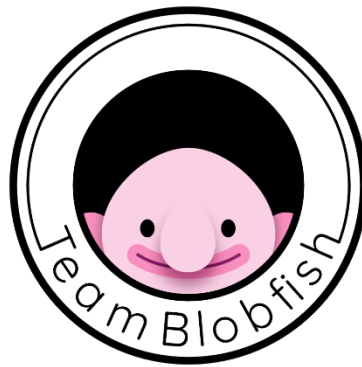


# Testkonzept V2.0

Von Team-Blobfish im Rahmen des SEP SS2021

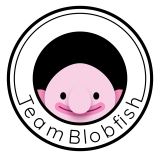
25.06.2021



Kontakt: [HSTeamBlobfish@gmx.de](mailto:HSTeamBlobfish@gmx.de)

Kunde: Capgemini

Verantwortliche Autoren: Maximilian Stinner und Florian Braasch



## Versionshistorie

Version	Datum	Vorgenommene Änderung	Autor
<b>2.0</b>	25.06.2021	Korrektur	Sven Reinemuth
<b>1.3</b>	25.06.2021	Einarbeitung des Feedbacks	Maximilian Stinner, Exoucia Mukubay
<b>1.2</b>	24.06.2021	Einarbeitung des Feedbacks	Maximilian Stinner
<b>1.1</b>	23.06.2021	kleine Anpassungen, Korrekturen, Formulierungen	Maximilian Stinner
<b>1.0</b>	7.06.2021	Anpassung an Kap. 3.1	Florian Braasch, Maximilian Stinner
<b>0.5</b>	02.06.2021	Korrektur	Sven Reinemuth
<b>0.4</b>	01.06.2021	Vervollständigung des Dokuments	Maximilian Stinner, Exoucia Mukubay, Florian Braasch
<b>0.3</b>	31.05.2021	Ausarbeitung der Kap. 1 - 4	Maximilian Stinner, Exoucia Mukubay, Florian Braasch
<b>0.2</b>	21.05.2021	Formeller Inhalt	Maximilian Stinner
<b>0.1</b>	17.05.2021	Dokumentstruktur angelegt	Maximilian Stinner

Die erste Nachkommastelle der Versionsnummer wird jedes mal wenn eine Änderung vollzogen wird hochgezählt. Ganzzahlige Versionsnummern stellen ein Inkrement dar, welches dem Kunden abgegeben wird.



<b>Versionshistorie</b>	<b>2</b>
<b>1. Einführung</b>	<b>4</b>
1.1 Einleitung	4
1.2 Testgegenstand	4
1.3 Ziel des Testkonzepts	4
1.4 Systemüberblick	4
<b>2. Testumfang</b>	<b>6</b>
2.1 Zu testende Elemente	6
2.2 Nicht zu testende Elemente	6
<b>3. Vorgehensweise</b>	<b>7</b>
3.1 Zeitplan	7
3.2 Komponententest	7
3.2.1 Backend-Test	7
3.2.2 Frontend-Test	10
3.2.3 Alexa-Test	11
3.3 Integrationstest	12
3.4 Performance	12
3.5 Usability	13
<b>4. Teststrategie</b>	<b>15</b>
<b>5. Testergebnisse</b>	<b>15</b>
<b>6. Tabellenverzeichnis</b>	<b>16</b>
<b>7. Abbildungsverzeichnis</b>	<b>16</b>
<b>8. Glossar</b>	<b>17</b>



# 1. Einführung

## 1.1 Einleitung

In dem Testkonzept von Team Blobfish werden alle Vorgehensweisen, Mittel und Abläufe in Bezug auf Tests für das Restaurant-Management-Systems "My-Thai-Star" dargestellt. Das Testkonzept nähert sich der Richtlinie "IEEE 829" an und bildet eine Grundlage auf der die Tests durchgeführt werden können.

## 1.2 Testgegenstand

Testgegenstand ist die Webanwendung "My-Thai-Star". Die Anwendung stellt ein Beispielprojekt vom Kunden "Capgemini" dar, welches mit dem [devon-Framework](#) entwickelt wurde. Eine grobe Darstellung und Zusammenfassung des Systems wird in [Kapitel 1.4 Systemüberblick](#) gegeben. Die von Capgemini gestellten Anforderungen bestehen primär aus der Weiterentwicklung und Erstellung von neuen Funktionen in der Weboberfläche, sowie der Implementierung und Anbindung der Spracherkennungssoftware Alexa.

## 1.3 Ziel des Testkonzepts

Das Testkonzept ist Planungsgrundlage für die Software-Tests. Mit dem Dokument wird sichergestellt, dass alle Betroffenen ein gemeinsames Verständnis über das Testvorgehen haben.

Durch das Testen sollen die Qualität des Produktes und die Zufriedenheit des Kunden angehoben werden. Zudem sollen Fehler so früh wie möglich durch das Testen identifiziert werden, damit das Projekt in dem zeitlichen Rahmen (siehe [Kapitel 3.1 Zeitplan](#)) möglichst ohne Fehler abgeschlossen werden kann.

## 1.4 Systemüberblick

Dieser Abschnitt gibt einen Überblick über das System, insbesondere hinsichtlich der Komponenten, die durch die Komponententests geprüft werden sollen.

Das Restaurant-Management-Systems "My-Thai-Star" setzt sich aus drei Schichten zusammen, die hierarchisch angeordnet sind (siehe Abbildung 1: Architektur Überblick). Die Drei-Schichten-Architektur besteht aus den folgenden Schichten: Präsentationsschicht, Logikschicht und Datenhaltungsschicht.

