# Cloud Computing

## Was ist Cloud Computing?

Cloud Computing beschreibt die Bereitstellung von IT-Infrastruktur und IT-Leistungen über das Internet, wie beispielsweise Speicherplatz, Rechenleistung, Datenbanken, Software usw.

Cloud Computing entbindet den Nutzer sozusagen von der kostenintensiven Bereitstellung, Installation und Betreuung eigener Rechensysteme.

Einfach ausgedrückt handelt es sich also um das Outsourcen von Soft- oder Hardware ins Internet zu einem externen Cloud Computing Anbieter. Wichtig ist auch noch, dass man in der Regel nur für die Clouddienste bezahlt, die man auch tatsächlich nutzt.

## Warum wird es verwendet?

- Kosten

Es fallen keinerlei Investitionskosten für den Erwerb von Hardware und Software oder die Einrichtung und den Betrieb lokaler Rechenzentren an, Stromversorgung sowie IT-Experten zur Verwaltung der Infrastruktur spart man sich ebenfalls.

- Skalierung

Zu den Hauptvorteilen zählt natürlich auch die Skalierung, was bedeutet das die richtige Menge an IT-Ressourcen (beispielsweise eine höhere oder niedrigere Rechenleistung, Speicherkapazität usw.) genau dann bereitzustellen, wenn diese benötigt wird.

- Produktivität

Die Produktivität steigert sich, da Betrieb und Wartung von IT-Ressourcen (beispielsweise im Unternehmen) entfallen, sodass man sich auf die wesentlichen Geschäftsziele konzentrieren kann.

- Geschwindigkeit

Da die meisten Cloud Computing-Dienste als Self-Service-Angebote bereitgestellt werden, lassen sich selbst äußerst große Mengen an Computerressourcen innerhalb weniger Minuten bereitstellen, mit meist nur wenigen Mausklicks.

# Nachteile

- Sicherheit

Nutzt man Cloud-Dienste, muss man meist sehr persönliche Daten preisgeben. In der Cloud werden die Daten gespeichert und zum Teil auch weitergegeben und kopiert.

Da die Server vieler Cloud-Anbieter im EU-Ausland liegen, greifen die Datenschutzgesetze des Heimatlandes. Diese fallen in der Regel weniger streng aus als die EU-Richtlinien. So riskiert man einen schlechteren Schutz der persönlichen Daten.

- Cyberkriminalität

Da Rechner-Anlagen von Cloud-Anbietern häufig Ziel von Hacker-Angriffen sind, droht ein Verlust und damit ein möglicher Missbrauch der persönlichen Daten.

- Abhängigkeit

Wenn man viele Cloud Dienste nutzt, ist man natürlich vom Anbieter abhängig. Zur Folge kann das problematisch werden wenn z.B. der Anbieter durch eine Insolvenz handlungsunfähig wird.

# Wer sind die größten Anbieter für Cloud Services?

Die drei weltweit größten Anbieter sind:

- Amazon Web Services (AWS)

AWS punktet vorallem mit den fortschrittlichsten Services und dem tollen Support für große Unternehmen.

- Microsoft Azure

Azure kann vorallem durch den integrierten Support von Microsoft-Produkten punkten.

- Google Cloud

Die Google Cloud punktet vorallem durch die CO2-neutrale Infrastruktur, die nur halb so viel Strom verbraucht wie gewöhnliche und durch den Fokus auf Kubernetes.

# Welche Service Modelle gibt es?

- Infrastructure as a Service (IaaS)

Soll die grundlegenden Bauteile einer IT-Infrastruktur zur Verfügung stellen. Zu den typischen Diensten gehören virtuelle Maschinen, Netzwerke und Speicher.

- Platform as a Service (PaaS)

Hiermit entfällt die Verwaltung der Infrastruktur, man muss sich nicht mehr um Ressourcenbeschaffung, Kapazitätsplanung, Softwarewartung, Patching usw. kümmern.

- Backend as a Service (BaaS)

Hier werden für den Entwickler alle bzw. viele Backend Aspekte wie z.B. User Authentication, Database Management oder Cloud Storage einer Web- oder Mobilanwendung in die Cloud ausgelagert bzw. von dieser bereitgestellt, so dass man nur das Frontend schreiben und warten müssen.

