
Abschlussprüfung Winter 2016

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

# Dashboard Entwicklung

**Entwicklung eines flexiblen Dashboards zur Anzeige aktueller Shop-Verkäufe**

Abgabedatum: München, den 03.11.2016

**Prüfungsbewerber:**

Jacob Groß

<Straße>

<PLZ Ort>

**Ausbildungsbetrieb:**

Jochen Schweizer Technology Solutions GmbH

Rosenheimer Str. 145E

81671 München

# Inhalt

## Inhalt

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Verzeichnis der Listings

# Abkürzungsverzeichnis

API.....6, 7, 8, 10, 11, *Application Programming Interface*

JSON .....7, 9, 11, *JavaScript Object Notation*

TLS .....9, *Transport Layer Security*

EPK.....9, *Ereignisgesteuerte Prozesskette*

HTTP.....9, 11, *Hyper Text Transfer Protocol*

XML .....9, 11, *Extensible Markup Language*

npm .....10, 11, 12, 13, *Node Package Manager*

HTML .....10, 12, 15, *Hyper Text Markup Language*

FIFO .....11, *First In First Out*

URL .....11, *Uniform Resource Locator*

DOM .....11, *Document Object Model*

ORM.....11, *object-relational mapping*

JS.....11, 12, *JavaScript*

CSS.....12, 14, *Cascading Style Sheets*

# 1 Einleitung

In der folgenden Projektdokumentation wird der Verlauf des IHK-Abschlussprojektes, welches im Rahmen der Ausbildung zum Fachinformatiker für Anwendungsentwicklung entstanden ist, genauer beschrieben.

## 1.1 Projektumfeld

Im Rahmen des Projekts wird ein Dashboard für die Firma Jochen Schweizer Technology Solutions GmbH, eine der führenden Firmen rund um das Erlebnis, deren Ausbilder und Auszubildenden am Standort Rosenheimer Str. 145 e-f, 81671 München sitzen, entwickelt.

Dabei handelt es sich um ein internes Projekt, welches im normalen Arbeitsumfeld von Jochen Schweizer umgesetzt wird. Laptop, Raspberry Pi und TV-Monitore werden bereitgestellt.

## 1.2 Projektziel

Um die Verkäufe von Jochen Schweizer Erlebnissen zu visualisieren, soll ein Dashboard entwickelt werden. In diesem sollen die aktuellen Verkäufe des Online-Shops auf einer Landkarte anzeigt werden.

Zudem soll das Dashboard eine Plattform darstellen, die relativ einfach zu erweitern ist (von Entwicklern). Um das sicherzustellen, wird das Dashboard in Form einer Webseite programmiert. Da dort JavaScript benötigt wird, bietet sich Node.JS für das Backend an, welches eine hohe Verbreitung hat.

Sofern möglich, sollen auch sonstige Bestandteile des Entwickler-Alltags integriert werden, wie Slack[1] Nachrichten, Inhalte aus dem Projekt-Management System Jira und Informationen von Jenkins[2] (z.B. Status, Fehler, Warnungen) jeweils in Form eines Widgets auf dem Dashboard.

## 1.3 Projektbegründung

Die Motivation hinter dem Projekt sind die großen Smart TVs, die sich in der IT-Abteilung sowie an der Rezeption befinden.

Bis lang wurden sie nur für YouTube Videos genutzt und sollen nun mit sinnvollem Inhalt bestückt werden. Damit werden sie auch vorzeigbar für Gäste.

Durch die anderen Widgets soll erreicht werden, dass sich Entwickler nicht mehr gegenseitig blockieren, oder Fehler unentdeckt bleiben.

[1] https://slack.com/ Messenger App für Teams
[2] Software zur kontinuierlichen Integration

### 1.4 Projektschnittstelle

Das Dashboard muss mit dem Online-Shop kommunizieren um die aktuellen Verkäufe zu lesen.

Da keine entsprechende Schnittstelle vorhanden ist, muss diese auch im Rahmen des Projektes umgesetzt werden, um das Ziel zu erfüllen.

Die Schnittstelle soll einfach gehalten werden, um den Server des Online-Shops bei Stoßzeiten (z.B. Weihnachtsgeschäft) nicht zu überlasten.

Das Backend des Dashboards übernimmt die Sortierung und Konvertierung der Daten.

## 2 Projektplanung

In der Projektplanung werden benötigte Ressourcen, Phasen und der Ablauf des Entwicklungsprozesses geplant.

### 2.1 Projektphasen

Das Projekt fand zur regulären Tagesarbeitszeit im Zeitraum vom 10.10.2016 bis zum 23.10.2016 statt.

In Tabelle 1: Grobe Zeitplanung sind die einzelnen Phasen kurz zusammengefasst. Die detailliertere Zeitplanung ist in Tabelle 6 in Anhang A1 zu finden.

| Projektphase | Geplante Zeit |
|---|---|
| Analysephase | 4 h |
| Entwurfsphase | 13 h |
| Implementation | 35 h |
| Qualitätsmanagement | 4 h |
| Deployment der Applikation | 1 h |
| Dokumentation | 11 h |
| **Gesamt** | **70 h** |

**Tabelle 1: Grobe Zeitplanung**

### 2.2 Ressourcenplanung

Für das Projekt wird ein Sitzplatz mit Sicht auf die TV-Monitore, ein mit Windows 7 x64 eingerichteter Laptop und ein mit dem Internet verbundener Raspberry Pi 2 benötigt.

Nur das letztere muss beschafft werden, da der Auszubildende bereits einen Arbeitsplatz und Laptop besitzt. Gegebenenfalls greift der Auszubildende auf das Know-How des Teams zurück um die Anforderungen zu erfüllen.

Als bevorzugte „IDE“ wird Sublime Text 3 verwendet und als Versionierungssystem Git. Eine Dokumentation über den Online-Shop ist Online zu finden.

### 2.3 Entwicklungsprozess

Die Entwicklung erfolgte nach agiler Arbeitsweise mit Unterstützung eines Scrum-Masters.

Der agile Entwicklungsprozess ist in der Firma Standard und benötigt deswegen keine weiteren Fortbildungsmaßnahmen.

## 3 Analysephase

Nach der Planung wird eine Analyse durchgeführt um den Ist-Zustand zu ermitteln und wirtschaftliche Aspekte näher zu betrachten.

### 3.1 Ist-Analyse

Bislang ist es nur möglich im Backend des Jochen Schweizer Shops Verkaufsdaten einzusehen.
Diese können im Excel Format exportiert werden. Die exportierte Datei ist unübersichtlich und nicht schön anzusehen.
Die Verkaufsdaten enthalten Kundendaten, Anzahl der Produkte, welche Versandmethode gewählt wurde und Name des Produkts. Dazu auch den Standardort des Verkaufs. Dies macht die Anzeige auf einer Landkarte sinnvoll.
Eine entsprechende Schnittstelle um diese Daten zu erhalten ist nicht vorhanden.

