



Hochschule
Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

Hochschule Zittau/Görlitz
Fakultät Elektrotechnik und Informatik

Erstellung einer Todo App mit React und AWS CDK

Beleg

Modul / Lehrveranstaltung: Web Engineering 3
Dozent: Christopher Hilgner

Niklas Kaulfers
Matrikelnummer: 1064032

Abgabedatum: 24. Januar 2026



Inhaltsverzeichnis

1	Einleitung	1
1.1	Serverless	1
2	Tech Stack	2
2.1	Backend	2
2.2	Frontend	2
2.3	Entwicklungsumgebung	2
3	Aufbau und Architektur	3
4	Implementierung	4
4.1	Endpunkte	5
4.2	Logik	5
4.3	Beispiel: attach	5
4.4	Tests	6
4.5	Frontend	6
5	Building	7
6	Fazit	8



Kapitel 1

Einleitung

Ein durchaus häufig erstelltes Projekt in der Softwareentwicklung ist eine Todo-App. Anhand solcher können grundlegende Kenntnisse der Entwickler zur jeweiligen Umgebung gezeigt werden. Im Verlauf dieser Arbeit wird der Entwicklungsprozess einer solchen App im Serverless Kontext mit einem React Frontend erläutert. Hierzu entstehen Einblicke in den technischen Aufbau der Anwendung. Auch die genaue Komposition der Endpunkte und einige Codeabschnitte werden betrachtet.

1.1 Serverless

Um den genauen Aufbau dieser Anwendung zu verstehen, muss der Leser vorkenntnisse im Bereich des *Cloud Computings* und *Serverless* haben. Serverless bezieht sich auf Softwareanwendungen, welche in der Cloud veröffentlicht werden, der Entwickler sich jedoch nicht um den tieferen Aufbau des Servers kümmern muss. So ist der Begriff Serverless durchaus irreführend. Server existieren, werden jedoch von dem Hyperscaler, in diesem Fall AWS, verwaltet. Viele Service wie AWS Lambda oder AWS DynamoDB fallen in die Kategorie von Serverless.



Kapitel 2

Tech Stack

2.1 Backend

Das Backend besteht vollständig aus AWS Services. Um genau zu sein ApiGateway, Lambda, DynamoDB und Cognito. Diese haben jeweils eigene Aufgaben. So ist ApiGateway verantwortlich dafür, eine RestAPI zur Verfügung zu stellen. Alle Endpunkte der Anwendung sind innerhalb dieser API zu finden. Der Service Cognito schützt die Endpunkte, in dem er nur verifizierten Nutzern den Zugriff auf die API gestattet. Ohne Verifizierung hat der Nutzer des Frontends keinen Zugriff auf die API Endpunkte. Lambdas stellen Funktionen im Serverless Kontext da. In diesen kann die Logik der Anwendung mit eigenem Code definiert werden. Innerhalb der hier behandelten Anwendung wird die gesamte Logik in Typescript und mittels Nodejs geschrieben. Für Persistenz sorgt DynamoDB, die NoSQL Datenbank von AWS.

2.2 Frontend

Um mit dem Backend zu interagieren wird ein React Frontend verwendet. Mit React können Single Page Applications (SPAs) erstellt werden. Eine solche SPA besteht aus mehreren Komponenten, welche individuell, voneinander unabhängig ausgeführt werden können. Dieses nutzt für styling *Tailwindcss* und *MUI*. Tailwind stellt eine Sammlung an css Code da, welcher im Code einfach durch HTML Klassen genutzt werden kann. MUI hingegen ist eine Komponenten Bibliothek für React. Zusammen sorgen MUI und Tailwind für ein anschauliches Erscheinungsbild.

Für Nutzerverifizierung im Frontend wird *react oidc* genutzt. Gemeinsam mit einem von AWS bereitgestellten Login-Bildschirm sorgt diese Bibliothek für eine Absicherung der Anwendung.

