird. Sie ist eine Software, welche, für den Endnutzer unsichtbar, auf den Servern des jeweiligen Shops ausgeführt wird und eingehende Anfragen verarbeitet und die fertige Seite an den Nutzer zurückliefert. Hierbei ist im Bereich Salesforce zwischen zwei Runtimes zu unterschieden. Die Aura Runtime und ihr Nachfolger, die Lightning Web Runtime.

2.2 Aura Runtime

Die Aura Runtime war über längere Zeit die einzig verfügbare Runtime für Salesforce Webshops. Sie bietet dem Entwickler die Möglichkeit sowohl Aura-Komponenten als auch ihren Nachfolger, die LWCs in seinem Shop einzubinden. Diese Komponenten werden im sogenannten Experience Builder dann per Drag-and-Drop zu einer kompletten Seite zusammengebaut (siehe Abbildung 1). Dem Entwickler stehen neben den eigenen Komponenten außerdem verschiedene vorgefertigte Standardkomponenten zur Auswahl. Hierbei ist aber wichtig anzumerken, dass es im Experience Builder nicht möglich ist, Komponenten als „Unterkomponente“ in eine andere Komponente hinzufügen und stattdessen nur vorgefertigte Layouts zur Verfügung stehen.

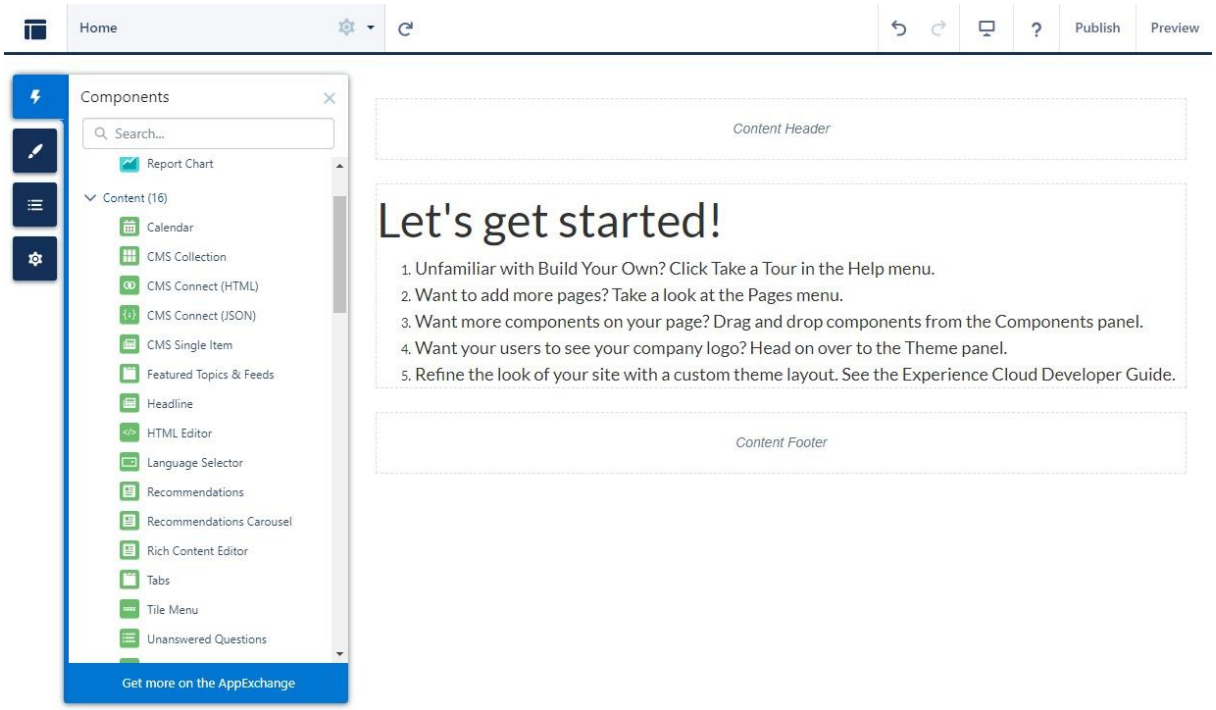

Abbildung 1: Der Experience Builder einer Beispiel Salesforce Seite

Über einen Workflow werden Änderungen veröffentlicht. Wenn jetzt ein Nutzer die Website betreten möchte, lädt die Aura Runtime die benötigten Komponenten und baut die Seite anhand der Informationen, an welchen Positionen sich welche Komponente befindet, zusammen und übergibt sie fertiggestellt dem Browser des Benutzers. Dieser Prozess wird mit jedem Seitenaufruf wiederholt und ist demnach verhältnismäßig langsam. Der Vorteil hierbei ist jedoch, dass Änderungen an Teilkomponenten sofort sichtbar werden.[1]

2.3 Lightning Web Runtime

Die Lightning Web Runtime ist der 2021 angekündigte Nachfolger der Aura Runtime und soll insbesondere das Leistungsproblem der Aura Runtime beheben. Salesforce spricht hierbei von Ladezeiten einer gesamten Seite niedriger als einer Sekunde.[2]

Diese Geschwindigkeitsverbesserungen erreicht die LWR insbesondere durch den Wegfall des Neuaufbaus bei jedem Besuch einer Seite. Stattdessen wird die Seite nach dem Veröffentlichen aus dem Experience Builder einmal erstellt und dann statisch auf dem jeweiligen Server bereitgelegt. Solch eine Herangehensweise bringt jedoch den Nachteil mit

[1] [Sal23b]
[2] [Sal23a]

sich, dass die Seite nach jeder Änderung an einer Teilkomponente, über den Experience Builder neu veröffentlicht werden muss, bevor die Änderungen für die Nutzer sichtbar werden.

Eine weitere Neuerung ist das Hinzufügen von Komponenten als Inhalt in andere Komponenten. Anders als bei der Aura Runtime ist es nun möglich eigenen Komponenten weitere „Unterkomponenten“ als Inhalt über den Experience Builder hinzuzufügen. Somit kann nun auch ein Administrator ohne Programmierkenntnisse diese Komponenten mit geringem Aufwand anpassen.

Darüber hinaus sind die Undo- und Redo-Buttons im Experience Builder aktuell nicht verfügbar. Änderungen werden demnach sofort gespeichert und können nicht mehr rückgängig gemacht werden. Abgesehen davon bleibt die Benutzeroberfläche, sowie die Funktionalität des Experience Builders jedoch weitgehend unverändert.

Des Weiteren bringt die Lightning Web Runtime neue Search Engine Optimization (SEO) Features wie zum Beispiel URL-Kurzbeschreibungen (Uniform Resource Locator) mit sich. Dabei werden URLs, welche Datensatz-IDs beinhalten, durch menschlich, und damit auch maschinell, besser lesbare Begriffe ersetzt. Diese Funktion ist jedoch momentan noch nicht öffentlich verfügbar und muss individuell über den Salesforce-Support freigeschaltet werden.