Die Präsentationsschicht besteht aus dem Angular-Frontend, das die dynamischen HTML-Seiten für den Browser des Users generiert, und dem Alexa Echo Gerät, das die Befehle des Users in Form von http-Requests an den Alexa Skill weiterleiten kann. Eine genauere Erklärung zu Alexa ist in der [Architekturdokumentation](#) (Kapitel 5.4 Ebene 2 – Alexa) vorzufinden.

Die Logikschicht befindet sich hauptsächlich im Spring Boot Backend. Das Spring Boot Backend kommuniziert über die REST-Schnittstelle mit dem Angular-Frontend und es werden http-Requests und http-Response verschickt. Der Alexa Skill und AWS Lambda kommunizieren ebenfalls über eine REST Schnittstelle mit dem Spring Boot Backend.

Durch die Hibernate Java-Bibliothek werden Daten aus der relationalen Datenbank auf Java-Objekte abgebildet. Hibernate greift über das JDBC-Treiber-Paket auf einen H2-Datenbankserver zu.

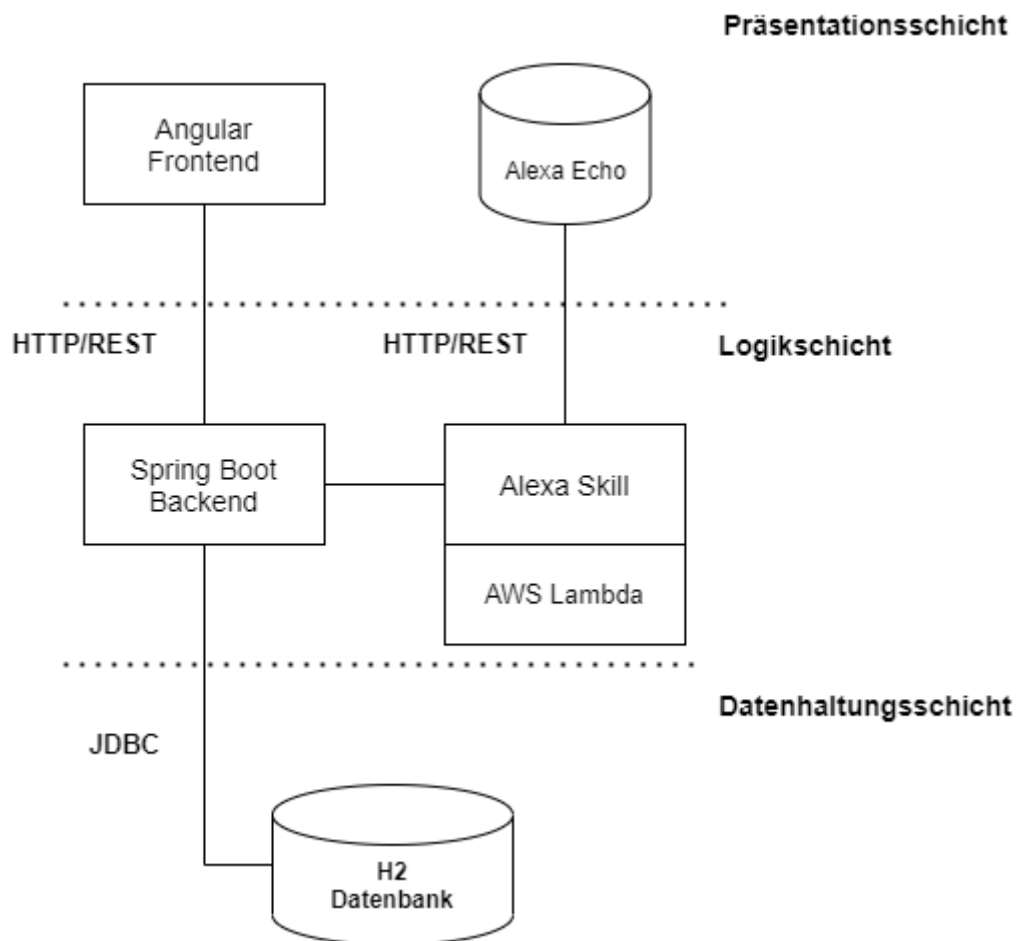


Abbildung 1: Architektur Überblick



## 2. Testumfang

### 2.1 Zu testende Elemente

Die Tests von Team Blobfish beziehen sich grundsätzlich nur auf die während des Projekts implementierten oder bearbeiteten Funktionen. Diese basieren auf den Use-Cases und Muss Anforderungen, welche mit dem Kunden vereinbart wurden und im [Pflichtenheft](#) aufgeführt sind.

Zu testende Elemente sind:

- Klassen und Methoden des Frontends
- Klassen und Methoden des Backends
- Funktionsweise von den Alexa-Skills
- Anbindung und Kommunikation aller Komponenten untereinander und mit dem vorliegenden System
- JDBC-Schnittstelle mit der externen Datenbank (Repository-Klassen)

### 2.2 Nicht zu testende Elemente

Nicht getestet werden alle schon vor dem Projekt implementierten und nicht abgeänderten Funktionen des "My-Thai-Star"-Systems.

Nicht zu testende Elemente sind:

- Bibliotheken und Frameworks (z.B. JDBC, Hibernate)
- Methoden, die lediglich auf Attribute eines Objekts zugreifen (get- und set-Methoden)
- Methoden, die schon vor dem Projekt im Backend implementiert wurden ([siehe Backend Tests](#))

## 3. Vorgehensweise

### 3.1 Zeitplan

Das Projekt begann am 09.04.2021 und endet mit einer Abschlusspräsentation am 05.07.2021.