### 3.2 Wirtschaftlichkeitsanalyse

Da das Projekt einzigartige Anforderungen hat, ist keine Make or Buy Entscheidung notwendig.

Somit sind einzig die Projektkosten und die Amortisationsdauer von Bedeutung.

#### 3.2.1 Projektkosten

Die Projektkosten setzen sich aus Kosten für die Entwicklungszeit, verwendete Ressourcen und Personalkosten zusammen.

Im Folgenden die Berechnung, wie viel der Auszubildende pro Stunde kostet.

$$8\frac{\text{h}}{\text{Tag}} \cdot 220\frac{\text{Tage}}{\text{Jahr}} = 1.760\frac{\text{h}}{\text{Jahr}}$$

$$\dots\frac{€}{\text{Monat}} \cdot 12\frac{\text{Monate}}{\text{Jahr}} = \dots\frac{€}{\text{Jahr}}$$

$$\frac{\dots\dots\frac{€}{Jahr}}{1.760\frac{\text{h}}{\text{Jahr}}} \approx \dots\frac{€}{\text{h}}$$

Somit beträgt dieser *5,86€*. Die Durchführung dauert 70 Stunden. Für die Nutzung von Ressourcen[3] wird ein pauschaler Stundensatz von *15 EUR* angenommen. Für die anderen Mitarbeiter wird pauschal ein Stundensatz von *25 EUR* angenommen.[4] Eine Aufstellung der Kosten befindet sich in Tabelle 2 und sie betragen insgesamt *1.665,00 EUR*.

| Vorgang | Zeit | Kosten / Stunde | Kosten |
|---|---|---|---|
| Entwicklung | 70 h | *…. € + 15 € = 20,86 €* | *……. €* |
| Fachgespräch | 3 h | *25 € + 15 € = 40,00 €* | *120,00 €* |
| Abnahme | 1 h | *25 € + 15 € = 40,00 €* | *40,00 €* |
| | | ***Gesamt*** | ***…… €*** |

**Tabelle 2: Kostenaufstellung**

### 3.2.2 Amortisationsdauer

Die Berechnung basiert auf der Annahme, dass eine Person die Excel Tabelle hätte auswerten und manuell Produkte zu Kundenstandort zuordnen müssen. Da diese Arbeit manuell nicht täglich passieren würde, wird von zweimal wöchentlich ausgegangen.

$$104\frac{\text{Tage}}{\text{Jahr}} \cdot 360\frac{\text{Minuten}}{\text{Tag}} = 37.440\frac{\text{Minuten}}{\text{Jahr}} \approx 624\frac{\text{h}}{\text{Jahr}}$$

Dadurch ergibt sich eine jährliche Einsparung von:

$$624\text{h} \cdot (25 + 15)\,\frac{€}{\text{h}} = 24.960\,€$$

Die Amortisationszeit beträgt also:

$$\frac{\ldots\ldots\,€}{24.960\frac{€}{Jahr}} \approx \cdots Jahre \approx \cdots Wochen$$

## 3.3 Nutzwertanalyse

Der Unterschied zwischen automatisiert und händisch durch einen Mitarbeiter ist in diesem Fall gravierend, da automatisiert ohne größeren Aufwand nahezu Echtzeit Updates garantieren kann, während manuell unter sehr großem Aufwand nur zweimal pro Woche dasselbe ausgibt.

Zudem ist das manuelle Erstellen mit deutlich mehr Aufwand verbunden und geht nur solange gut, wie der Mitarbeiter nicht krank wird. Ein Raspberry Pi ohne redundante

[3] Räumlichkeiten, Arbeitsplatzrechner etc.
[4] (Macke)

Struktur hat zwar keine totale Ausfallsicherheit (SD Karten Ausfälle, Fehler im Programm), aber fällt dennoch weniger wahrscheinlich aus als ein Mitarbeiter[5].

| Eigenschaft | Gewichtung | *Manuell* | *Automatisiert* |
|---|---|---|---|
| Aktualität | 8 | 1 | 8 |
| Erstellen/Handhabung | 10 | 1 | 10 |
| Ausfallsicherheit | 2 | 0 | 1 |
| Gesamt | 20 | 18 | 166 |
| **Nutzwert** | | **0,9** | **8,3** |

**Tabelle 3: Nutzwertanalyse, je höher desto besser**

### 3.4 Anforderungs-Analyse

Unter Berücksichtigung von Projektumfeld, Ziel und Begründung wurden von dem Auszubildenden und den Teams um die TV-Monitore herum Anforderungen in Form eines Lastenheftes definiert. In A2 findet sich ein Auszug davon.

### 3.5 Anwendungsfälle

Das Zusammenspiel zwischen Schnittstelle, Dashboard und Datenbank ist anhand eines Use-Case-Diagramms in Anhang A3 zu sehen.

## 4 Entwurfsphase

Nach der Analyse soll die Entwurfsphase dazu dienen das Projekt effizient umzusetzen. Ziel dabei ist es eine Skizze zu entwerfen, die zeigt wie die Software später aussehen soll und mit welchen Mitteln es umgesetzt wird.

### 4.1 Planung Online-Shop Schnittstelle

Unter Berücksichtigung der Prinzipien des Datenschutzgesetzes wurde darauf geachtet so viel wie notwendig, aber so wenig wie möglich des Kunden preiszugeben. Hierfür musste im Backend des Online-Shops recherchiert werden, welche Daten vorhanden sind und welche für das Dashboard bzw. die Landkarte von Nutzen sind.

Da der Online-Shop auf dem Demandware[6] (DW) System basiert, muss die Schnittstelle mit Demandware-Script umgesetzt werden. Diese basiert auf ECMAScript und ist somit JavaScript nah.

[5] (Statistisches Bundesamt)
[6] http://demandware.de – eCommerce Framework

Anschließend musste anhand der API Dokumentation geprüft werden, ob alle benötigten Daten auch durch die DW-API bereitgestellt werden. Verwendete Daten sind im Pflichtenheft unter A4 zu finden.

## 4.2 Recherche von Dashboard Frameworks

Um das Rad nicht neu zu erfinden und die Anforderung „flexible Erweiterung" zu gewährleisten, wurde eine Eigenentwicklung zwar in Betracht gezogen, aber als nicht sinnvoll gewertet, da die Entwicklung eines eigenen Modul-Systems zu zeitintensiv gewesen wäre.

Somit galt es also ein passendendes Framework zu finden, welches für Echtzeitkommunikation geeignet ist und sich einfach erweitern lässt.

Dadurch, dass DW-Script Ähnlichkeiten mit JavaScript besitzt, wurde besonderer Augenmerk auf Node.JS[7] gelegt, da dort WebSockets nativ unterstützt werden und es ressourcenschonender arbeitet als vergleichbare Programmiersprachen[8].