- Software as a Service (SaaS)

Da wird eine Anwendung eingekauft, beispielsweise eine fertige Email Umgebung. Man kann E-Mails senden und empfangen, ohne die Funktionen des E-Mail-Programms verwalten oder die Server und Betriebssysteme, auf denen das E-Mail-Programm ausgeführt wird, warten zu müssen

# Firebase

## Was ist Firebase überhaupt?

Firebase ist ein Backend as a Service. Es ist eine Entwicklungs-Plattform, gehostet von Google, für mobile und Webanwendungen.

Sie stellt über ein Software Development Kit Tools und Infrastruktur zur Verfügung, die es einem Entwickler ermöglichen sollen, einfacher und effizienter Funktionen mittels Programmierschnittstellen bereitzustellen.

Man braucht beispielsweise keine Server zu managen, muss keine APIs entwickeln usw. Das macht alles Firebase: Firebase ist dein Server, deine API und dein Datastore.

# Features

- Authentication
- Realtime Database / Cloud Firestore
- Cloud Storage - Speichern von User Content z.B. Bilder
- Cloud Functions - man kann Backend Code laufen lassen, ohne einen eigenen Server managen zu müssen ("serverless")
- Hosting
- ML Kit - gebrauchsfertige API über die man z.B. Gesichter erkennen kann
- Crash Reporting - bugs finden
- ...

Das alles wird über eine Konsole im Web administriert.

# Konkurrenzprodukte zu Firebase bzw. Alternativen

- Back4app (bietet sehr ähnliche Funktionalitäten wie Firebase)
- Kuzzle (zusätzlich für IoT Backend)
- Couchbase (NoSql Database)
- BaasBox (bietet Backend nur für Mobile Entwicklung)

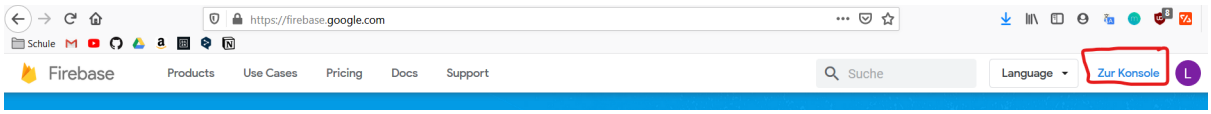
# Coding Beispiel

## Basics

Benötigt wird:

- Editor (z.B. VSCode)
- Nodejs (um die Libraries von Firebase zu installieren)

Um ein Firebase Projekt zu erstellen, geht man auf die Website von Firebase und klickt auf die Option "Zur Konsole" (wie im Screenshot markiert ist).




Ein Projekt ist nur ein Container für Ressourcen auf der Google Cloud Platform, wie z.B. die Datenbank, Dateispeicher, Webhosting und solche Dinge. Man kann alle App-Ressourcen von der Firebase-Administrationskonsole aus verwalten, und ein Projekt kann sich über mehrere Plattformen erstrecken. Firebase bietet Software-Entwickler-Kits für's Web, iOS, Android usw. an.

Um die Firebase Command Line Tools zu installieren muss folgendes Commando in der Konsole ausgeführt werden:

```
npm install firebase-tools -g
```

## Deploy App

Nun kann ich euch zeigen, wie einfach es ist ein Projekt mithilfe der Command Line zu erstellen und zu deployen. Man öffnet einfach einen leeren Ordner in VSCode, wo das Projekt erstellt werden soll und führt folgenden Befehl auf der Command Line aus:

```
firebase init hosting
```

Wenn man diesen Befehl das erste Mal ausführt, wird man einen Error bekommen der sagt, dass Kommando braucht authentication. Man führt einfach folgenden Befehl aus:

```
firebase login
```

Dann wird man auf die Firebase Seite weitergeleitet und muss sich nun mit seinem Account anmelden.