Deadline der Abgabe des Quellcodes und jeglicher Dokumentation ist jedoch schon am 01.07.2021.

Ab dem 09.04.2021 wurde mit der Implementierung der geforderten, neuen Funktionen begonnen und wird spätestens bis zum 30.06.2021 fortgeführt. Parallel dazu, in demselben zeitlichen Rahmen, erfolgt die Durchführung und Dokumentation aller geplanten Tests.

Wichtig ist die Reihenfolge der Durchführung der verschiedenen Teststufen. Die Komponententests erfolgen prinzipiell vor den anderen Tests.

Integrationstests entstehen nach Implementierung einer Funktionalität, die auf einem Use-Case basiert, und dem Bestehen der entsprechenden Komponententests.

Im Anschluss zu den Integrationstests erfolgen Performance- und Usability-Tests.

### 3.2 Komponententest

Die Komponententests, bzw. Unit-Tests, stellen eine Teststufe dar, bei der sich auf das Testen der einzelnen Komponenten des Systems bezogen wird.

#### 3.2.1 Backend-Test

Das Backend des Restaurant-Management-Systems "My-Thai-Star" besteht ausschließlich aus Java-Klassen. Im [Architekturdokument](#) (Kapitel Ebene 2 - Backend) ist eine Übersicht über alle Java-Klassen mit Erklärung vorzufinden.

Im Java-Backend Projekt "mtsj-core" wird die gesamte Programmlogik und Funktionalität der Anwendung gespeichert. Darin werden die Java Klassen in den Paketen, "ordermanagement", "usermanagement" und "dishmanagement" mit JUnit-Testfällen geprüft.

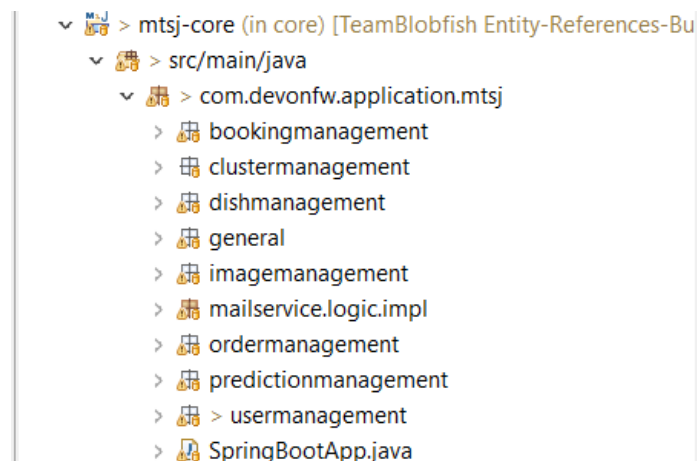


Abbildung 2: Backend - Paketstruktur

Die automatisierten Backend-Tests werden von Exoucia Mukubay durchgeführt. Das Framework für die automatisierten Test stammen aus dem Spring Framework.

Vor Beginn der Implementierung der JUnit-Tests werden bereits [Blackbox](#)-Testfälle erstellt, die dann begleitend zur Implementierung genutzt werden. Danach erfolgt das Testen mit JUnit.

Hier wird ein Beispiel dargestellt wie eine neu implementierte Methode getestet wurde:

Die Methode `setNewPaymentStatus()` aus der Klasse `"OrdermanagementImpl.class"` im Paket `"ordermanagement"` wurde implementiert, damit der Kellner auf dem Waiter-Cockpit (siehe [Architekturdokument](#) Kapitel 5.2 Ebene 2 - Frontend) den Bestellstatus der Kunden ändern kann.

```

585
586 @Override
587 public OrderEto setNewPaymentStatus(Long id, Long paymentstatus) {
588
589     Long invalidPaymentStatus = (long) 3;
590     if (paymentstatus >= invalidPaymentStatus) {
591         throw new WrongPaymentStatusException();
592     }
593     OrderEntity entity = getOrderDao().find(id);
594     entity.setPaymentStatus(paymentstatus);
595     OrderEntity resultOrderPayment = getOrderDao().save(entity);
596     LOG.debug("The payment status with id '{}' has been updated.", resultOrderPayment.getPaymentStatus());
597
598     return getBeanMapper().map(resultOrderPayment, OrderEto.class);
599 }
600

```

Abbildung 3: Testbeispiel-Methode

Zu dieser Methode wurden einige BlackBox-Testfälle in einer Excel Tabelle notiert:

Beispiel eines Blackbox-Testfalls:

Ordermanagement: setNewPaymentStatus(Long id, Long paymentstatus)					
Story: Der Kellner möchte den Bezahlstatus eines Benutzkontos ändern.					