Somit hat das Framework Dashing, welches in Ruby geschrieben wurde, in der Kategorie Architektur keine Punkte erhalten. Ebenso wird Dashing nicht mehr gewartet und fällt damit hinter Mozaik.

Anhand der Entscheidungsmatrix in Tabelle 4 wurde für die Implementierung der Anwendung das Node.JS-Framework Mozaik ausgewählt.

Bevor diese Entscheidung allerdings getroffen wurde, hat der Autor die genannten Frameworks genauer untersucht und Beispielseiten aufgesetzt um eine genauere Übersicht über den Code und die Funktionsweise zu erhalten.

| Eigenschaft | Gewichtung | *Mozaik* | *Strapi* | *Dashing* | Eigenentwicklung |
|---|---|---|---|---|---|
| Dokumentation | 4 | 3 | 3 | 3 | 0 |
| Anpassbar durch Widgets | 4 | 4 | 0 | 3 | 4 |
| Tests | 2 | 2 | 2 | 2 | 2 |
| Programmiersprache/Architektur | 2 | 2 | 2 | 0 | 2 |
| Aktualität | 2 | 2 | 1 | 0 | 2 |
| Gesamt | 12 | 40 | 22 | 28 | 28 |
| **Nutzwert** | | **3,33** | **1,83** | **2,33** | **2,33** |

**Tabelle 4: Entscheidungsmatrix**

[7] Serverseitige Plattform zur Ausführung von JavaScript
[8] (Wikipedia)

## 4.3 Entwurf der Benutzeroberfläche

Das Framework Mozaik beinhaltet verschiedene „Themes“, welche „on the fly“ ausgetauscht werden können. Als Standard wurde „Snowwhite“ genommen, da dieses am besten in die räumliche Farbgestaltung um die Monitore herum passt.

Die Landkarte soll so viel Platz wie möglich einnehmen, um sämtliche Verkaufsregionen anzuzeigen. Daraus hat sich ergeben, dass ein rotierendes Dashboard am sinnvollsten ist. Mozaik unterstützt dies auch. So wird zuerst die Landkarte angezeigt und danach andere Dashboards mit Widgets, die z.B. für die Entwickler nützlich sind.

Durch den Webshop verkaufte Produkte sollen auf der Landkarte platziert werden. Je mehr an einem Standort verkauft wurde, desto größer soll die Markierung werden.

Bei einem Klick auf die Markierung sollen Details zum Verkauf angezeigt werden. Am Aussagekräftigsten ist ein Vorschaubild des verkauften Erlebnisses und der Titel. Auch interessant sind Name des Käufers, Uhrzeit und in welchem Webshop er eingekauft hat (Deutschland, Österreich oder Schweiz). Wenn vorhanden, sollen noch andere Daten angezeigt werden (ob Newsletter abonniert, Kundenkonto vorhanden usw.).

## 4.4 Planung des Service

Der Service stellt die Verbindung zwischen Landkarte und API Schnittstelle her. Damit alle Komponenten des Projekts möglichst atomar sind, wird der Service so gestaltet, dass er unabhängig vom Dashboard laufen und gestartet werden kann.

Das hat auch den Vorteil, dass nicht das gesamte Programm abstürzt, wenn es einen Fehler gibt.

Eine Aufgabe des Service ist, die Daten der Schnittstelle aufzubereiten (z.B. Integer, die in der JSON Ausgabe als String ausgegeben wurden wieder in Integer verwandeln) und in der Datenbank zu speichern.

Die andere Aufgabe ist, die Daten auch wieder auszugeben und damit wiederrum als Schnittstelle für Mozaik / die Landkarte zu fungieren.

## 4.5 Datenmodell

Als Datenbanksystem wird MongoDB[9] genutzt. Das bringt den Vorteil, dass JSON Objekte direkt in die Datenbank eingefügt werden können. Somit müssen die Objekte nicht umgewandelt werden und sind direkt nutzbar.

Es gibt zwei Tabellen. Eine speichert die Daten der Verkäufe: `Order`.

Die andere Tabelle `Count` speichert wie viele Verkäufe es pro Tag gab und den Trend im Vergleich zum Tag davor.

[9] NoSQL Datenbanksystem

Da die API Schnittstelle die Anzahl der Verkäufe einfach ermitteln und ausgeben kann, speichert die Datenbank die Anzahl der Einträge direkt. So wird vermieden, dass die Datenbank bei jedem Aufruf gefragt werden muss, wie viele „Order" für den jeweiligen Channel an dem gefragten Tag in der Datenbank gespeichert sind.

Die Tabelle Count fungiert also ähnlich wie ein Cache um Rechenaufwand zu reduzieren, da so auch keine großen Redundanzen entstehen.

Die Produkte in Order wurden bewusst nicht ausgelagert, da es nicht die Aufgabe des Service ist, Produkte zu verwalten. Somit entstehen zwar Redundanzen, aber hier wurde sich wieder dafür entschieden den Rechenaufwand zu reduzieren und höheren Bedarf an Speicherplatz zu akzeptieren (da der Speicherplatz im Raspberry einfacher zu erweitern ist, als eine schnellere CPU einzubauen).

Dies führt zu der in Abbildung 1: Datenmodell gezeigten Struktur.

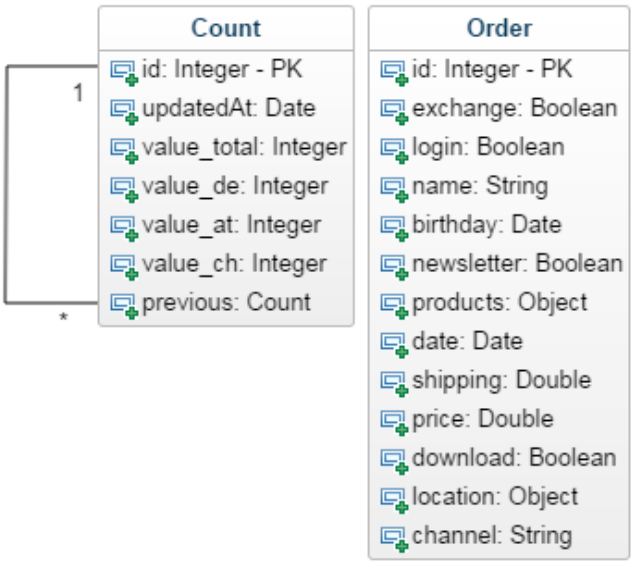


**Abbildung 1: Datenmodell**

## 4.6 Maßnahmen zur Qualitätssicherung

Für die Datenbank und die Services werden Modul-Tests erstellt, welche über die Kommandozeile ausgeführt werden können.

Vor dem Deployment testet der Autor auf generelle Funktionalität. Zudem sind die TV-Monitore für Entwickler gut sichtbar und es kann im Fehlerfall sofort reagiert werden.