Es werden dann ein paar Files erstellt. Das erste ist firebase.json, welches alle Hosting Regeln beinhaltet und das .firebaserc file ist ein Skript welches das Projekt einfach identifiziert. Der public Folder mit der index.html Datei ist

unsere tatsächliche App. Im Header werden einfach die firebase web sdk's importiert um die ganzen Funktionalitäten von Firebase auch zu nutzen. Wenn man z.B. nur die Datenbank benötigt, dann reicht es wenn man nur das importiert und kann den Rest weggeben. (defer = sagt dem Browser dass das Skript nicht ausgeführt werden soll, bis die Seite geladen ist)

Um die App nun lokal auf port 5000 zu hosten:

```
firebase serve
```

Man sieht nun den Content der im index.html schon definiert ist. (Boilerplate code beim erstellen) Es ist nun ganz einfach möglich die App an eine live hosted URL zu deployen die über das Internet dann ereichbar ist:

```
firebase deploy
```

(Currently unavailable!)

Nun zurück zur index.html. Ich lösche jetzt einfach den ganzen Boilerplate Code, bis auf die imports. (script und style) Unten im Body tag verweisen wir dann auf eine app.js Datei, wo der ganze javascript code zu finden ist.

```
<script src="app.js">
</script>
```

Dann erstellen wir das File.

## Database

Jetzt eines der coolsten Features von Firebase und zwar die Datenbank. Seit Oktober 2017 gibt es zwei verschieden Datenbank Optionen, beide sind NoSql Datenbank wie z.B. Mongo. Es gibt zum einen eine real time database und zum anderen die sogenannte cloud firestore. Man kann beide gleichzeitig verwenden, aber meistens entscheidet man sich zwischen einen von den beiden und am besten findet man das heraus, indem man sich die Preise der beiden ansieht. Für die real time database zahlt man 5$ pro Gigabyte der gespeichert wird und 1$ pro Gigabyte der gedowloaded wird. Bei firestore hingegen zahlt man 18 Cent pro Gigabyte der gespeichert ist aber man zahlt auch für jede einzelne read and write operation für ein Document der DB. Das

bedeutet also, dass, wenn wir ein Datenmenge haben, bei der häufig gelesen wird sollte man die real time database verwenden aber für größere, komplexere relationale daten sollte man firestore verwenden. In den meisten Fällen wird aber Cloud Firestore verwendet.

Jetzt muss man die Datenbank auf der Firebase Website unter dem Register Database erstellen und im Testmodus starten, um gewisse Sicherheitsregeln zu deaktivieren damit am Anfang jeder drauf zugreifen kann. Die Datenbank selber ist so strukturiert, dass es sogenannte Collections mit Documents gibt. Es folgt dem Muster wie auch eine MongoDB aufgebaut ist. Das erste was wir tun ist, eine Collection zu erstellen, mit dem Namen posts. Eine Collection ist einfach ein Container für Documents. Danach fügen wir unser erstes Document hinzu. Jedes Document hat eine eindeutige ID, die man entweder selber erstellen kann, oder automatisch. In meinem Fall werde ich sie selber erstellen mit der ID "firstpost". Die aktuellen Daten für unsere App werden dann ganz einfach als Felder mit einem name-value pair gesetzt. Man kann diese Felder auch verwenden, um mit Queries nur bestimmte Documents zu erhalten aber dazu später mehr. Ein weiters Feature ist, dass die Benutzerauthentifizierung direkt an die Datenbank gebunden ist. Unter dem Tab Rules kann man Security Logik definieren, dass beispielsweise ein bestimmtes Set von Documents nur eingeloggte User sehen können, was natürlich für jede App wichtig ist, dass die Daten auch sicher sind. Man kann mit Firestore auch Indexes erstellen. Standardmäßig wird für jedes Feld bei einem Dokument ein Index erzeugt aber wenn man Abfragen durch mehrere Felder machen will, kann man sich auch seinen eigenen Index erstellen.

Innerhalb von app.js wollen wir dann sicherstellen, dass unser Code nach dem Laden des Dokumentinhalts ausgeführt wird, da Firebase erst danach verfügbar sein wird.

```
document.addEventListener("DOMContentLoaded");
```

Sobald der Inhalt geladen ist, können wir auf die Firebase-Ressourcen unter diesem Firebase-Namespace zugreifen.