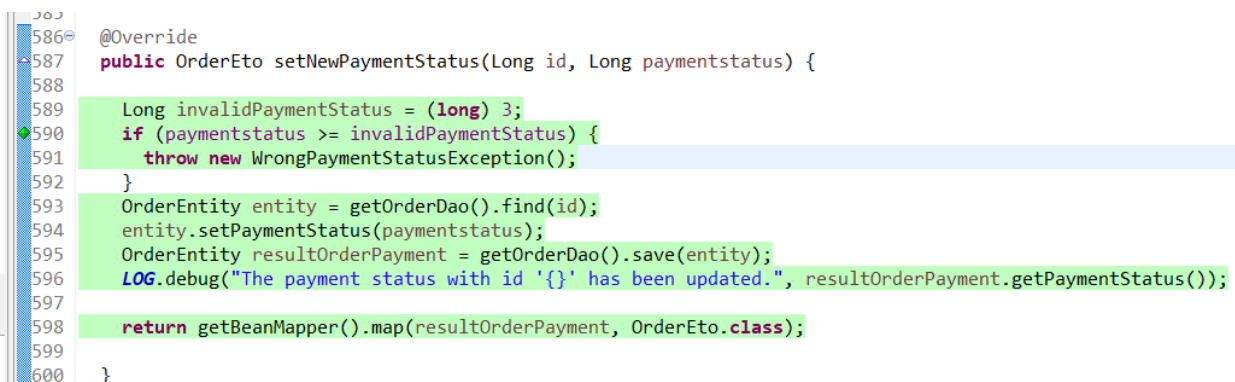
Abbildung 4: Blackbox-Testfall



Zunächst werden die Randwerte für den “paymentstatus” festgelegt. Dieser kann die Werte 0 bis 2 annehmen. Wenn der “paymentstatus” größer als 2 ist, wird eine Meldung angezeigt, dass diese nicht existiert, da es nur drei Bezahlstatus gibt.

Hier ist zu beachten, dass im Angular-Frontend die Randwerte schon festgelegt wurden und der Kellner über die Webanwendung nicht die Möglichkeit hat einen “ungültigen” *paymentstatus* auszuwählen, der nicht existiert und auch eine “ungültige” *id* für die Bestellung auszuwählen. Jedoch müssen diese Ausnahmefälle getestet werden.

Nach Abschluss der Implementierung einer Komponente wird diese dann durch manuelle Whitebox-Tests geprüft. Es wird kontrolliert, ob alle Anweisungen ausgeführt werden ([Anweisungsüberdeckung](#)) und alle möglichen Zweige durchlaufen werden ([Zweigüberdeckung](#)).



```

586 @Override
587 public OrderEto setNewPaymentStatus(Long id, Long paymentstatus) {
588
589     Long invalidPaymentStatus = (long) 3;
590     if (paymentstatus >= invalidPaymentStatus) {
591         throw new WrongPaymentStatusException();
592     }
593     OrderEntity entity = getOrderDao().find(id);
594     entity.setPaymentStatus(paymentstatus);
595     OrderEntity resultOrderPayment = getOrderDao().save(entity);
596     LOG.debug("The payment status with id '{}' has been updated.", resultOrderPayment.getPaymentStatus());
597
598     return getBeanMapper().map(resultOrderPayment, OrderEto.class);
599
600 }

```

Abbildung 5: Whitebox-Test

Hier ist zu sehen, dass die Anweisungsüberdeckungen wie if-Anweisungen automatisiert getestet wurden.

Ein JUnit-Test gilt als bestanden, wenn der Balken (siehe Abbildung 6) grün angezeigt wird. Das bedeutet, dass das Ergebnis, das erwartet wurde, richtig war und keine Fehlermeldung geworfen wurde. Wenn der JUnit-Test misslingen würde, würde der Balken rot anzeigen und eine Fehlermeldung und die entsprechende Ursache angezeigt.

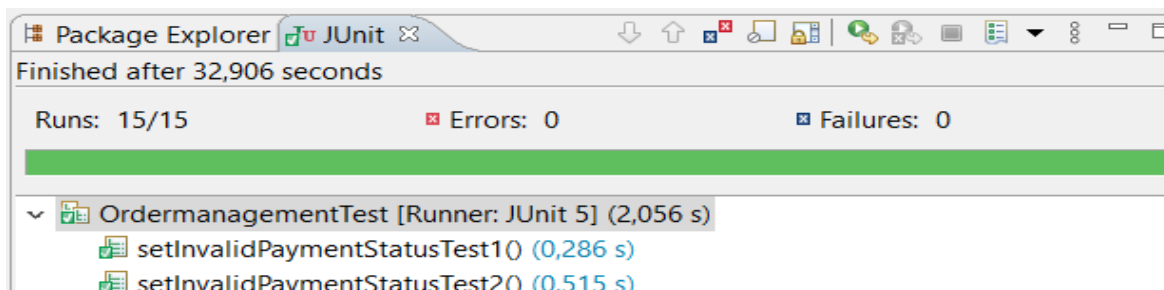


Abbildung 6: JUnit Testergebnis

### 3.2.2 Frontend-Test

Das Frontend, sprich Angular, besteht aus HTML-, CSS- und TypeScript-Dateien. UI-Elemente (z.B. Buttons, Listen) und Typescript-Methoden werden nach Implementierung mit automatisierten Unit-Tests auf richtige Funktionsweise geprüft, damit sichergestellt ist, dass sie ihr erwartetes Verhalten erfüllen. Diese Unit-Tests, welche auf dem Jasmine-Framework basieren, laufen über den Test-Runner Karma.

Geschrieben werden die automatisierten Tests von den Frontend-Entwicklern. Dabei werden schon vorliegende Testklassen erweitert, um die neu implementierten Funktionen abzudecken.