Trotz der vielen Vorteile bringt die LWR auch einige Limitierungen mit sich. So wurde beispielweise die Unterstützung für Aura Komponenten aufgehoben und die moderneren LWCs sind demnach die einzige Möglichkeit Komponenten für LWR-Seiten zu erstellen. Vorhandene Aura-Komponenten müssen demnach zur Migration zuerst als Lightning Web Komponente neu implementiert werden.

Des Weiteren sind viele Standardkomponenten noch nicht für die LWR verfügbar. Demnach ist eine Migration einer Seite, die stark von den Standardkomponenten abhängig ist, aktuell noch sehr mühsam.

Nennenswert ist außerdem, dass die sogenannte „Audiences“ Funktion zunächst nicht verfügbar ist. Diese Funktion erlaubt es, den Inhalt einer Seite für bestimmte Nutzergruppen verschieden aufzubauen. Eine Alternative hierfür soll zukünftig die „Expression-Based Visibility“ darstellen, welche aktuell jedoch ausschließlich für ausgewählte Salesforce Kunden freigeschaltet ist.

3 Migration eines Kundenprojekts zur Lightning Web Runtime

3.1 IST-Zustand

Die [redacted] GmbH ist ein in Baden-Württemberg ansässiger Hersteller von Lochblech und Streckmetallen. Das Unternehmen besitzt einen umfangreichen B2B-Onlineshop, welcher auf der Salesforce Plattform aufgebaut ist. Geschäftskunden haben in diesem Onlineshop unter anderem die Möglichkeit Standardprodukte zu kaufen, aber auch die vorgefertigten Produkte zu konfigurieren und vollständige, individuelle Metallprodukte zu designen.

Da der Webshop sich ausschließlich an Unternehmen richtet, bietet [redacted] zusätzlich Funktionen wie unternehmensspezifische Angebote oder eine schnelle, direkte Bestellung über den Header der Seite durch Artikelnummern an. Dies erleichtert den Bestellvorgang insbesondere für Stammkunden, die in der Regel eine genaue Vorstellung von den gewünschten Produkten haben und nicht lange im Shop suchen möchten.

Des Weiteren umfasst der Shop der [redacted] GmbH einen „Showroom“ Bereich. In diesem Bereich werden beispielsweise Interviews von Kunden und Mitarbeitern, sowie Anwendungsbeispiele und Tutorials für Kunden veröffentlicht.

Aktuell basiert der Salesforce Webshop auf der Aura Runtime und bringt dementsprechend einige Nachteile mit sich. Unter anderem die Ladezeiten, sowie die SEO könnten verbessert werden.

3.2 SOLL-Zustand

Um die aktuellen Mängel des Webshops zu beseitigen, möchte die [redacted] GmbH ihren vorhandenen Shop von der Aura Runtime auf die Lightning Web Runtime umstellen. Dabei erhofft sich das Unternehmen insbesondere im Bereich Geschwindigkeit und der Search Engine Optimization Verbesserungen.

Dieser Migrationsprozess besteht jedoch nicht nur aus dem Umlegen eines Schalters und bedarf stattdessen einem teilweisen Neuaufbau des bestehenden Webshops. Hierbei kann ein Großteil der bereits vorhandenen Komponenten jedoch wiederverwendet werden. Der Umbau betrifft außerdem ausschließlich die Storefront, also die Benutzeroberfläche des Webstores. Demzufolge bleiben der CRM-Bereich, sowie die vorhandenen Produkt- und Kundendaten vom Umzug größtenteils unberührt und können direkt im neuen Store eingebunden werden.

Damit so ein umfangreicher Prozess möglichst unproblematisch abläuft, bedarf es einer geordneten Informationssammlung. Das im nächsten Abschnitt folgende Konzept soll eine Übersicht über alle notwendigen Änderungen darstellen, um die Storefront eines bestehenden Aura Shops auf die Lightning Web Runtime zu migrieren.

3.3 Konzept

Bevor die Migration eines Stores durchgeführt wird, sollten einige Voraussetzungen geklärt werden. Aktuell ist es nicht für alle Shops empfehlenswert, auf die LWR umzustellen, da spezifische Funktionen aus der Aura Runtime momentan noch nicht vollständig in die LWR übernommen sind. Prinzipiell gilt aber, dass sich jede dieser Funktionalitäten eines Aura Runtime Shops, auf die LWR übertragen lässt, der Aufwand hierfür kann aber teilweise zu groß werden. Diese Funktionen umfassen beispielsweise die Aura Komponenten, einen Großteil der Standardkomponenten und Page-Variations. Wenn der zu migrierende Store stark abhängig von einer oder mehrerer dieser Funktionalitäten ist, sollten der Aufwand und der tatsächliche Nutzen eines schnelleren und SEO-freundlicheren Shops individuell abgewägt werden.

3.3.1 Migration vorhandener Aura-Komponenten

Der erste Schritt einer Migration sollte darin bestehen die noch vorhandenen Aura-Komponenten als Lightning Web Components neu zu implementieren. Bei moderneren Shops, die nach der Veröffentlichung von Salesforces „Lightning“ aufgebaut wurden und demnach zum Großteil LWCs verwenden sollten, kann dieser Schritt fast vollständig wegfallen.

Der Funktionsumfang der LWCs deckt den der Aura Komponenten fast vollständig ab und demzufolge kann eine Migration hierbei sehr unkompliziert und unaufwändig ablaufen. Bei kleineren Komponenten, welche beispielsweise nur eine Benutzeroberfläche anzeigen sollen, reicht es die einzelnen Aura Funktionen in die äquivalenten LWC-Funktionen umzuschreiben und die neuere Syntax anzuwenden. Insbesondere bei umfangreicheren Aura Komponenten empfiehlt Salesforce jedoch, die Komponente vollständig neu aufzubauen, da sich die Programmiermodelle der Aura Komponenten und LWCs fundamental unterscheiden.

Wenn eigene Theme- oder Page-Layouts in Verwendung sind, müssen diese ebenfalls als LWCs umgebaut werden, da in der Aura Runtime ausschließlich Aura Komponenten als Layout eingebunden werden können und in der LWR ausschließlich als LWCs. Diese Layouts werden mit Hilfe der neuen Slot-Funktionalität umgesetzt. Der Entwickler hat damit die

Möglichkeit in seiner Komponente Slots festzulegen, welche dann im Experience Builder als Bereich angezeigt werden, in dem andere Komponenten hinzugefügt werden können.[3]

Salesforce bietet eine ausführliche Dokumentation, wie eine Aura Komponente als LWC neu implementiert werden kann[4]. Diese zusammenfassende Quelle ermöglicht einen unproblematischen Wechsel auf LWCs im gesamten Store.