2.3 Entwicklungsumgebung

Die gesamte Anwendung wurde im Umfeld von *IntelliJ IDEA* erstellt. Als Programmiersprache wird ausschließlich *Typescript* verwendet, für Frontend, Backend und Infrastruktur. Um die Anwendung ordentlich zur Verfügung zu stellen wird das Backend über AWS CDK erstellt. Dies ist ein Infrastructure as Code (IaC) Tool für AWS. In AWS CDK kann ein *Cloudformation Stack* mittels einer Programmiersprache definiert werden. Dieser Stack stellt ein Abbild der zu erwartenden Infrastruktur innerhalb der AWS Cloud da. Mittels dieses Stacks werden die einzelnen Serverless Service miteinander verknüpft. Auch werden hier Zugriffsrechte mit AWS IAM definiert.



Kapitel 3

Aufbau und Architektur

Die Anwendung stellt eine Todo-App da. In dieser kann ein Nutzer eigene Todos erstellen. Ein Todo besteht aus einem Titel, einer Beschreibung, einem Boolean-Wert und einer eindeutigen ID. Zudem einen Verweis auf in welche Listen sich dieses Todo befindet. Auch die Listen haben einen Verweis auf alle Todos, welche sich in dieser Liste befinden. Listen und Todos haben Methoden, um diese mit neuen Werten zu überschreiben. Auch besteht die Möglichkeit sie zu löschen. Man kann einzelne Parameter Updaten, hier ist der Verweis auf die jeweilige andere Liste jedoch nicht funktional. Das liegt daran, dass es sich bei der Datenbank nicht um eine relationale Datenbank handelt. Entsprechende Logik zu den Relationen muss somit in der Logik eingebaut werden.



Kapitel 4

Implementierung

Die Entwicklung fand in zwei alleinstehenden Projekten statt – dem Backend und dem Frontend. Vor allem von Bedeutung ist das Backend. Dort wird die Logik definiert. Das Backend besteht aus einem AWS CDK Projekt. In diesem werden verschiedene *Constructs* verwendet. Ein Construct definiert den Rahmen in welchem eine Komponente verwendet wird und ermöglicht so Objektorientiert Programmierung (OOP). In Aws gibt es 3 Level von Constructs, nämlich L1, L2 und L3. Diese definieren die Herkunft und Komplexität des Constructs. Die grundlegende Version sind L1 Constructs, diese stellen eins zu eins ein Abbild der Konfiguration für Cloudformation da. Hingegen sind L2 Ressourcen bereits angepasst von AWS, um sinnvoll und ohne zu viel Arbeit verwendet werden zu können. Mit L3 Ressourcen werden mehrere Ressourcen erstellt, welche sinnvoll miteinander interagieren¹. Das untenstehende Construct für ApiGateway ist ein in diesem Projekt verwendetes L3 Construct, in dem die verschiedenen Endpunkte, Definitionen für SwaggerIO und Integrationen definiert sind.

Auch werden in diesen Constructs die Trust-Relationen erstellt. Diese werden vom Service IAM genutzt, um Services Zugriff zu anderen Services zu erlauben. So hat ApiGateway ohne IAM Rollen nicht die nötigen Recht, um die Lambdafunktion mit der Entsprechenden Logik auszuführen.

```
1 export class ApiGatewayConstruct extends Construct {  
2     private readonly props: ApiGatewayProps;  
3     private readonly _restApi: RestApi;  
4  
5     constructor(scope: Construct, id: string, props: ApiGatewayProps) {  
6         super(scope, id);  
7         this.props = props;  
8         // ...  
9     }  
10 }
```

Abbildung 4.1: Beispiel eines L3 Constructs aus dem Projekt

Viele dieser Constructs stellt AWS selber zur Verfügung, jedoch kann der Nutzer eigene erstellen. Dies geschieht, indem eine Klasse mit der Klasse *Construct* erweitert wird. Mittels Benennung über den Parameter *id* lässt sich ein solches Construct leichter Debuggen. So sind zusammenhänge besser ersichtlich, falls es während des Deploys zu einem Error kommt. Als Error wird in der Konsole das verantwortliche Construct ausgegeben. Auch in der AWS Konsole (dem GUI von AWS) kann auf der Seite von Cloudformation das Construct mit dem Error genauer betrachtet werden. Dort ist von AWS auch ein vermutlicher *root cause* angegeben.