Beispiel eines Testfalls:

Im Waiter-Cockpit wurde ein Undo-Button implementiert, welcher die letzte Änderung im Waiter-Cockpit zurücksetzen soll. Diese Funktionalität wird im Code der folgenden Abbildung 7 überprüft:

```
it('should revert changes on click', () => {
  spyOn(component, 'undoLastChange').and.callThrough();
  fixture.detectChanges();
  component.applyChanges(orderData[0], 1, 1);
  expect(component.undoValues.length === 1).toBeTruthy();
  fixture.detectChanges();
  const btn = el.query(By.css('.undoButton'));
  expect(btn).toBeTruthy();
  click(btn);
  expect(component.undoLastChange).toHaveBeenCalled();
  expect(waiterCockpitService.setOrderStatus).toHaveBeenCalledWith(
    orderData[0].order.id,
    orderData[0].order.orderStatus,
  );
  expect(waiterCockpitService.setPaymentStatus).toHaveBeenCalledWith(
    orderData[0].order.id,
    orderData[0].order.paymentStatus,
  );
  expect(component.undoValues.length === 0).toBeTruthy();
});
```

Abbildung 7: Test-Code Undo-Button

Wird nach Durchlauf der Tests mit Karma "SUCCESS" in der Konsole angezeigt, so hat der Test bestanden.

### 3.2.3 Alexa-Test

Das Testen der Alexa Funktionalitäten wird manuell von Tony Friedrich durchgeführt. Anfangs wurde vom Projektteam ein Ansatz zum automatisierten Testen der Alexa-Komponente in Erwägung gezogen. Der Grund, wieso dieser Ansatz verworfen wurde, war primär, dass ein enormer zeitlicher Aufwand für die Erstellung der automatisierten Tests geschätzt wurde. Aufgrund der begrenzten Zeit und der Einsicht, dass Ressourcen in anderen Gebieten besser angebracht wären, wurde nach Absprache mit dem Kunden entschieden, dass die Komponententests von Alexa manuell durchgeführt werden.

Für das manuelle Testen ist ein Pfadüberdeckungstest angedacht. Dieses Verfahren beschreibt den Durchlauf von verschiedenen Pfaden, angefangen in einem Startknoten und beendet in einem Endknoten. Zur Veranschaulichung der möglichen Pfade wird ein Kontrollflussgraph angelegt.

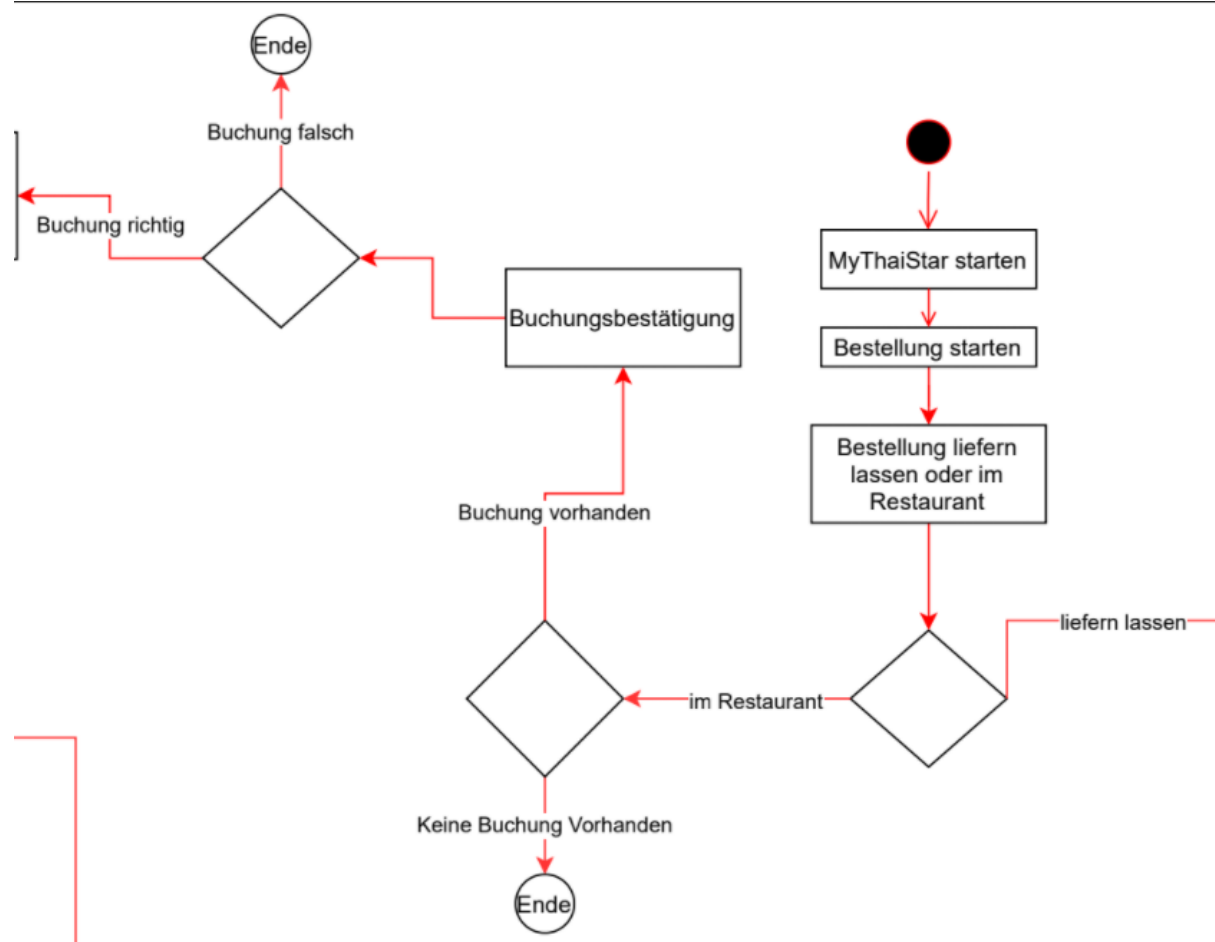


Abbildung 8: Kontrollflussgraph Ausschnitt



Die in Abbildung 8 gezeigte Darstellung ist ein Ausschnitt aus dem Kontrollflussgraph, der den Vorgang "Bestellung durchführen" (Use-Case 11 aus dem [Pflichtenheft](#)) über Alexa-Home beschreibt. In den Use-Cases und Anforderungen werden Alexa-Home und Alexa-Inhouse getrennt. Da die Vorgänge jedoch sehr ähnlich sind, wird nur Alexa-Home getestet und in einem Google-Docs-Dokument dokumentiert.