3.3.2 Ablösen der Standardkomponenten

Im nächsten Schritt werden nun die im Shop verwendeten, aber nicht mehr unterstützten Standardkomponenten nachgebaut. Auch hier pflegt Salesforce eine Liste aller aktuell verfügbaren Komponenten aus der Lightning Web Runtime[5]. Die Anzahl der Standardkomponenten soll zukünftig weiter ansteigen, weshalb es bei einer umfangreichen Verwendung dieser Komponenten empfehlenswert sein kann, eine Migration auf die LWR vorerst zu verzögern.

3.3.3 Alternativen für Audiences & Page-Variations

Eine weitere Funktion, welche mit der Lightning Web Runtime vorerst wegfällt, sind die Audiences und damit auch die Page-Variations. Über Page-Variations ist es möglich den Inhalt einer Seite, abhängig von bestimmten Kriterien, z.B. der aktuellen Domain oder bestimmter Felder am Nutzer, beliebig zu variieren.

Zukünftig soll die „Expression-Based Visibility“ eine Alternative für diese Funktionalität darstellen, aktuell müssen jedoch eigene Komponenten entwickelt werden, um die Funktionen der Page-Variations in Teilen nachzubauen. Solch eine Komponente lässt sich ebenfalls mit Hilfe der neuen Slot-Funktionalität umsetzen. Dabei kann, abhängig von der nachzubauenden Funktion, der Umfang dieser Komponente stark variieren.

Eine Komponente, die als Ersatz für eine Page-Variation dienen soll, besitzt in der Regel zwei Slots. Der eine soll dabei den „True“-Fall darstellen und der andere den „False“-Fall. Nun führt die Komponente einen Vergleich durch. Abhängig vom Ergebnis dieses Vergleiches, wird ein Slot ausgeblendet und der andere wird angezeigt. So ein Vergleich kann beispielsweise sein: „Das Land des aktuellen Nutzers ist Deutschland“. Im Experience Builder werden dann alle Komponenten, die nur für einen der beiden Fälle angezeigt werden sollen, zum jeweiligen Slot

[3] [Sal23i]
[4] [Sal23c]
[5] [Sal23d]

hinzugefügt. Das können sowohl die gesamte Seite, als auch nur vereinzelte Komponenten sein.

3.3.4 Expressions durch eigene Implementierungen ersetzen

Expressions sind Platzhalter, die Komponenten aus dem Experience Builder übergeben werden können. Diese Platzhalter werden beim Aufrufen einer Komponente durch die dazugehörigen Daten ersetzt.

Die verfügbaren Expressions für Komponenten, welche auf der Lightning Web Runtime laufen, wurden ebenfalls geändert[6]. Die wohl wichtigste Änderung ist hierbei der Wegfall der „CurrentUser“ Expression. Diese Expression ermöglicht einen einfachen Zugriff auf verschiedene Felder des aktuellen Nutzers der Seite. Eines der wichtigsten dieser Felder ist das „effectiveAccountId“ Feld, welches die ID des zu dem Nutzer gehörenden Account Objekts liefert.

Wenn in einer Komponente, welche auf der LWR läuft, auf die Felder des CurrentUser zugegriffen wird, muss demnach eine eigene Lösung entwickelt werden. Dazu gibt es die eine Möglichkeit, die benötigten Felder über den getRecord Wire-Service abzurufen, und eine zweite Möglichkeit, die Felder über eine Apex-Methode mit Hilfe einer SOQL-Query zu erhalten.

```
1. @AuraEnabled(cacheable=true)
2. public static Id getEffectiveAccountId(Id userId) {
3.     User effectiveUser = [SELECT Contact.AccountId FROM User WHERE Id = :userId LIMIT 1];
4.
5.     if (effectiveUser == null) {
6.         return null;
7.     }
8.
9.     if (effectiveUser.Contact == null) {
10.        return null;
11.    }
12.
13.    return effectiveUser.Contact.AccountId;
14. }
```

Abbildung 2: Apex-Methode zum Erhalt der effektiven AccountId

[6] [Sal23e]

3.3.5 Migration auf die Custom Component APIs

Eine weitere Verbesserung der Lightning Web Runtime betrifft die ConnectApi. Die ConnectApi bietet eine Schnittstelle, mit der aus Lightning Web Components auf Daten des Shops zugegriffen wird, oder Aktionen des Benutzers an den Shop weitergeleitet werden. Solche Daten und Aktionen umfassen beispielsweise Produktdaten, oder das hinzufügen eines Produktes zum Warenkorb.

In der Aura Runtime muss die ConnectApi zwangsweise aus einer Apex-Klasse aufgerufen werden, da es keine einfache Anbindung für die LWCs gibt. Dementsprechend wird zuzüglich zur LWC mindestens eine Apex-Klasse benötigt, welche intern die ConnectApi Aufrufe tätigt, und die dazu noch über Unit-Tests getestet werden muss. Darüber hinaus bringt dieser Prozess eine geringere Geschwindigkeit mit sich, da insgesamt mindestens zwei API-Aufrufe (Application Programming Inferface) getätigt werden müssen.

Mit der Lightning Web Runtime wurden neben der ConnectApi nun weitere APIs hinzugefügt, mit denen es möglich ist, direkt aus LWCs Daten aus dem Shop zu erhalten oder Nutzeraktionen an den Shop zu senden[7]. Diese APIs werden „Custom Component APIs“ genannt und werden mithilfe des Wire-Service der LWCs aufgerufen. Die Aufrufe dieser APIs aus LWCs verlaufen dabei schneller als ein Aufruf der ConnectApi, da hier nur noch ein API-Aufruf getätigt werden muss. Zuzüglich fällt der Aufwand einer Apex-Klasse ebenfalls weg.

Ein Umstieg auf die neuen APIs ist unkompliziert und kaum zeitintensiv und demnach bei einer Migration auf die LWR empfehlenswert. Das Datenmodell der neuen APIs ist exakt gleich aufgebaut wie das der ConnectApi, wodurch neben dem Aufruf der API, keine Änderungen an der Logik der Komponente nötig sind.