Im gewissen Maße trägt zur Qualitätssicherung auch die wie in 2.3 beschriebene agile Arbeitsweise bei. Denn vor dem Beginn der Entwicklung wurde das Projekt in „User Storys" geschnitten. Dies ermöglicht eine bessere Übersicht über die umzusetzenden Aufgaben, da sich auf möglichst kleine Aufgaben konzentriert wird und somit der Überblick jederzeit gewährleistet ist.

### 4.7 Pflichtenheft

Zum Schluss der Entwurfsphase wurde ein Pflichtenheft erstellt. Dieses ist im Anhang A4 zu finden und beschreibt die Umsetzung der Anfangs definierten Ziele und Anforderungen.
Somit dient es als Leitfaden während der Entwicklung des Projektes.

## 5 Implementierung

Die in der Entwurfsphase entworfenen Skizzen werden nun genauer betrachtet und es geht an die eigentliche Umsetzung des Projektes, durch Programmcode.

### 5.1 Programmierung der Online-Shop Schnittstelle

Um die in 4.4 beschriebene EPK umzusetzen, wird eine neue Pipeline angelegt. Pipelines stellen einen Einstiegspunkt für HTTP-Anfragen dar. Um diese vor unbefugten Zugriff zu schützen, wird sie mit einem automatisch generierten Passwort, welches idealerweise monatlich gewechselt wird, geschützt. Zudem wird die „secure" Flag auf `true` gesetzt, damit Kommunikation ausschließlich per TLS stattfinden kann. Dadurch wird die Kommunikation zwischen API und Consumer verschlüsselt und sensitive Daten werden geschützt.

Die Abbildung 6: Grafische Darstellung der Pipeline zeigt das finale Design der Pipeline.

Um Käufe in einem bestimmten Zeitraum zu holen, wird das DW-Script „queryordersbydate" benutzt. Dieses ist schon vorhanden und liefert eine iterierbare Liste zurück, welche als Attribut in der global verfügbaren Variable `pdict` zu finden ist.

Die nun eigentliche Aufgabe ist die Liste in ein JSON Objekt umzubauen und dabei datenschutzrelevante Daten herauszufiltern. Der entsprechende Code ist in A8 zu finden.

In der kopfgesteuerten Schleife werden somit alle Verkäufe in dem Zeitraum abgearbeitet und in ein neues Array eingesetzt. Bei einem Verkauf können mehrere Produkte vorhanden sein, diese werden mit einer weiteren Schleife eingefügt. Dabei werden noch Attribute ergänzt, die für den Consumer der Schnittstelle nicht mehr so einfach einzufügen wären (z.B. ob es sich um ein Online-Ticket handelt, oder ob eine Verpackung gewählt wurde).

Die Standort Daten sind allerdings in XML vorhanden und werden als Attribut mit dem Typ String ausgegeben. Durch das Wegfallen der Konvertierung wird der Webserver entlastet.

## 5.2 Integration und Anpassung von Mozaik

Zu allererst wurde Github[10] nach Mozaik Widgets durchsucht. Andere Entwickler haben für Mozaik bereits Widgets geschrieben und somit konnten diese direkt benutzt werden.

Durch den Paket-Manager von Node.JS „npm" verlief die Installation von den Jenkins, Slack, iframe und Jira[11] Widgets sehr einfach.

Um Widgets anzuzeigen, müssen die genannten Node Module Mozaik bekannt gemacht werden. Das erfolgt über drei Konfigurationsdateien. In einer davon werden Größe und Länge sowie Position festgelegt.

Die schon installierten Module wurden dann auf eigene Bedürfnisse angepasst, wie z.B. das Jenkins Widget mit neueren Versionen kompatibel gemacht. Entsprechende Änderungen wurden dann im Zuge von Open-Source als Pull Request auf Github veröffentlicht.

Die anspruchsvollere Aufgabe war es, ein neues Widget zu erstellen, welches die Verkäufe auf einer Landkarte anzeigt.

Bei Mozaik Widgets erfolgt eine Trennung von Client (Browser) und Server. Für den Server wird ein API Client geschrieben, welcher lediglich dafür da ist, Daten von einer definierten Schnittstelle zu holen.

Im Browser agiert das Framework React[12] als Fundament. Somit wurde „react-leaflet" für die Kartenansicht verwendet. Dieses ist auf Github als Open-Source Projekt verfügbar und ließ sich auch mit npm installieren.

In React werden HTML-Elemente durch JavaScript definiert und verfügen über mehrere Methoden, die das Rendern des Elements steuern. Zudem können die Klassen beliebig erweitert werden. Mozaik sieht hier eine Methode vor, die das HTML-Element mit dem API Client verbindet.

Wenn der API Client also neue Daten holt, wird eine Methode im Widget aufgerufen mit den Daten als Parameter. Daraufhin ist es die Aufgabe des React-Elements die neuen Daten abzuarbeiten und auf der Landkarte einzutragen.

Die Verkäufe enthalten Längen- und Breitengrade des Käufers und können somit einfach auf der Karte platziert werden. Die Markierung erfolgt durch einen Kreis. Je mehr Verkäufe an einem Standort getätigt werden, desto größer wird der Kreis eingestellt.

Erst bei Klick auf einen Kreis wird ein „Popup" auf der Landkarte geöffnet und mit den in 4.3 beschriebenen Details befüllt. Da TV-Monitore, die das Dashboard im Browser

[10] http://github.com – Plattform, die Git Repository zur Verfügung stellt
[11] Webanwendung für Projektmanagement
[12] JavaScript Framework von Facebook

laden werden, über wenig RAM verfügen, ist diese Maßnahme sinnvoll um keinen Speicher zu verschwenden. Gleichzeitig heißt das auch, dass nur ein Popup zur selben Zeit sichtbar ist.

Falls kein Klick erfolgt, wird nach und nach ein Popup geöffnet. Das geschieht nach dem FIFO Prinzip. Um genau zu erkennen, wo ein Verkauf getätigt wurde, wird auch an den Standort herangezoomt. Da Längen- und Breitengrade nur auf die Stadt genau sind, wird die Privatsphäre nicht verletzt.

Nachdem sich ein Popup schließt, werden Referenzen auf die DOM-Knoten entfernt um Memory Leaks zu verhindern.

Die Kreise und Popups sind jeweils eigene React-Elemente und werden unabhängig voneinander gerendert. Dies führt gerade bei Geräten mit schwächeren Prozessoren zu erheblichen Geschwindigkeitsverbesserungen.

Zum Schluss muss das Widget nun noch in die Konfigurationsdateien eingebunden werden.

## 5.3 Implementierung des Service

Auch hier werden wieder npm Module zur Hilfe genommen. Einmal das Web-Framework Express, um die Daten für die Landkarte auszugeben. Dazu noch das Modul „Caminte“, welches für ORM zuständig ist. Dadurch erhält man die Flexibilität, falls nötig, später MongoDB gegen ein anderes System zu tauschen.

Für das automatische Abholen von Daten der DW-API, wird das Modul „cron“ benutzt.