Jeder Testdurchlauf beginnt mit der Aufforderung an Alexa Essen zu bestellen. Nach Abschluss eines Pfades wird ein anderer durchgeführt. Wenn alle Pfade getestet wurden, ist der gesamte Test für den Vorgang "Bestellung durchführen" abgeschlossen.

### 3.3 Integrationstest

Im Integrationstest wird das Zusammenspiel der Komponenten des Systems untereinander getestet.

In unserem Fall erfolgt dieser nach dem [Bottom-Up-Prinzip](#).

Das bedeutet, dass zunächst einzelne, abgegrenzte Subsysteme definiert werden können und der Integrationstest auf diese angewendet wird. Das hat den Vorteil, dass Module/Funktionen, anders als beim Top-Down-Prinzip, frühzeitig implementiert und diese direkt danach mit dem vorhandenen System getestet werden können, unabhängig von der Vollständigkeit des Gesamtsystems.

Wichtig ist in unserem Fall, dass das Waiter-Cockpit und die Admin-Oberfläche problemlos mit dem schon vorhandenen "My-Thai-Star"-System funktionieren und damit ein gemeinsames funktionsfähiges System darstellen.

Besondere Aufmerksamkeit wird jedoch auch dem Zusammenspiel der Alexa-Komponente mit dem Backend zugeteilt, da Alexa eine externe Komponente darstellt, die noch nicht im System vorkam und komplett neu eingebunden werden musste.

Nach fertiger Implementierung von Use-Case-Funktionalitäten und dem Bestehen der jeweiligen Komponententests, werden von den Entwicklern Testfälle erstellt und anhand dessen manuelle Tests durchgeführt. Die vollzogenen Schritte und Ergebnisse werden vom Tester in einem Dokument festgehalten.

### 3.4 Performance

Um die Performance des Systems zu testen werden Stresstests durchgeführt, die Funktionen des Systems möglicherweise belasten könnten.

Dieser Test wird, wie schon im Kapitel Zeitplan erläutert, parallel zu den Usability-Tests von Entwicklern des Front- und Backends durchgeführt und umfasst Funktionen der Weboberfläche als auch Alexa.



Für die Weboberfläche werden folgende Bereiche getestet:

- Erstellen von Benutzerkonten in der Admin-Oberfläche (Use-Case 4 aus dem [Pflichtenheft](#)).  
-> wird getestet, indem eine große Menge an Konten erstellt wird und anschließend beobachtet wird, wie sich das System verhält.
- Bestellungen im Waiter-Cockpit (Use-Case 2 aus dem [Pflichtenheft](#)).  
-> wird getestet, indem eine große Menge an Bestellungen an das Waiter Cockpit gesendet werden und anschließend beobachtet wird, wie sich das System verhält, sprich ob man immer noch einwandfrei den Status editieren kann, Bestellungen löschen kann, etc.

Für Alexa wird folgendes getestet:

- Können einer Bestellung eine große Quantität an Gerichten hinzugefügt und dann problemlos abgeschickt werden? (Use-Case 8 & 11 im [Pflichtenheft](#))

## 3.5 Usability

Die Usability-Tests dienen dazu festzustellen, ob die Benutzbarkeit und Verständlichkeit der Anwendung gegeben ist. Geklärt wird diese Frage, indem projektfremde Personen hinzugezogen werden, die die Anwendung unter Aufsicht benutzen und ihre Meinung bezüglich der Bedienbarkeit abgeben.

Usability-Tests sind sowohl für die Weboberfläche als auch für Alexa angesetzt. Durchgeführt werden sie von Entwicklern in einem fortgeschrittenen Stadium der Entwicklung.

Für beide Bereiche werden sich Testaufgaben überlegt, welche die Testnutzer durchführen müssen. Testnutzer sind die Personen, die sich der Befragung/Test unterziehen.

Der Ablauf der Testdurchführung sieht wie folgt aus:

1. Einführung und Instruktion der Testpersonen
  - Hinweis geben: Der Prototyp, nicht der Testbenutzer, wird getestet
  - Testnutzer bitten, laut zu denken.
2. Bearbeitung der Testaufgaben
  - Nur eingreifen, wenn eine Aufgabe sonst unerledigt bleiben würde
  - keine suggestiven Fragen
  - Nicht über Testnutzer urteilen
  - Nur Moderator darf sich mit Testnutzer während des Tests unterhalten
  - Anmerkungen vom Testnutzer werden während des Testens dokumentiert und in der Nachbefragung angesprochen
3. Nachbefragung
  - Fragebogen austeilen

Für den Fragebogen wird eine Vorlage ([System Usability Scale \(SUS\)](#)) verwendet:



Aussage	stimme gar nicht zu		neutral		stimme voll zu
Ich kann mir sehr gut vorstellen, das System regelmäßig zu nutzen.					
Punkteverteilung:	0	1	2	3	4
Ich empfinde das System als unnötig komplex.					
Punkteverteilung:	4	3	2	1	0
Ich empfinde das System als einfach zu nutzen.					
Punkteverteilung:	0	1	2	3	4
Ich denke, dass ich technischen Support brauchen würde, um das System zu nutzen.					
Punkteverteilung:	4	3	2	1	0
Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind.					
Punkteverteilung:	0	1	2	3	4
Ich finde, dass es im System zu viele Inkonsistenzen gibt.					
Punkteverteilung:	4	3	2	1	0
Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherrschen lernen.					
Punkteverteilung:	0	1	2	3	4
Ich empfinde die Bedienung als sehr umständlich.					
Punkteverteilung:	4	3	2	1	0

Tabelle 1: SUS-Fragebogen



-> Die Auswertung der Punkteanzahl wird in [Kapitel 5 Testergebnisse](#) behandelt

## 4. Teststrategie

Da das Projektteam eine Deadline für die Abgabe des Produktes hat, passt sich die Teststrategie dementsprechend daran an.

Ein zeitiges, frühes Testen ist deshalb von Vorteil, weil der zeitliche Rahmen (siehe [Kapitel 3.1 Zeitplan](#)) eingehalten werden muss. Es wird deshalb versucht, nach der Implementierung einer neuen Funktion, diese auch direkt zu testen. Besonderer Fokus wird auf die Alexa-Usability gelegt, da diese Funktion Hauptaugenmerk des Projekts ist.

## 5. Testergebnisse

Alle Testergebnisse werden von den entsprechenden Testern direkt nach Durchführung der Tests in Dokumenten, welche im teaminternen [Google Drive](#) gespeichert werden, festgehalten.

Jedoch gehört zur Dokumentation nicht nur das Ergebnis des Tests, sondern auch die Testdurchführung, damit Fehlerursachen ersichtlich und reproduzierbar bleiben.

Der Umgang mit Testergebnissen sieht wie folgt aus:

- Bei Fehler/nicht bestandener Test:
  - Klassifizierung (z. B. nach Fehlerursache, Fehlerschwere).
  - angemessene Fehlerbeschreibung und Erläuterung.
  - Der Testfall bleibt offen und Testobjekt wird in der Entwicklung betrachtet, um den Fehler zu lösen.
- Bei Erfolg:
  - Testfall gilt als erledigt

Bei z.B. automatisierten Tests ist leicht zu erkennen, ob ein Test fehlschlägt oder besteht. Anders jedoch ist dies bei der Usability. Um eine Aussage über das Testergebnis eines Usability-Tests zu treffen, wird der SUS-Fragebogen verwendet.

Das Ergebnis der Usability-Tests hängt von der erzielten Punktzahl ab, die aus dem SUS-Fragebogen hervorgeht.

Auswertung des Fragebogens:

1. Die Punkte aller 10 Fragen werden addiert und mit 2.5 multipliziert. Daraus ergibt sich der SUS-Score, der eine Ausprägung zwischen 0 (schlechteste vorstellbare Anwendung) und 100 (beste vorstellbare Anwendung) annehmen kann.
2. Aus allen Tests wird der durchschnittliche SUS-Score ermittelt. Dieser Durchschnittswert kann dann als Prozentwert interpretiert werden:
  - 100% entsprechen einem perfekten System ohne Usability-Probleme.
  - Werte über 80% deuten auf eine gute bis exzellente Usability hin.
  - Werte zwischen 60% und 80% sind als grenzwertig bis gut zu interpretieren.



- Werte unter 60% sind Hinweise auf erhebliche Usability-Probleme.
- 3. Der Usability Test besteht dann, wenn ein Prozentwert von mindestens 70% erzielt wird.

Anmerkung: Der SUS-Score liefert keine Erkenntnisse darüber, *welche* Usability-Probleme vorliegen, es ist lediglich eine Methode herauszufinden, *ob* es Probleme gibt und wie groß das Problem ist. Zu den Testergebnissen gehören jedoch auch die auftretenden Probleme. Diese werden vom Tester in einer Tabelle dokumentiert:

TestNr.	Teilnehmer	Testaufgabe	Kritische Ereignisse
1	1	2	Testnutzer wusste am Anfang nicht, wie er eine Bestellung aufnimmt, ...

Tabelle 2: Beispiel-Tabelle für Usability-Dokumentation

## 6. Tabellenverzeichnis

Tabelle 1: SUS-Fragebogen	11
Tabelle 2: Beispiel-Tabelle für Usability-Dokumentation	16

## 7. Abbildungsverzeichnis

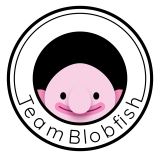
Abbildung 1: Architektur Überblick	5
Abbildung 2: Backend - Paketstruktur	7
Abbildung 3: Testbeispiel-Methode	8
Abbildung 4: Blackbox-Testfall	8
Abbildung 5: Whitebox-Test	9
Abbildung 6: JUnit Testergebnis	9
Abbildung 7: Test-Code Undo-Button	10
Abbildung 8: Kontrollflussgraph Ausschnitt	11