3.3.6 Neuimplementierung des Checkout Prozesses

Flows sind eine Funktion von Salesforce, komplexe Prozesse ohne Verwendung von Quellcode zu automatisieren[8]. Aktuell arbeiten Salesforce Shops, die auf der Aura Runtime aufgebaut sind, den Checkoutprozess mithilfe eines Checkout-Flows ab. Dieser Flow bestimmt, wann bei diesem Vorgang, welche Aktionen durchgeführt werden müssen und was dabei dem Nutzer angezeigt wird, damit die Produkte im Warenkorb zu einer fertigen

[7] [Sal23f]
[8] [Sal23j]

Bestellung zusammengestellt werden. Dabei werden vom Käufer relevante Informationen, wie z.B. die Lieferadresse und Bezahlmethode, abgefragt.
Mit der Lightning Web Runtime ist es nicht mehr möglich Checkout-Flows in den Shop einzubinden. Stattdessen wird der Checkoutprozess nun über eine normale Seite mit Hilfe von Lightning Web Components gesteuert.[9]

Um den Checkoutprozess auf den neuen Shop zu übertragen, gibt es zwei Möglichkeiten. Die erste, von Salesforce empfohlene Variante, ist den Checkout als Konstrukt aus LWCs zu rekonstruieren. Das bringt unter anderem den Vorteil mit sich, dass der neue Checkoutprozess deutlich schneller als ein Checkout-Flow abläuft. Wenn der vorhandene Checkout-Flow allerdings sehr umfangreich und dementsprechend aufwändig zu migrieren ist, wäre eine andere Variante, den aktuellen Checkout-Flow zu einem Screen-Flow umzustrukturieren. Da der Checkout-Flow lediglich ein Screen-Flow mit zusätzlichen Standardkomponenten ist, ist dieser Vorgang deutlich weniger aufwändig als eine vollständige Neuimplementierung. Hierbei müssen lediglich die verwendeten, aber nicht mehr verfügbaren Komponenten nachgebaut werden. Zusätzlich muss die ID des aktuellen Warenkorbs manuell an den Flow übergeben werden, da dies bei Aura-Shops die Checkout-Flow Komponente übernimmt. Das kann beispielsweise durch URL-Parameter erreicht werden. Diese Variante sollte allerdings nur als eine Vorrübergehende Lösung betrachtet werden.

3.3.7 Migration des Stylings

Im letzten Schritt sollte nun das Styling im Shop migriert und anschließend überprüft werden. Die eigentliche Migration ist dabei abhängig von der verwendeten Styling Möglichkeit im Shop. Besitzen Komponenten beispielsweise jeweils ihre eigenen Stylesheets, sind keine weiteren Schritte notwendig, um das Styling zu übertragen. Sollte allerdings eine Haupt-CSS-Datei verwendet werden, die über den HTML-Head (HyperText Markup Language) eingebunden wird, muss diese im neuen Shop ebenso im Head eingebunden werden. Dabei besteht der einzige Unterschied darin, dass es mit der LWR nun möglich ist, die vordefinierten Variablen „`basePath`“ und „`versionKey`“ im Head-Markup zu verwenden. Die Variable „`basePath`“ liefert das Basisverzeichnis des aktuellen Shops. Wenn die URL beispielsweise „`https://www.[domain}.de/shop/s`“ lautet, wird die Variable durch den Text „`shop/s`“ ersetzt. Die zweite Variable „`versionKey`“ beinhaltet den aktuellen Versionsschlüssel des Shops und wird mit jeder Neuveröffentlichung der Website durch einen Neuen ersetzt. Durch diese

[9] [Sal23g]

Änderung müssen beide Werte nun nicht mehr manuell für jeden Shop gepflegt werden und erleichtern dementsprechend die Wartung des Shops.[10]
Insbesondere wenn im Shop vermehrt das Salesforce Lightning Design System verwendet wird und die eigenen CSS-Regeln (Cascading Style Sheets) die Standardstile teilweise überschreiben, können durch undokumentierten Änderungen in der LWR vereinzelt unerwünschte Fehler auftreten. Hierbei sind die eigenen CSS-Stile der jeweiligen Elemente an die neuen Standardstile anzupassen.

3.4 Durchführung

In diesem Abschnitt soll nun das Migrationskonzept auf einige Teilkomponenten des Kunden B2B Shops angewendet werden. Ziel dabei ist es, die Produktdetailseite, inklusive Header und Footer, vollständig auf einen LWR-Shop zu übertragen und anschließend Geschwindigkeitstests durchzuführen. Zusätzlich wird der vorhandene Checkout-Prozess auf diesen LWR-Shop migriert, sowie eine Komponente entwickelt, welche die Aufgabe einer Page-Variation übernimmt.

3.4.1 Migration des Theme-Layouts

Der erste Schritt des Konzepts besteht darin, vorhandene Aura-Komponenten als Lightning Web Components neu zu implementieren. Im Fall des Kunden sind optimalerweise bereits alle Komponenten als LWCs aufgebaut, was diesen Schritt sehr vereinfacht. Die einzige Ausnahme bildet das Theme-Layout, welches auf der Aura Runtime zwangsweise als Aura Komponente implementiert wird.

Das Theme-Layout beinhaltet bei dem Kunden neben der Grundstruktur, zusätzlich den Header, sowie den Footer. Auch hierbei wurden bereits alle Teilkomponenten des Headers und des Footers als Lightning Web Components implementiert. Da das Layout keine Aura-Spezifischen Funktionen verwendet und lediglich die Struktur der Seite, sowie einige LWCs beinhaltet, benötigt die Komponente fast ausschließlich Syntaxänderungen um als Lightning Web Component zu funktionieren. Aus Komponentenaufrufen wie „`<c:cmppHeaderLogo />`“ wird
„`<c-cmpp-header-logo></c-cmpp-header-logo>`“ und aus Variablenübergaben wie „`<div class="{!className}">`“ wird „`<div class={className}>`“. Anschließend müssen die

[10] [Sal23h]

Eigenschaften, welche im Experience Builder angezeigt werden sollen, in die Meta-, sowie die JavaScript-Datei der LWC übernommen werden.

Damit schlussendlich Komponenten in die jeweiligen Bereiche des Layouts eingefügt werden können, wird die Slot-Funktion der LWR verwendet. Im HTML-Code wird an der Stelle, wo eine Komponente hinzugefügt werden soll, ein Slot-Element mit einem bestimmten Namen eingefügt. Dieser Slotname wird anschließend in der JavaScript-Datei unter Verwendung der JSDoc-Notation referenziert, um den Experience-Builder zu informieren, welche Slots als Drag-and-Drop Region verfügbar sind. Der Hauptslot, zu dem der eigentliche Inhalt hinzugefügt werden soll, muss als Namenslos definiert sein.

```
1. <div class="cmpp-searchsuggest">
2.     <slot name="search"></slot>
3. </div>
```

Abbildung 3: HTML-Code des „search“ Slots

```
1. /**
2. * @slot search
3. */
```

Abbildung 4: JSDoc Notation im JavaScript-Code

Die Struktur des neuen Theme-Layouts ist nun vollständig migriert und wird anschließend über den Experience-Builder als Layout für die Produktdetailseite ausgewählt. Schlussendlich werden zusätzlich die Such-, Navigations- und Direktbestellungskomponenten zu den jeweiligen Slots hinzugefügt. In einem folgenden Abschnitt werden die letzten notwendigen Änderungen an der Logik des Theme-Layouts angewendet, um die Migration vollständig abzuschließen.