Als weiteres Modul wird „xml2js“ benötigt. Denn wie erwähnt enthält das Attribut der Standortdaten XML. Mit Hilfe des Modules wird es dann zu JSON konvertiert.

Ansonsten wird noch „request“ verwendet, welches HTTP-Requests vereinfacht und „Bluebird“ um die Promises-Performance zu erhöhen.

In A9 Code des Service sieht man einen Ausschnitt der Klasse App. Die Methode `request` benutzt das Modul request um einen GET-Request an die in der Konfigurationsdatei festgelegte URL zu schicken. In der Callback Funktion werden die übrigen Methoden aufgerufen.

Die Definition der Datenbank Modelle erfolgt in JS-Dateien. Diese werden von Caminte gelesen und sind dann im Code nutzbar.

Die `giftmap` ist ein Objekt, welches die angegebenen Geschenkverpackungen Bilder-URLs zuordnet.

Der gesamte Code ist zudem in ES6 geschrieben. Damit wird zwar die Kompatibilität zu älteren Node Versionen reduziert, aber die neue JavaScript Version steigert die Flexibilität der Sprache.

# 6 Qualitätsmanagement

Für den Service wurden Tests geschrieben, die von dem JavaScript Test Framework „Jasmin“ ausgeführt werden. Die Tests werden über die Kommandozeile gestartet.

Das Testen der Landkarte und der Widgets hat der Entwickler übernommen, da entsprechende Tests aufwendiger zu schreiben gewesen wären und das Testen sehr einfach durchführbar war: Entweder die Widgets zeigen die richtigen Daten an und das Layout passt, oder eben nicht.

Zudem wurde auf Fehler und Warnungen in der Kommandozeile des Servers geachtet, sowieso in den Entwickler-Tools im Browser.

# 7 Deployment

Zuerst wurden mit Gulp[13] die CSS, JS und HTML Dateien von Mozaik konkateniert und minifiziert.

Danach wurden die Dateien des Dashboards und die des Service per FTP manuell auf den Raspberry Pi übertragen.

Darüber hinaus musste Node.JS und MongoDB installiert werden. Die Applikationen wurden dann mit Hilfe von npm über die Kommandozeile ausgeführt.

# 8 Dokumentation

Die Dokumentation erfolgte im Code, sowie in Jira. Darin ist beschrieben, wie Entwickler anderer Teams schnell neue Widgets erstellen und integrieren können.

Dort sind auch die Symbole erklärt, die man in den Popups auf der Landkarte sieht.

# 9 Fazit

Zum Schluss wird ein Überblick über den Verlauf des Projektes gegeben. Außerdem folgt, was während dem Projekt gelernt wurde und Ausblick wie es danach weitergehen kann.

## 9.1 Soll-/Ist-Vergleich

Entwickler, Scrum-Master und Mitarbeiter anderer Abteilungen haben die Landkarte gesehen und als nützlich empfunden. Somit wurde das Projektziel erreicht und die Teams um die TV-Monitore zufriedengestellt.

Der Zeitplan wurde bis auf die Entwurfsphase eingehalten. Die eine Stunde Differenz ist damit zu begründen, dass das Erstellen des Pflichtenhefts schneller ging als erwartet, durch gemeinsames Brainstorming des Auszubildenden und seines Teams.

[13] http://gulpjs.com/ - JavaScript Task Runner

| Phase | Geplant | Tatsächlich | Differenz |
|---|---|---|---|
| Analysephase | 4 h | 4 h | |
| Entwurfsphase | 13 h | 12 h | -1 h |
| Implementation | 35 h | 35 h | |
| Qualitätsmanagement | 4 h | 4 h | |
| Deployment der Applikation | 1 h | 1 h | |
| Dokumentation | 11 h | 12 h | +1 h |
| **Gesamt** | **70 h** | **70 h** | |

**Tabelle 5: Soll-/Ist-Vergleich**

## 9.2 Lessons Learned

Während des Projektes sind dem Autor einige Sachen aufgefallen, die er hier gerne zu Wort bringen möchte.

Ein großer Zeitfresser war das manuelle Deployment. Da Mozaik das Clientseitige JavaScript bündelt, muss nach jeder Änderung alles kompiliert werden. Es gibt zwar eine watch-Methode, aber diese funktioniert nur in der Entwicklungsumgebung und nicht auf dem Raspberry. Somit musste also nach jedem Bug Fix 20-30 Sekunden gewartet und dann alles hochgeladen werden, was schon mal bis zu 2 Minuten gedauert hat. Dies ist dem Ordner „node_modules" von npm geschuldet. Jedes Modul hat seine eigenen Abhängigkeiten und bringt somit wiederrum Module mit. Dies geht zwar zum Glück nicht unendlich, dennoch steigt die Größe des Ordners und die Anzahl der Dateien.

Ein automatisches Deployment System wie Jenkins, welches nach Git Commits automatisch baut, hätte da zu deutlichen Zeitersparnissen geführt.

Zudem hat Facebook (in Kooperation mit Google) einen alternativen Paket Manager für Node veröffentlicht, namens Yarn[14]. Dieser verspricht u.a. deutlich schnellere Installationszeiten und könnte damit auch das Problem der Node Module lösen.

Dadurch, dass Mozaik ein Open-Source Framework ist, hat sich darum schon eine kleine Community gebildet, welche dem Autor deutlich Arbeit abgenommen hat. So gab es, wie erwähnt, schon für das Projekt wichtige Module.
Durchgeführte Änderungen wurden u.a. auch schon in das originale Projekt integriert und Vorschläge übernommen. Somit profitiert nicht nur einer von den Änderungen, sondern viele.

[14] https://code.facebook.com/posts/1840075619545360/yarn-a-new-package-manager-for-javascript/

Weiterhin war es schwer auf dem TV-Monitor zu debuggen. Hat etwas in der lokalen Umgebung funktioniert, musste es nicht heißen, dass es dann auch auf dem Monitor geht. Leider gibt es keine entsprechende Schnittstelle, was dazu führt, dass Fehlermeldungen erstmal mit „window.alert" ausgegeben werden müssen.

Dazu kommt, dass der Browser (basierend auf Webkit) laut Dokumentation das CSS3-Feature „Flexbox" beherrscht, was sich in der Praxis nicht bestätigt hat und somit andere Techniken verwendet werden mussten. Dennoch hat das Tool „Autoprefixer" dabei viel geholfen, welches vom Autor in den Gulp-Prozess von Mozaik eingebaut wurde und nun auch im Original zu finden ist[15].

## 9.3 Ausblick

Zukünftig ist geplant die Microservices vom Raspberry Pi in die Microservices-Cloud-Infrastruktur zu migrieren, um die in der Datenbank gespeicherten Daten nicht nur für das Dashboard zu verwenden, sondern zum Beispiel für das Auswerten von Standort-zu-Erlebnis oder im Online-Shop live anzeigen zu können, wie oft ein Erlebnis in den letzten Stunden / Tagen gekauft wurde.
Um dies umzusetzen, müssten die Services in Docker[16]-Container eingebettet werden.