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();
    console.log(app)
});
```

Wenn man nun "firebase.app()" aufruft und das Objekt dann auf der Konsole loggt, dann sollte man ein Objekt mit all unseren Firebase Anmeldedaten und anderen Dingen, die mit diesem Objekt verbunden sind, bekommen. Man muss das nicht tun, aber wenn man sicherstellen will, dass Firebase verfügbar ist, ist das eine Möglichkeit, das herauszufinden.

Nun zurück zu der App. Um Firestore zu verwenden, müssen wir den firebase Datenkbankimport durch Firestore ersetzen. Das erste das ich tun möchte, ist das Document was vorher erstellt wurde einfach abzurufen und zu lesen. Als erstes macht man eine Referenz zu Firestore als Variable.

Anschließend kann man mit dieser variable eine Referenz zu der Collection erstellen, in dem man den Namen der Collection als Parameter mitgibt und dann referenziert man noch das Dokument mit der ID, welches man haben will. Zu diesem Zeitpunkt haben wir eine Refernz auf das Document, welche wir verwenden könnten, um es zu erhalten, zu löschen, zu updaten oder einfach nur auf Änderungen zu horchen.

Wir möchten es einfach nur erhalten. Um es zu erhalten, ruft man einfach die asynchrone Methode get auf, welche einen Promise mit dem Document, dass wir in firestore gespeichert haben, returned. Die Daten werden als Variable gespeichert und danach gebe ich den Titel am browser windows aus. Wenn man die Seite refresht, sollte man das Ergebnis sehen.

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();

    const db = firebase.firestore();

    const myPost = db.collection('posts').doc('firstpost');

    myPost.get()
        .then(doc => {
            const data = doc.data();
            document.write(data.title + `<br>`)
        })
});
```

Aber einer der Hauptvorteile von Firebase ist, dass man auf Datenänderungen in Echtzeit reagieren kann. Wenn man Daten hat, welche mehrere User benutzen, und sich die Daten irgendwo ändern, werden alle User sofort benachrichtigt, wenn sich die Daten ändern. Das wäre ein eher kompliziertes Feature, wenn man es komplett von Anfang an programmiren müsste, aber wir können den vorigen Code ein bisschen abändern um dies zu erreichen.

Die Methode "get" wird ersetzt durch "onSnapshot" und anstatt einen Promise zu returnen, wird ein stream returned. Damit ersparen wir uns ".then()" und es wird einfach eine Callback Methode aufgerufen bei einer Änderung. Der Code im then Block kann in den Callback kopiert werden. Jedes Mal wenn sich das Document ändert, wird ein neues Document zu der Funktion emitted und man kann dann auf die Änderungen entsprechend reagieren. Ändert man nun den Titel in Firestore sieht man im Browser, dass es automatisch upgedated wird. Wenn man sowos für sei eigene App braucht, damit man also auf Änderungen sofort reagieren kann, ist das extrem hilfreich.

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();

    const db = firebase.firestore();

    const myPost = db.collection('posts').doc('firstpost');

    myPost.onSnapshot(doc => {
        const data = doc.data();
        document.write(data.title + `<br>`)
    })
});
```

Neben dem Lesen von Dokumenten können wir auch Dokumente aus unserem client seitigen Code updaten. Was ich hier tun werde, ist eine Formulareingabe einzurichten und jedes Mal, wenn sich diese Eingabe ändert, wird eine Update an firestore ausgeführt.

```
<h1 id="title"></h1>
<input onchange="updatePost(event)">
```

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();

    const db = firebase.firestore();

    const myPost = db.collection('posts').doc('firstpost');

    myPost.onSnapshot(doc => {
        const data = doc.data();
        document.querySelector('#title').innerHTML = data.title;
    })
});
```