## 8. Glossar

Angular	Ist ein TypeScript-basiertes Front-End-Webapplikationsframework Quelle: <a href="#">Link</a>
Anweisungsüberdeckung	Anweisungsüberdeckungstests testen jede Anweisung mindestens ein Mal. Wurde jede Anweisung in einem Programm mindestens einmal ausgeführt, spricht man von vollständiger Anweisungsüberdeckung. Wurde vollständige Anweisungsüberdeckung erreicht, dann steht fest, dass kein toter Code (Anweisungen, die niemals durchlaufen werden) im Programm existiert.
Architekturdokument	Das Architekturdokument beschreibt die Architektur eines Systems.  Download Link: <i>Architekturdokument V.3.0</i> <a href="https://www.file-upload.net/download-14622053/Architekturdokumentation_V.2.0_TeamBlobfish.pdf.html">https://www.file-upload.net/download-14622053/Architekturdokumentation_V.2.0_TeamBlobfish.pdf.html</a>
Blackbox-Test, Blackbox-Testfälle	Black-Box-Testverfahren sind spezifikationsorientierte Testverfahren. Bei einem Black-Box-Test werden die Testfälle ausschließlich aus der Spezifikation des zu testenden Objekts abgeleitet, ohne dabei dessen innere Struktur, also Architektur und Code, zu berücksichtigen (- diese werden als „Black Box“ behandelt). Es wird also nur das von außen sichtbare Verhalten des Testobjektes beobachtet.  Ein Testfall besteht aus den Eingabedaten und den erwarteten Ausgabedaten
Bottom-Up-Prinzip	Es werden zunächst die einzelnen Programmbestandteile (z.B. Funktionen, Klassen, Module) definiert. Anschließend immer weiter zusammengesetzt bis das vollständige System erstellt ist.
devon-Framework/devonfw	Ist eine Softwareentwicklungsplattform, welche eine Reihe von Technologien und Best-Practises enthält und bereitstellt. Quelle: <a href="#">Link</a>
Front-/Backend	Das Backend bezieht sich auf Teile einer Computeranwendung oder eines Programmcodes, die den Betrieb des



	<p>Programms ermöglichen und auf die ein Nutzer der Applikation nicht zugreifen kann. Die Schicht oberhalb des Backends ist das Frontend und umfasst die gesamte Software oder Hardware, die Teil einer Benutzerschnittstelle ist.</p>
Google Drive	<p>Google Drive ist ein Filehosting-Dienst des Unternehmens Google LLC. Er ermöglicht Benutzern das Speichern von Dokumenten in der Cloud, das Teilen von Dateien und das gemeinsame Bearbeiten von Dokumenten.</p>
Hibernate	<p>Bei Hibernate handelt es sich um ein Framework zur Abbildung von Objekten auf relationalen Datenbanken für die Programmiersprache Java - es wird auch als Object Relational Mapping Tool bezeichnet. Quelle: <a href="#">Link</a></p>
http-Request, http-Response	<p>Im HTTP-Protokoll gibt es verschiedene Anfragemethoden, die es dem Browser ermöglichen, Informationen, Formulare oder Dateien an den Server zu senden. Der http-Request stellt eine Anfrage dar, die ein Nutzer an einen verbundenen Server sendet. Als Antwort auf einen derartigen Request sendet der Server eine http-Response.</p>
IEEE 829	<p>Die Definition IEEE 829 Standard for Software Test Documentation ist ein vom IEEE (Institute of Electrical and Electronics Engineers) veröffentlichter Standard, der einen Satz von acht Basis-Dokumenten zur Dokumentation von Softwaretests beschreibt.</p>
Pflichtenheft	<p>Das Pflichtenheft beschreibt in konkreter Form, wie der Auftragnehmer die Anforderungen des Auftraggebers zu lösen gedenkt.</p> <p>Download Link:  <i>Pflichtenheft V.3.0</i>  <a href="https://www.file-upload.net/download-14622057/Pflichtenheft-v3.0-TeamBlobfish.docx_1.pdf.html">https://www.file-upload.net/download-14622057/Pflichtenheft-v3.0-TeamBlobfish.docx_1.pdf.html</a></p>



REST-API	REST steht für Representational State Transfer, API für Application Programming Interface. Das Representational State Transfer beschreibt wie verteilte Systeme miteinander kommunizieren können
Jasmine-Framework	Jasmine ist eine freie Modultest-Bibliothek für JavaScript. Jasmine ist auf jeder JavaScript-fähigen Plattform ausführbar und erfordert keine Anpassung des Prüfsystems. Quelle: <a href="#">Link</a>
Karma	Karma ist ein Tool, welches Browser erstellt und die Jasmine Tests auf ihnen ausführt. Quelle: <a href="#">Link</a>
SUS	Mithilfe eines einfachen Fragebogens, der aus insgesamt zehn Fragen auf Basis von Likert-Skalen mit je fünf Optionen besteht, kann ermittelt werden, wie nutzerfreundlich eine Software wahrgenommen wird. Der SUS-Fragebogen enthält fünf positiv und fünf negativ formulierte Aussagen zur Usability des zu bewertenden Systems.  Quelle: <a href="#">Link</a>
White-Box Test	Ein White-Box-Test ist eine Software-Testmethodik, die den Quellcode eines Programms verwendet, um Tests und Testfälle für die Qualitätssicherung zu entwerfen. Beim White-Box-Test ist die Codestruktur dem Tester bekannt, im Gegensatz zum Blackbox-Test, einer Methodik, bei der die Codestruktur dem Tester nicht bekannt ist
Zweigüberdeckung	Der Zweigüberdeckungstest wird auch Entscheidungsüberdeckungstest genannt. Im Gegensatz zum Anweisungsüberdeckungstest durchläuft der Zweigüberdeckungstest alle Zweige.  Mit Hilfe des Zweigüberdeckungstests lassen sich nicht ausführbare Programmzweige aufspüren. Anhand dessen kann man dann Softwareteile, die oft durchlaufen werden, gezielt optimieren.