Dies hätte zugleich den Vorteil, dass das Deployment automatisch durch Jenkins, der in der Infrastruktur integriert ist, passieren würde und somit manuelle Entwickler-Arbeit deutlich minimiert.

Zudem wäre es sinnvoll einen Cronjob einzurichten, der in der Datenbank-Tabelle Count duplizierten Inhalt sammelt, in eine neue Tabelle schreibt und darauf referenziert.
Dadurch kann Speicherplatz eingespart werden, auf Kosten von Rechenzeit. Dies wäre allerdings in der Cloud-Infrastruktur kein Problem mehr.

Fehlermeldungen, die der TV Browser produziert, sollten zukünftig nicht mehr per alert ausgeben werden, sondern in einem ansprechenderen Design als „Benachrichtigung" in einer Ecke.

[15] https://github.com/plouc/mozaik/pull/92
[16] https://www.docker.com/

# Literaturverzeichnis

Grashorn, Dirk. *Entwicklung von NatInfo – Webbasiertes Tool zur Unterstützung der Entwickler*. Vechta, 2010.

ISO/IEC 9126-1. „Software-Engineering – Qualität von Software-Produkten – Teil 1: Qualitätsmodell." *ISO/IEC 9126-1 2001*. Juni 2001.

Macke, Stefan. Oktober 2016. 1. Oktober 2016. <http://dieperfekteprojektdokumentation.de/>.

Statistisches Bundesamt. *Statistisches Bundesamt*. 2015. Oktober 2016. <https://www.destatis.de/DE/ZahlenFakten/Indikatoren/QualitaetArbeit/Dimension2/2_3_Krankenstand.html>.

*Wikipedia*. kein Datum. Oktober 2016. <https://de.wikipedia.org/wiki/Node.js>.

# Eidesstattliche Erklärung

Ich, Jacob Groß, versichere hiermit, dass ich meine Dokumentation zur betrieblichen Projektarbeit mit dem Thema

> *Dashboard Entwicklung – Entwicklung eines flexiblen Dashboards zur Anzeige aktueller Shop-Verkäufe*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

München, den 30.03.2017

_____

JACOB GROß

# Anhang

## A1 Detaillierte Zeitplanung

| | | |
|---|---|---|
| **Analysephase** | | **4 h** |
| 1. Durchführung Ist-Analyse | 2 h | |
| 2. Anforderungs-Analyse | 2 h | |
| **Entwurfsphase** | | **13 h** |
| 1. Planung Online-Shop Schnittstelle | 2 h | |
| 2. Recherche von Dashboard Frameworks | 4 h | |
| 3. Planung des Service | 1 h | |
| 4. Planung der Datenbank Struktur | 1 h | |
| 4. Entwurf der Oberfläche | 1 h | |
| 5. Erstellung Pflichtenheft | 4 h | |
| **Implementation** | | **35 h** |
| 1. Programmierung der Online-Shop Schnittstelle | 4 h | |
| 2. Implementierung des Dashboards | 31 h | |
| 2.1. Integration und Anpassung des Dashboard Frameworks | 9 h | |
| 2.2. Programmierung des Service zur Kommunikation zwischen Webshop und Dashboard | 22 h | |
| **Qualitätsmanagement** | | **6 h** |
| 1. Erstellen von Modultests | 1 h | |
| 2. Funktionstest | 1 h | |
| 3. Fehlerbehebung | 4 h | |
| **Deployment der Applikation** | | **1 h** |
| **Erstellen der Dokumentationen** | | **11 h** |
| 1. Projektdokumentation | 9 h | |
| 2. Entwicklerdokumentation | 2 h | |
| **Gesamt** | | **70 h** |

**Tabelle 6: Detaillierte Zeitplanung**

## A2 Lastenheft (Auszug)

Es folgt ein Auszug aus dem Lastenheft mit Fokus auf die Anforderungen:

**Die Anwendung muss folgende Anforderungen erfüllen.**

.1. Entwicklung einer Schnittstelle im Online-Shop

- 1.1. Sensible Daten müssen anonymisiert werden.
- 1.2. Die Ausgabe muss im JSON Format erfolgen.
- 1.3. Ausgabe soll Server nicht belasten.

.2. Darstellung der Daten in Form eines Dashboards

- 2.1. Das Dashboard muss sich mit Hilfe der Schnittstelle eigenständig aktualisieren können.
- 2.2. Das Layout muss übersichtlich gestaltet sein, um es Gästen zeigen zu können.
- 2.3. Das Design soll auf Samsung Smart TVs optimiert werden.
- 2.4. Entwickler anderer Teams sollen das Dashboard flexibel erweitern können.

.3. Sonstige Anforderungen

- 3.1. Die Anwendung muss im Firmennetz über einen regulären Webbrowser erreichbar sein.
- 3.2. Die Anwendung soll jederzeit erreichbar sein.
- 3.3. Der Source-Code soll für jeden in der Firma einsehbar sein.