Wir erstellen also eine Funktion names "updatePost" und machen dann wie zuvor einen Verweis auf das Dokument, aber dieses Mal rufen wir update auf und geben die Informationen weiter, die der Benutzer in das Formular eingegeben hat. Wenn die Daten aus Firstore aktualisiert werden, macht es etwas wirklich Cooles, nämlich optimistische Updates oder Latenzausgleich. Wenn Sie einen real time listener für die Daten haben, wie wir ihn zuvor eingerichtet haben, wird die Ansicht sofort aktualisiert, so dass der Benutzer nicht auf einen Server warten muss, um
eine Reaktion, die ein oder zwei Sekunden dauert zu verhindern die den Bentuzer vielleicht stört.

```
function updatePost(e){
    const db = firebase.firestore();
    const myPost = db.collection('posts').doc('firstpost');
    myPost.update({title: e.target.value})
}
```

In den meisten Fällen werden wir aber wahrscheinlich nicht
ein einzelnes Dokument, sondern versuchen, eine Sammlung von Dokumenten abzufragen. Um das zu testen muss man als erstes in der Firebase Datenbank mehrere Documents erstellen, in dem Fall products mit jeweils einen Namen und einem Preis. Ein weiters feature von firestore ist, dass
man mehrere Dokumente auf sehr aussagekräftige Weise abfragen kann.

Im Code werden wir zuerst einen Verweis auf die Produktsammlung machen und dann wenn wir eine Teilmenge der Dokumente in dieser Sammlung erhalten wollen, können wir die Methode "where" bei der Produkt Refernz aurufen. Der erste Paramter ist das Field bei dem wir filtern wollen, der zweite wie wir filtern sollen (also <, >, =, ≤,...) und der dritte Paramter ist der Vergleichswert für das Feld, in unserem Fall wollen wir alle Produkte bei denen der Preis größer als 10 ist.

Jetzt müssen wir genau das gleiche machen, was wir auch beim einzelnen Document gemacht haben. Nur hier wird nicht ein einzelnes returned, sondern mehrere. Man muss nun nur über die Documents iterieren um die Daten für jedes einzelne zu bekommen. Für jedes Document wird der Name und der Preis im Browser ausgegeben. Im Browser werden jetzt alle Produkte angezeigt, bei denen der Preis größer 10 ist. Es gibt noch einige Methoden wie z.B. "orderBy" um die Elemente irgendwie anzuordnen. Wenn man alle Produkte mit absteigenden Preis erhalten will, gibt man als zweiten Parameter einfach "desc"

mit. Es ist auch möglich die Anzahl der returned Documents zu limitieren mit der limit Methode, was wichtig sein kann wenn man mit einer rießigen Collection arbeitet.

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();

    const db = firebase.firestore();
    const productsRef = db.collection('products');

    const query = productsRef.where('preis', '>=', 10);
    //const query = productsRef.orderBy('preis', 'desc').limit(2);

    query.get()
        .then(products => {
            products.forEach(doc => {
                data = doc.data();
                document.write(`${data.name} mit dem Preis von ${data.preis} <br>`);
            })
        })
});
```

## Storage

Firebase Storage ist ein Cloud Storage von Firebase, der auch an das Firebase Projekt gebunden ist. Hier können Sie nutzergenerierte Inhalte wie hochgeladene Bilder und Videos oder ähnliches ablegen.

Das erste, was man tun sollte ist, in den Regel-Abschnitt zu gehen, und derzeit ist es so eingerichtet, dass nur authentifizierte Benutzer Dateien hochladen können, aber das wollen wir ändern. Im Moment werden wir jedem gestatten, Dateien in unseren Speicherbereich hochzuladen, zum testen.

Zurück in der index.html erstellt man eine Formulareingabe, für die Eingabe von Datein. Dann beim "onchange" event wird die "uploadFile" Funktion aufgerufen mit den File als Parameter. Es wird auch ein Bild Tag erstellt, um das upgeloadete Bild dann anzuzeigen, das derzeit keine Quelle hat, aber wir werden die Quelle asynchron setzen, nachdem der Upload abgeschlossen ist.