3.4.2 Entwickeln einer Page-Variations Alternative

Der nächste Schritt besteht nun darin eine Alternative für Audiences & Page-Variations zu entwickeln. Der Kunde verwendet diese Funktion aktuell auf verschiedenen Seiten. Ein Beispiel hierfür ist die AGB-Seite, auf welcher der Inhalt abhängig vom aktuellen Land des Stores ist. Um dies zu erreichen, gibt es für die einzelnen Länder jeweils eine Audience, welche überprüft, ob die aktuell besuchte URL der Domain des jeweiligen Stores entspricht. So gibt es beispielsweise eine „Kunden FR“ Audience, die als Bedingung besitzt, dass die Domain gleichdem String „www.{domain}.fr“ ist.

Im Folgenden soll nun eine Komponente entwickelt werden, die ebenfalls so eine Funktionalität bietet, dabei jedoch nicht auf Audiences und Page-Variations angewiesen ist

und somit auch auf der Lightning Web Runtime funktioniert. Der Aufbau ist dabei gleich wie im Konzept erwähnt. Die Komponente soll über den Experience Builder eine Domain erhalten und anschließend beim Aufrufen überprüfen, ob die aktuelle URL der eingegebenen Domain entspricht und je nach Ergebnis einen der beiden Slots anzeigt. Die Komponente besteht, wie alle LWCs, aus einem HTML-Teil, sowie einen JavaScript-Teil.

Der HTML-Teil der Komponente ist sehr unkompliziert aufgebaut. Er besteht lediglich aus zwei „`template`“-Elementen die jeweils ein „`slot`“-Element beinhalten. Die Templates besitzen außerdem beide ein „`if:true`“-Attribut, welches an eine Variable gebunden ist. Über diese Variable kann der jeweilige Slot ein- und ausgeblendet werden. In der JavaScript-Datei müssen zusätzlich beide Slots jeweils über die JSDoc-Notation referenziert werden, um sie im Experience-Builder anzuzeigen.

```
1. <template>
2.     <template if:true={showTrueSlot}>
3.         <slot name="true"></slot>
4.     </template>
5.     <template if:true={showFalseSlot}>
6.         <slot name="false"></slot>
7.     </template>
8. </template>
```

Abbildung 5: HTML-Code der Domain-Page-Variation Komponente

Anschließend wird die Logik der Komponente implementiert. Diese ist ebenfalls sehr kompakt, da lediglich nach Einbinden der Komponente überprüft werden muss, ob die aktuelle Domain dem zuvor eingegebenen String entspricht. Wenn das der Fall ist, wird die „`showTrueSlot`“ Variable auf „`true`“ gesetzt, im anderen Fall erhält „`showFalseSlot`“ den Wert „`true`“. Des Weiteren muss unterschieden werden, ob die Komponente im Experience Builder aufgerufen wird, oder auf der normalen Seite. Im Experience Builder sollen in allen Fällen beide Slots angezeigt werden, damit der Entwickler die Möglichkeit hat, den Inhalt der Slots zu modifizieren.

Um direkt nach dem Laden einer Komponente Code auszuführen, gibt es für Lightning Web Components die „`connectedCallback`“ Methode. In dieser Methode findet der eigentliche Vergleich statt. Die aktuelle Domain wird über die „`host`“ Eigenschaft am „`window.location`“ JavaScript-Objekt erhalten. Dieses Objekt wird von allen aktuell üblichen Browsern unterstützt und kann somit ohne Bedenken verwendet werden. Das Überprüfen, ob die Komponente gerade im Experience Builder ausgeführt wird, übernimmt die eigene Methode „`isInSitePreview`“. Diese gibt „`true`” zurück, sobald in der URL ein Keyword gefunden wird,

welches auf den Experience Builder deutet. Wird keines dieser Keywords gefunden, gibt die Funktion „`false`“ zurück.

```
import { api, LightningElement } from 'lwc';

/**
 * @slot true
 * @slot false
 */
export default class CmppDomainAudienceTest extends LightningElement {
    @api
    domainValue;

    showTrueSlot;
    showFalseSlot;

    connectedCallback() {
        if (this.isInSitePreview()) {
            this.showTrueSlot = true;
            this.showFalseSlot = true;

            return;
        }

        if (window.location.host == this.domainValue) {
            this.showTrueSlot = true;
            this.showFalseSlot = false;
        }
        else {
            this.showTrueSlot = false;
            this.showFalseSlot = true;
        }
    }



    isInSitePreview() {
        let url = document.URL;

        return (url.indexOf('sitepreview') > 0
            || url.indexOf('livepreview') > 0
            || url.indexOf('live-preview') > 0
            || url.indexOf('live.') > 0
            || url.indexOf('.builder.') > 0);
    }
}
```

Abbildung 6: JavaScript-Code der Page-Variation Komponente

Somit ist die Komponente vollständig und kann nun an Stelle einer Page-Variation eingebunden werden, um von der Domain abhängigen Inhalt anzuzeigen. In die Slots können anschließend beliebig viele Komponenten hinzugefügt werden.

3.4.3 Neuimplementierung der Aura-Expressions

Einige Komponenten der Produktdetailseite, wie auch das Theme-Layout selbst, sind auf die effektive AccountId des aktuellen Nutzers angewiesen. Diese wird bisher mithilfe der „`{!CurrentUser.effectiveAccountId}`“ Expression übergeben. Da diese Expression mit der Lightning Web Runtime nicht mehr verfügbar ist, wird die Apex-Methode aus Abbildung 2 des Konzepts als Alternative verwendet. Diese Methode wird anschließend im JavaScript-Code der jeweiligen Komponente über den Wire-Service aufgerufen und speichert das Ergebnis in der Variable „`effectiveAccountId`“. Der Wire Aufruf, sowie die nötigen Imports müssen zu jede der Komponenten hinzugefügt werden.

```
1. import { LightningElement, api, wire } from 'lwc';
2. import getEffectiveAccountId from '@salesforce/apex/CmppLWRHelpers.getEffectiveAccountId';
3. import userId from '@salesforce/user/Id';
4.
5. /**
6. * @slot navigation
7. * @slot search
8. * @slot quickorder
9. * @slot content
10. */
11. export default class CmppStandardThemeLayout extends LightningElement {
12.     @api
13.     ouNumber;
14.     effectiveAccountId;
15.
16.     @wire(getEffectiveAccountId, { userId: userId })
17.     wiredEffectiveAccountId(res) {
18.         if (res.data) {
19.             this.effectiveAccountId = res.data;
20.         }
21.     }
22. }
```

Abbildung 7: JavaScript-Code des Layouts mit Wire-Aufruf der Apex Methode

Dieser und der erste Schritt reichen bereits aus, um alle Komponenten der Produktdetailseite funktionsfähig auf der Lightning Web Runtime laufen zu lassen. An der Seite selbst werden in den folgenden Abschnitten lediglich Verbesserungen, sowie Styling Anpassungen durchgeführt.