## Anhang

## A3 Use-Case-Diagramm

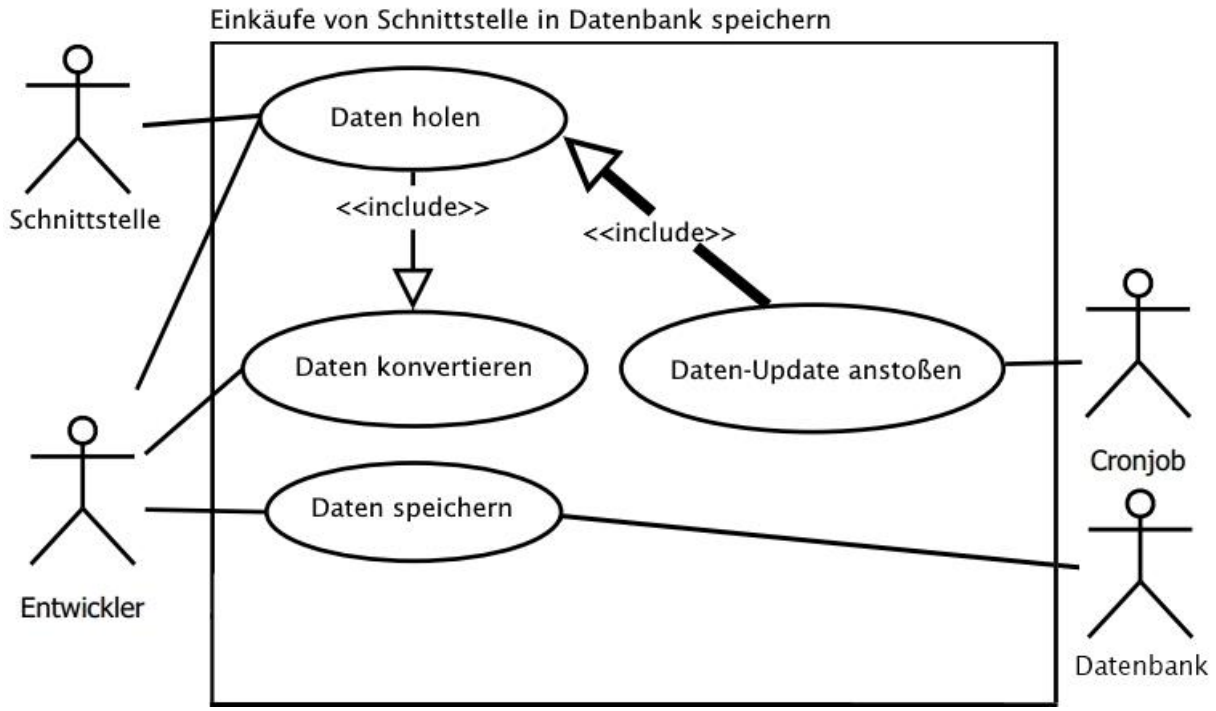


**Abbildung 2: Use-Case-Diagramm**

## A4 Pflichtenheft (Auszug)

### Zielbestimmung

.1. Musskriterien

1.1. Schnittstelle

- Sensible Daten werden anonymisiert, somit wird der Nachname weggelassen, sowie andere genaue Adressdaten des Kunden.
- Es werden nur Daten verarbeitet, die einen nicht auf den genauen Kunden schließen lassen. Länge- und Breitengrad ist nur auf die Stadt genau, es ist also zulässig das Geburtsdatum zu verwenden, da der Vorname allein nicht reicht um die Daten genau einer Person zu zuordnen.
- Die Ausgabe erfolgt im JSON Format.
- Der Standort ist als Attribut vom Datentyp String integriert und enthält XML Tags, damit der Server nicht durch die Konvertierung belastet wird.

1.2. Dashboard

- Der Service holt alle drei Minuten die aktuellen Daten von der Schnittstelle.
- Das Dashboard holt alle 30 Sekunden aktuelle Daten von dem Service.
- Die Karte ist auch für Gäste überschaubar und selbsterklärend.

## Anhang

- Das CSS und Layout wurde für TV-Monitore angepasst und ist von größerer Distanz lesbar.
- Das Framework Mozaik lässt sich durch Konfigurationsdateien schnell verändern und erweitern.

1.3. Sonstiges

- Die Anwendung ist im Firmennetz per Browser erreichbar.
- Die Hardware, auf der die Anwendung läuft, ist dauerhaft an Strom und Internet angeschlossen.
- Der Source-Code ist in einem Git Repository von Jira einsehbar.