```
<h1>Storage Uplods</h1>
<input type="file" onchange="uploadFile(this.files)">
<hr>
<img id="imgUpload" src="" width="100vm">
```

In der Javascript Datei wird eine Refernz zu Firebase Storage erstellt, wie es vorher mit der Datenbank auch gemacht wurde. Wir werden einen Pfad in diesem Speicherbereich referenzieren, hier tun wir dass aber mit der Methode "child" und geben als Parameter den Namen des Bildes mit, dass upgeloadet wird. Mit diesem Verweis können wir entweder eine Datei hochladen oder eine bestehende Datei-URL herunterladen.

Unsere Formulareingaben gibt eine file list zurück, so dass man einfach das erste Elemnt der Liste mit dem Index 0 nimmt und dann mit der Refernz für das Bild die Methode "put" mit dem file aufruft, um das Bild upzuloaden. Es wird eine download URL und noch ein paar andere Daten zum Upload zurückgegeben. Wir wollen die URL für das Bild, welches wir eben upgeloaded haben, als "src" Attribut bei unserem Image Tag setzen.

```
document.addEventListener("DOMContentLoaded", event => {
    const app = firebase.app();
});

function uploadFile(files){
    const storageRef = firebase.storage().ref();
    const logoRef = storageRef.child('logo.png');

    const file = files.item(0);

    const task = logoRef.put(file);

    task.snapshot.ref.getDownloadURL().then(function(downloadURL) {
        const url = downloadURL;
        document.querySelector("#imgUpload").setAttribute("src", url);
      });
}
```

Wenn man nun ein File auswählt und dieses uploadet, sieht man das Bild im Browser und wenn man auf die Firebase Konsole zum Tag Storage speichert, sieht man den Eintrag mit dem Bild.

## Authentication

Nun möchte ich zeigen, wie Sie Firebase zur Verwaltung Ihrer Benutzerauthentifizierung verwenden können. In der Vergangenheit, auch heuto noch, war bzw. ist die Benutzerautorisierung immer ein sehr schwierig zu implementierendes Feature, aber mit Firebase können wir es mit ein paar Zeilen Code realisieren. Ich habe hier nur einen einfachen HTML-Button definiert, der die "googleLogin" Funktion in unserer Javascript Datei aufruft, wenn dieser angeklickt wird.

```
<button onclick="googleLogin()">
  Login With Google
</button>
```

Bevor wir uns tatsächlich einloggen können, müssen wir unter Authentifizierung in die Firebase-Konsole gehen und die gewünschten Methoden aktivieren. In diesem Fall wollen wir Google als OAuth-Provider verwenden, aber wir könnten auch E-Mail-Passwort, anonym, Twitter, Facebook, github usw. benutzen.

Jetzt schreiben wir eine Funktion, die den Benutzer einloggt. Das erste, was man tun muss, ist, den Provider zu finden, den wir verwenden wollen, was in diesem Fall der firebase auth Google auth provider.

Von dort aus können wir die firebase auth library referenzieren und die Methode "signInWithPopup" aufrufen mit dem Provider als Argument. Das ist eigentlich der ganze Code, den wir brauchen, um den Nutzer über Google anzumelden. Dieser Vorgang geschieht asynchron, was bedeutet, dass die Methode einen Promise zurückgibt. Wir sagen also: Melden Sie sich mit dem Pop-up an, dann, wenn uns die Methode das Result liefert, speichern wir den Nutzer in einer Variable. Dann kann man den Anzeigenamen in den Dom schreiben und wir loggen ebenfalls das user object auf die Konsole.

```
function googleLogin(){
    const provider = new firebase.auth.GoogleAuthProvider();