3.4.4 Ersetzen der ConnectApi durch die Custom Component APIs

Die Produktdetailseite verwendet im aktuellen Shop die ConnectApi, um auf die Daten der einzelnen Produkte zuzugreifen. Insbesondere in diesem Fall ist es von Vorteil, auf die schnellere Custom Component API umzusteigen, da die Seite SEO-relevant ist und der Gesamtshop dementsprechend stark von einer schnelleren Ladezeit auf der Produktdetailseite profitiert.

Der Aufwand einer Migration ist hier, wie bereits im Konzept erwähnt, sehr gering. Es wird in der Hauptkomponente lediglich an einer Stelle die ConnectApi aufgerufen, um Produktdaten zu erhalten. Dieser Vorgang wird bereits mit Hilfe des Wire-Services durchgeführt. Der Wire-Service ruft eine Apex Funktion „`getProduct`“ auf, die anschließend die ConnectApi aufruft und somit die Produktdaten zurückliefert. Dieses Ergebnis wird dann in der Variablen „`product`“ gespeichert. Um diesen Prozess durch die Custom Component APIs zu ersetzen, muss lediglich der „`ProductAdapter`“ importiert werden, und anschließend der Wire-Aufruf auf diesen Adapter zeigen. Wie im Konzept beschrieben, bleibt der Datentyp der gleiche und somit sind keine weiteren Änderungen an der Komponente notwendig.

```
1. import { ProductAdapter } from 'commerce/productApi'
2.
3. @wire(ProductAdapter, { productId: '$recordId' })
4. product;
```

Abbildung 8: Der Wire-Aufruf des ProductAdapters

3.4.5 Migration des Checkout-Flows

Anschließend soll der Checkout-Flow auf die Lightning Web Runtime migriert werden. Da der Checkout-Flow des Shops sehr umfangreich ist, wird aus Aufwandsgründen die zweite Variante der Checkout-Migration im Konzept gewählt.

Dafür wird zuerst der vorhandene Checkout-Flow als Screen-Flow umgesetzt. Der Flow wird zunächst dupliziert und anschließend wird der Typ geändert. Da der Typ eines vorhandenen Flows über den Flow-Builder nicht geändert werden kann, muss hierfür die Meta-Datei des Flows direkt bearbeitet werden. Des Weiteren müssen die Komponenten durch Alternativen ersetzt werden, die nicht für einen Screen-Flow verfügbar sind. In diesem Fall sind das zum einen die „Redirect“-Komponente, die den Besucher auf eine andere Seite im Store weiterleitet, und die „Commerce Screen Wait“-Komponente, die eine Ladeanimation anzeigt, solange der Checkout-Prozess intern verarbeitet wird.

Die Redirect-Komponente wird lediglich als letzten Baustein eingebunden, um auf die Bestellbestätigungsseite weiterzuleiten. Diese Komponente übernimmt für den eigentlichen Flow keine funktionsrelevante Aufgabe und kann daher für den Rahmen dieser Arbeit durch ein einfaches Textelement mit dem Inhalt „Bestellung erfolgreich“ ersetzt werden.

Die „Commerce Screen Wait“-Komponente ist für die Funktion des Checkouts essenziell und kann deshalb nicht durch eine temporäre Alternative ausgetauscht werden. In diesem Fall muss die Funktionalität der Standardkomponente durch eine eigene Lightning Web

Component ersetzt werden. Salesforce bietet das Objekt „`CheckoutSession`“, welches verschiedene Daten eines Checkout Prozesses speichert. Dieses Objekt besitzt unter anderem das Feld „`isProcessing`“, welches auf „`true`“ gesetzt wird, wenn der Checkout intern verarbeitet wird, und auf „`false`“, sobald die Verarbeitung abgeschlossen ist. Mit dieser Information kann eine Apex-Methode „`getCheckoutSessionProcessing`“ entwickelt werden, welche die ID zu einer Checkout Session als Eingabeparameter erhält, und als Ergebnis den Wert des Feldes „`isProcessing`“ zurückliefert. Die Komponente muss anschließend periodisch diese Methode aufrufen und überprüfen, ob das Ergebnis der Methode „`false`“ ist. Sobald dies eintrifft, sendet die Komponente an den Flow ein Event, um zu signalisieren, dass der Checkout fortfahren kann.

```
@AuraEnabled(cacheable=true)
public static Boolean getCheckoutSessionProcessing(Id checkoutSessionId) {
    CartCheckoutSession session = [SELECT IsProcessing FROM CartCheckoutSession
        WHERE Id = :checkoutSessionId LIMIT 1];

    if (session == null) {
        return true;
    }

    return session.IsProcessing;
}
```

Abbildung 9: Apex-Methode zum Erhalt des Verarbeitungsstatus eines Checkout

```
import { api, LightningElement } from 'lwc';
import getCheckoutSessionProcessing from
    '@salesforce/apex/CmppLWRHelpers.getCheckoutSessionProcessing';
import { FlowNavigationNextEvent } from 'lightning/flowSupport';

export default class CmppCommerceScreenWaitLWR extends LightningElement {
    @api
    checkoutSessionId;

    connectedCallback() {
        setTimeout(this.checkSession.bind(this), 1000);
    }

    checkSession() {
        getCheckoutSessionProcessing({ checkoutSessionId: this.checkoutSessionId })
            .then(res => {
                if (res === false) {
                    this.dispatchEvent(new FlowNavigationNextEvent());
                    return;
                }

                setTimeout(this.checkSession.bind(this), 1000);
            })
            .catch(err => {
                console.log("cmppCommerceScreenWait error: ", err);
            });
    }
}
```

Abbildung 10: JavaScript-Code einer „Commerce Screen Wait“ Alternative

Der HTML-Code dieser Komponente beinhaltet die Ladeanimation, die während des Prozesses angezeigt werden soll. Für den Rahmen dieser Arbeit ist hierbei ein einziges Textabsatz-Element mit dem Inhalt „Laden…“ ausreichend. Die Komponente ist somit vollständig und kann in die jeweiligen Bereiche des ursprünglichen „Commerce Screen Wait“ eingesetzt werden.