## Anhang

## A5 Ereignisgesteuerte Prozesskette

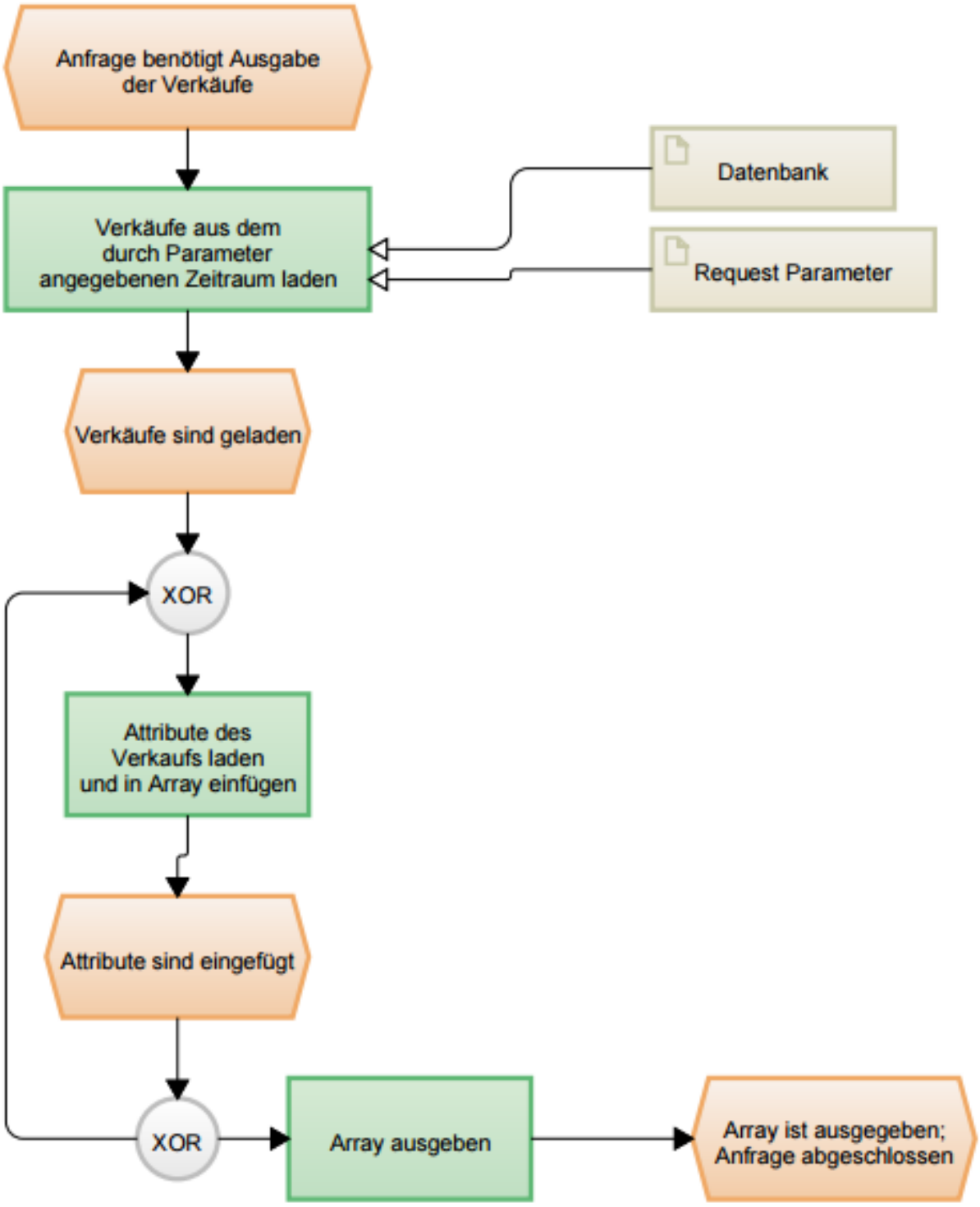


**Abbildung 3: Ausgabe der Verkäufe der Schnittstelle**

## Anhang

## A6 Bilder der Anwendung

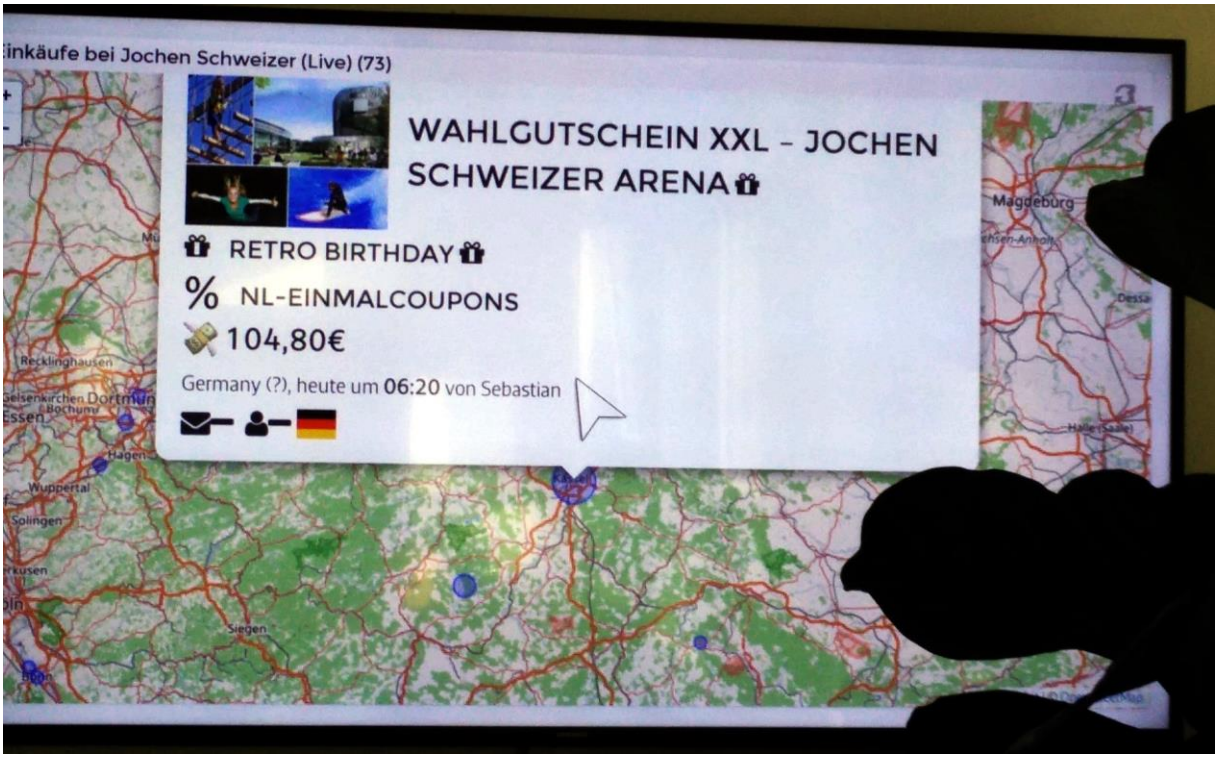


**Abbildung 4: Landkarten Ansicht auf einem Samsung Smart TV**

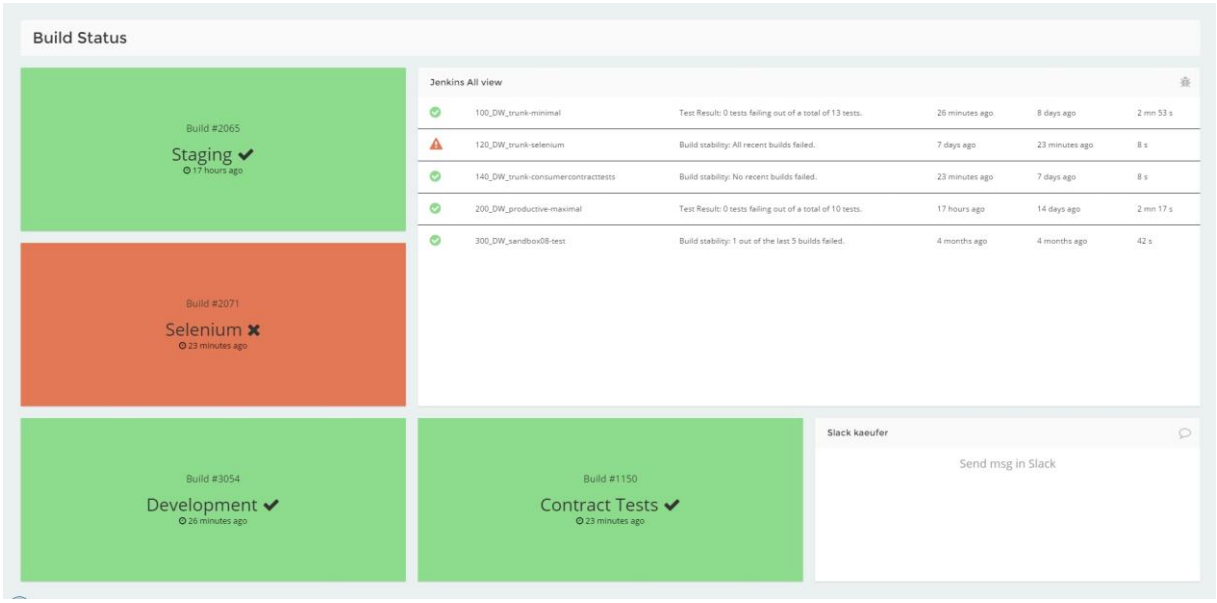


**Abbildung 5: Dashboard mit Widgets (hier: Jenkins Build Status)**

## A7 Pipeline

<Bild>

**Abbildung 6: Grafische Darstellung der Pipeline**

## Anhang

### A8 Code der Pipeline

Kommentare werden nicht gezeigt.

```
    <Code>
```

**Listing 1: Code der Pipeline**

### A9 Code des Service

Kommentare werden nicht gezeigt.

```
class App {
    request (channel, date) {
        var str = '&fromDate=' + date.getDate() + '.' +
    (date.getMonth() + 1) + '.' + date.getFullYear()


        request.get(config['JS' + channel.toUpperCase() +
    'API'] + str, {
            timeout: 55000,
            gzip: true,
            headers: {
                'User-Agent': 'Mozilla/5.0 (Windows NT 6.1;
    WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/51.0.2704.7 Safari/537.36',
                Accept: 'application/json',
                'Accept-Language': 'de-DE,de;q=0.8,en-
    US;q=0.6,en;q=0.4'
            }
        },
        function (error, response, body) {
            if (error || response.statusCode != 200) return
    this.emit('error', new Error(error))
            body = JSON.parse(body)
            this.parseOrders(body.orders, channel, date)
            this.saveCount(body.count, channel, date)
        }.bind(this)).on('error', function (err) {
            var date = new Date()
            date.setHours(0)
            this.lastOrders[channel] = { date: date }
```

## Anhang

```
            }.bind(this))
        }
...
```

**Listing 2: Ausschnitt des Service Code**

## A10 Klassendiagramm


**Abbildung 7: Klassendiagramm**