    firebase.auth().signInWithPopup(provider)
        .then(result => {
            const user = result.user;
            document.write(`Hello ${user.displayName}`);
            console.log(user)
        })
        .catch(console.log('error'))
}
```

Alle Benutzerinformationen werden aus dem tatsächlichen Google-Konto stammen, mit welchem man sich eingeloggt hat. Wenn man das nun testet und auf den Login Button mit der Google-Schaltfläche klickt, wird ein Pop-up-Fenster angezeigt, in dem sich der Nutzer in sein tatsächliches Google-Konto einloggt. Dann wird der Code ausgeführt, denn wir vorher im .then Callback definiert haben. Firebase verwendet Json-Web-Token zur Authentifizierung, was bedeutet, dass es einen
verschlüsselten Token gibt, das diesen Benutzer eindeutig identifiziert. Nun

muss man zurück auf die Firebase Konsole gehen und jetzt sieht man, dass unser Benutzer im Tab Authentication erstellt wrude. Dort können wir auch noch ein paar Änderungen vornehmen, wie z.B Passwort zurücksetzen, Konto löschen bzw. deaktivieren usw.

## Cloud Functions

Man kann sich die Cloud Functions wie einen eigenen nodejs-Server vorstellen, der bei Bedarf läuft. Anstatt also ein riesiges monolithisches Framework zu haben, das den gesamten Back-End-Code verwaltet, schreiben Sie Microservices, die eine bestimmte Cloud Function ausführt.  Man kann also Backend Code laufen lassen, ohne dass man einen eigenen Server managen muss. Wir können Cloud Functions initialisieren indem wir folgendes Kommando ausführen:

```
firebase init functions
```

Dieser Befehl erstellt ein neues Directory, nämlich functions. Dadurch wird der Backend Code isoliert. Man kann Funktionen auf verschiedene Weise auslösen, aber die Art und Weise, wie wir es tun werden, ist mit einem Firestore-Datenbank-Trigger. Das bedeutet, dass wir bei einem bestimmten Document pfad quasi zuhören und wenn dort ein neues Dokument erstellt wird, löst es diesen Funktionscode aus, der dann im Backend ausgeführt wird.

Was wir in diesem Fall tun werden, ist einfach eine Nachricht von der Funktion zurück an die Datenbank zu senden. Die Idee hier ist, Ihnen zu zeigen, wie Informationen zwischen Ihrem Backend-Code und Ihrer Datenbank ausgetauscht werden können. Dann können wir unsere Funktionen in der Datei index.ts definieren, und eine weitere einzigartige Sache, die wir hier tun können, ist die Verwendung des Firebase-Admin-SDK. Das adminSDK kann nur in einer Backend-Umgebung wie der Cloud-Funktion verwendet werden und umgeht alle Sicherheitsregeln, die Sie in Ihrer Firebase-App definiert haben.

Das erste was man tun muss, ist express aufzurufen und dann den Namen der Funktion zu definieren, in unserem Fall "sendMessage". Dann werden wir diese Funktion auf documents in der products collection verweisen. Die Klammern machen die ProductID zu einem Platzhalter bzw. Wildcard, so dass diese Funktion jedes Mal ausgeführt wird, wenn ein neues Dokument erstellt wird. Dann definieren wir noch den Trigger, in dem Fall "onCreate" weil die Funktion jedes Mal aufgerufen werden soll, wenn ein neues Product Document erstellt

wird. Innerhalb muss ein Promise zurückgegeben werden und mit den event objects erhaltet man Zugang zu Infos über die einkommenden Requests. Die Daten wie z.B. die ID des Documents oder den Namen erhalten wir durch die event objects. Nun empfängt die Cloud-Funktion einige Daten und möchte diese Daten als Aktualisierung desselben Dokuments zurücksenden. Um das zu machen verweisen wir auf firestore auf die collection mit der document id die wir erhalten haben. Das letzte was zu tun ist, ist ein update durchzuführen mit einer neuen Message, als neues Feld, beim Document.

```
const admin = require('firebase-admin');
admin.initializeApp(functions.config().firebase);

exports.sendMessage = functions.firestore
    .document('products/{productId}')
    .onCreate((snapshot, context) => {
        const docId = context.params.productId;
        const value: any = snapshot.data();
        const name: string = value.name;
        const productRef = admin.firestore().collection('products').doc(docId);

        return productRef.update({message: `Nice ${name}! - Love Cloud Functions`})
    });
```

Das letzte was zu tun ist, ist folgenden Befehl auszuführen:

```
firebase deploy --only functions
```

Dieser Befehl deployed unseren geschriebenen Code auf Firebase. Um das zu verifizieren kann man auf der Firebase Konsole unter dem Register Functions nachschauen, ob die Funktion zu sehen ist. Wenn man nun ein neues Produkt Document hinzufügt, sollte nach ein kurzer Zeit die Message als neues Feld erscheinen.