Nachdem der eigentliche Flow nun vollständig migriert ist, kann er mit Hilfe der Flow-Standardkomponente über den Experience Builder zu der Checkoutseite hinzugefügt werden. Der Flow benötigt anschließend die ID des Warenkorbs als Eingabe. Diese ID kann mithilfe der neuen Expressions über „`{!Route.cart}`“ als URL-Parameter an die Flow-Komponente übergeben werden. Dabei ist „cart“ der Name des URL-Parameters und der dazugehörige Wert ist die jeweilige Einkaufswagen ID. Nun gibt es verschiedene Ansätze, wie die ID als Parameter in die URL gelangt. Es wäre beispielsweise möglich, überall dort, wo auf die Checkout-Seite weitergeleitet wird, gleichzeitig den Parameter hinzuzufügen, oder eine Komponente zu entwickeln, die für den aktuellen Nutzer die korrekte Warenkorb ID findet und diese auf der aktuellen Seite nachträglich in die URL einträgt. Da hierbei lediglich der eigentliche Checkout migriert werden soll, ist es für Testzwecke ausreichend, die ID eines Warenkorbs manuell in die URL zu schreiben.

3.4.6 Einbinden und Anpassen des Stylings

Die in den letzten Abschnitten migrierten Komponenten sind jetzt vollständig auf der LWR funktionell, sehen jedoch noch nicht benutzerfreundlich aus und unterscheiden sich optisch stark vom aktuellen Shop. Um dieses Problem zu lösen, wird nun der letzte Schritt des Konzepts angewendet und das Styling des aktuellen Shops auf den Neuen übertragen.

Der Shop verwendet Salesforce Lightning Design System (SLDS), sowie ein eigenes Stylesheet, welche alle benötigten CSS-Klassen und deren Attribute beinhaltet, um die einzelnen Komponenten zu stylen. Dieses Stylesheet wird über den HTML-Head der Seite eingebunden und ist somit für alle Unterkomponenten der Seite verfügbar. Die Einbindung erfolgt, wie auch im aktuellen Aura Shop, über den Experience Builder. Dabei werden das aktuell fest eingetragene Basisverzeichnis, sowie der Versionsschlüssel durch die im Konzept genannten, neuen Variablen ersetzt.

```
<link
  rel="stylesheet"
  href="{ basePath }/assets/styles/salesforce-lightning-design-system.min.css?{ versionKey }"
/>

<link
  rel="stylesheet"
  href="{ basePath }/sfsites/c/resource/theme/dist/css/main.min.css?{ versionKey }"
/>
```

Abbildung 11: Einbinden der CSS-Dateien im Head-Markup

Nachdem nun die nötigen CSS-Regeln eingebunden sind, ist nach einer Überprüfung aller Komponenten festzustellen, dass zwei weitere Änderungen vorgenommen werden müssen, damit die Komponenten exakt so aussehen, wie auf dem aktuellen Shop. Das ist unter anderem dem geschuldet, dass einige dieser Komponenten sich auf das Standardstyling von Salesforce berufen, welches allerdings undokumentiert geändert werden kann. Diese zwei Änderungen werden dem bereits verwendeten Stylesheet hinzugefügt und sind somit auf jeder Seite angewendet. Nach hinzufügen dieser Änderungen entsprechen die Komponenten nun nicht nur funktionell, sondern auch optisch ihren Gegenstücken im Aura-Shop.

```
h1, h2, h3, h4, h5, h6, p, ol, ul, dl, fieldset {
    margin: 0;
    padding: 0;
}

dxp_layout-column {
    position: relative;
}
```

Abbildung 12: Nötige Änderungen an der CSS-Datei

3.5 Auswertung

Mithilfe des erstellten Migrationskonzept sind die Produktdetailseite, der Header und Footer, sowie der Checkout-Prozess des Shops nun prototypisch auf die Lightning Web Runtime migriert. Für eine vollständige Migration muss an dieser Stelle zusätzlich ein Funktionstest der Komponenten erfolgen. Nach der Migration ist auf dem LWR-Shop eine messbare, sowie auch beim Nutzen des Shops deutlich bemerkbare Geschwindigkeits- verbesserung festzustellen. Die größten Verbesserungen sind hierbei insbesondere bei einem vollständigen Aufbau der Seite zu erkennen. Diese Situation tritt beispielsweise ein, wenn der Shop über einen direkten Link, wie etwa von einer Suchmaschine, aufgerufen wird. Somit sind diese Geschwindigkeiten insbesondere für die Search Engine Optimization der Seite relevant.

Nennenswerte Verbesserungen sind bei einem internen Seitenwechsel, in dem Fall von der Produktsuche auf die Produktdetailseite, ebenso zu finden. Dieser Prozess wurde bereits auf

der Aura Runtime insofern optimiert, dass nur vereinzelte Teilkomponenten bei einer Navigation durch den Shop neugeladen werden müssen. Aus diesem Grund sind die Unterschiede in dieser Situation etwas weniger auffällig, aber dennoch spürbar. Insbesondere durch das deutlich schnellere Anzeigen des ersten Inhalts wird das Gefühl eines kürzeren Ladeprozesses erweckt. Die Gesamtladezeit bleibt jedoch unverändert, was dadurch begründet werden kann, dass lediglich die Produktdetail-Hauptkomponente geladen wird und diese weiterhin verschiedenste Daten aus dem Backend abfragen muss.

	Aura Runtime	**Lightning Web Runtime**
Zeit bis zum Anzeigen des ersten Inhalts	1,9s	1,3s (-31%)
Zeit bis zum vollständigen Anzeigen des Inhalts	3,0s	1,6s (-47%)
Zeit bis zum vollständigen Laden der Seite	4,8s	2,5s (-48%)
Zeit bis zum Anzeigen des ersten Inhalts nach Auswählen eines Produktes	1,1s	0,6s (-45%)
Vollständige Ladezeit nach Auswählen des Produktes	1,7s	1,7s (0%)

Tabelle 1: Geschwindigkeit der Produktdetailseite auf der Aura Runtime und LWR

Die Geschwindigkeiten in Tabelle 1 wurden jeweils mit Hilfe des Google Chrome „Lighthouse“ Tools erfasst. Dabei ist jedoch anzumerken, dass es sich hierbei um Geschwindigkeiten auf einer Scratch-Org handelt. Diese Scratch-Orgs dienen lediglich dem Zweck, dem Entwickler eine Basis zu bieten, um neue Features zu entwickeln und werden deshalb nicht von Salesforce auf reine Geschwindigkeit optimiert. Auf einer Sandbox oder Produktions-Org, können diese Geschwindigkeiten deshalb deutlich abweichen, die relativen Unterschiede zwischen den beiden Runtimes sollten aber dennoch ähnlich ausfallen. Diese Eigenschaft wurde bereits bei Aura Runtime Orgs festgestellt, weshalb davon auszugehen ist, dass sich LWR-Orgs dabei ähnlich verhalten.

4 Zusammenfassung

In dieser Arbeit wurde ein Überblick über eine Migration eines Salesforce Shops von der Aura Runtime, auf die Lightning Web Runtime gegeben. Dafür wurden einführend beide Runtimes erläutert und verglichen. Aus diesem Vergleich lässt sich ableiten, dass die Lightning Web Runtime ein angemessener Nachfolger der Aura Runtime darstellt, der die größten Schwachstellen im Bereich Geschwindigkeit und SEO beseitigt, allerdings noch nicht den vollen Funktionsumfang ihres Vorgängers besitzt. Es ist deshalb für vorhandene Shops empfehlenswert, wenn möglich, auf die LWR zu migrieren.

Anschließend wurde für solch eine Migration ein allgemeingültiges Konzept entwickelt. In diesem Konzept sind die nötigen Änderungen, sowie Alternativen für die nicht vorhandenen Funktionen schrittweise aufgeführt. Das Konzept geht außerdem darauf ein, wann mit einer Migration des Shops vorerst abgewartet werden sollte.

Im nächsten Schritt wurde das Konzept an Teilkomponenten des Shops angewendet. Diese Teilkomponenten umfassen die Produktdetailseite, den Header und Footer, sowie den Checkout-Prozess. Bei den ausgewählten Komponenten bestanden zunächst die größten Bedenken hinsichtlich einer möglichen Shop-Migration. Es wurde gezeigt, dass diese Komponenten ohne größere Probleme vollständig auf die LWR migriert werden können. Diese Bedenken sind daher ausgeräumt und somit sollte eine weitere Migration des Shops ebenfalls möglich sein.

Schlussendlich wurde die Geschwindigkeit des neuen LWR-Shops mit dem Aura-Shop verglichen. Dabei ist eine deutliche Geschwindigkeitssteigerung, sowohl bei der gesamten Seitenladezeit, als auch beim Navigieren des Shops festzustellen. In diesem Punkt ist es für einen Store also lohnend auf die LWR umzusteigen.

Insgesamt kann gesagt werden, dass die Lightning Web Runtime der bisherigen Aura Runtime in vielen Punkten überliegt und es deshalb empfehlenswert ist, zumindest eine Migration in Betrachtung zu ziehen. Aktuelle Limitierungen werden zukünftig von Salesforce weitestgehend beseitigt, was eine Migration somit weiter vereinfachen wird. Auch zum aktuellen Stand ist es, wie am Beispiel durchgeführt, mit einer geordneten Informationsübersicht in Form eines Konzeptes oder ähnlichem, problemlos möglich einen vorhandenen Aura-Shop auf die Lightning Web Runtime zu migrieren.

5 Fazit

Das Ziel der Arbeit bestand darin, ein Shop-unabhängiges Konzept zu erstellen, mit dessen Hilfe es möglich sein soll, einen vorhandenen Salesforce Shop von der Aura-Runtime, auf die Lightning Web Runtime zu migrieren. Die Herangehensweise dafür war gut gewählt, da zuerst die Unterschiede der beiden Runtimes erläutert wurden und im nächsten Schritt, im eigentlichen Konzept, eine Übersicht über die nötigen Änderungen und Voraussetzungen für eine Migration gegeben wurden. Da anschließend mittels dieses Konzeptes ohne Probleme kritische Teilkomponenten des Web-Shop auf die Lightning Web Runtime migriert werden konnten, gilt diese Zielstellung somit als erfüllt.

Durch gute Planung im Vorfeld und regelmäßige Kommunikation mit anderen des Projektes, konnte eine prototypische Migration der einzelnen Komponenten des Shops unaufwändiger und umfassender durchgeführt werden, als ursprünglich erwartet. Dies führte dazu, dass im Rahmen dieser Projektarbeit ein größerer Umfang an zu migrierenden Funktionalitäten bearbeitet werden konnte. Somit kann die geleistete Arbeit im Rahmen dieser Projektarbeit, für die Entwickler des Projektes und auch anderen Salesforce Entwicklern, zukünftig bei einer vollständigen Shopmigration eine noch größere Hilfe darstellen.

Literaturverzeichnis

[Sal23a] Salesforce, Inc.: Lightning Web Runtime - Get Started, 2023, https://developer.salesforce.com/docs/platform/lwr/guide/lwr-intro, Abgerufen am: 16.03.2023

[Sal23b] Salesforce, Inc.: Lightning Web Runtime for Experience Cloud, 2023, https://trailhead.salesforce.com/de/content/learn/modules/lightning-web-runtime-for-experience-cloud/get-started-with-lightning-web-runtime, Abgerufen am: 16.03.2023

[Sal23c] Salesforce, Inc.: Migrate Aura Components to Lightning Web Components - Salesforce Lightning Component Library, 2023, https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.migrate_introduction, Abgerufen am: 30.03.2023

[Sal23d] Salesforce, Inc.: Standard Components for the Build Your Own (LWR) Template, 2023, https://help.salesforce.com/s/articleView?id=sf.networks_byo_lwr_components.htm&type=5, Abgerufen am: 31.03.2023

[Sal23e] Salesforce, Inc.: Use Expressions in LWR Sites | LWR Sites for Experience Cloud | Salesforce Developers, 2023, https://developer.salesforce.com/docs/atlas.en-us.exp_cloud_lwr.meta/exp_cloud_lwr/advanced_expressions.htm, Abgerufen am: 31.03.2023

[Sal23f] Salesforce, Inc.: Custom Component APIs | B2B Commerce and B2B2C Commerce Developer Guide | Salesforce Developers, 2023, https://developer.salesforce.com/docs/atlas.en-us.b2b_b2c_comm_dev.meta/b2b_b2c_comm_dev/b2b_b2c_comm_display_lwc_apis.htm, Abgerufen am: 03.04.2023

[Sal23g] Salesforce, Inc.: Create a Custom Checkout Component for a B2B or B2C Store (LWR) | B2B Commerce and B2B2C Commerce Developer Guide | Salesforce Developers, 2023, https://developer.salesforce.com/docs/atlas.en-us.b2b_b2c_comm_dev.meta/b2b_b2c_comm_dev/b2b_b2c_comm_create_checkout_component.htm, Abgerufen am: 06.04.2023

[Sal23h] Salesforce, Inc.: Head Markup | LWR Sites for Experience Cloud | Salesforce Developers, 2023, https://developer.salesforce.com/docs/atlas.en-

us.exp_cloud_lwr.meta/exp_cloud_lwr/template_differences_markup.htm, Abgerufen am: 13.04.2023

[Sal23i] Salesforce, Inc.: Create Custom Layout Components | LWR Sites for Experience Cloud | Salesforce Developers, 2023, https://developer.salesforce.com/docs/atlas.en-us.exp_cloud_lwr.meta/exp_cloud_lwr/get_started_layout.htm, Abgerufen am: 14.04.2023

[Sal23j] Salesforce, Inc.: Flow Reference, 2023, https://help.salesforce.com/s/articleView?id=sf.flow_ref.htm&type=5, Abgerufen am: 18.04.2023