¹Amazon Web Services, *AWS CDK Constructs*, <https://docs.aws.amazon.com/cdk/v2/guide/constructs.html>, Accessed: 2026-01-01, 2026.



4.1 Endpunkte

Alle Funktionen dieser Anwendung sind mit eigenen Endpunkten im Backend ausführbar. Da die gesamte Logik auf Lambdafunktionen basiert erfolgen Lambdaintegrationen. Mittels dieser kann der Rückgabewert der Lambdafunktion direkt als Rückgabewert des Api-Endpunkts verwendet werden. Ein öffentlich ausführbarer Endpunkt muss innerhalb von AWS als eine *Stage* erstellt werden. Erst dann kann er über eine URL aufgerufen werden. Innerhalb des Codes ist dies durch das Construct *Deployment* geregelt. In diesem wird die Stage *prod* erstellt. Alle Kommunikation mit diesem Endpunkt gelingt nun über *Basis-URL* + *'/prod'*. Mittels *Model* werden die erlaubten Rückgabewerte und die Dokumentation definiert. Diese kann anschließend innerhalb der AWS-Konsole als SwaggerIO JSON exportiert werden. Eine solche Datei liegt in der Root des Projekts. Sie kann mittels Preview oder mittels OpenAPI-Generator in IntelliJ betrachtet werden. Um best practices zu folgen ist Cors aktiviert. Zugriff auf die API kann von `http://localhost:5173` und der Frontend-URL erfolgen. Alle Endpunkte sind zudem mittels eines *Cognito Authorizers* geschützt.

Es bestehen Endpunkte für alle CRUD-Funktionen für die Listen- und die Todotabelle. Einzelne Todos und Listen können über ein *POST-Request* über die Endpunkte */ToDo* und */List* erstellt werden. Für *UPDATE*, *DELETE* und *GET* wird die id des Todos oder der Liste als *path parameter* benötigt. Auch besteht mit dem */addToList*-Endpunkt eine Methode um ein Todo einer Liste hinzuzufügen. Eine Vielzahl an *OPTIONS*-Schnittstellen existieren für Cors-Preflight.

4.2 Logik

Zum publishen von Lambdafunktionen, welche die Logik beinhalten, wird das Construct *NodeJsFunktion* genutzt. Dieses erlaubt bundeling von Code und Runtime-Definitionen. Auch besteht die Möglichkeit eine NodeJsFunktion direkt mit Typescript in Aws zu integrieren, hierbei kann die Kompilierung sowohl mittels Ressourcen in der Cloud, als auch lokal, ausgeführt werden. Im Gegensatz dazu muss in diesem Projekt nach einer Abänderung der Lambdafunktion diese manuell kompiliert werden. Die Funktionen erhalten eine Rolle mit sehr großzügigen Rechten, diese müssen im Entwicklungsverlauf noch beschränkt werden.

Die Logik ist auch für entsprechende Datenbankoperationen zuständig. Grund hierfür ist die NoSQL Datenbank DynamoDB. Um beim DELETE eines Todos dieses aus allen Listen zu entfernen muss eine Transaktion erfolgen, welche das Todo aus dem Todo-Table löscht und in dem List-Table alle Listen durchgeht, in welchen das Todo sich befindet, um die Referenz aus diesen zu löschen.

4.3 Beispiel: attach

Der genauere Ablauf ist anhand eines Beispiels am verständlichsten. Dieser ist in der untenstehenden Grafik 4.3 nochmals visuell erkenntlich. Das Szenario stellt hier bereit, dass der Nutzer angemeldet ist und sowohl ein Todo, als auch eine Liste, vorhanden sind. Nun möchte der Nutzer das Todo in eine Liste einfügen. Dazu klickt er im Frontend auf den entsprechenden Button. Erstellt wird ein Post-Request auf */ToDo/addToList* mit einem Body und einem Header. Im Header befindet sich ein JWT-Token als Authorization, während im Body die Id der Liste und die Id des Todos gelistet sind. Der Request wird zuerst mittels Cors geprüft über ein Options-Request, ob die Origin-Adresse valide ist. Hier ist sie es. Anschließend wird der tatsächliche Post-Request an ApiGateway übergeben. An diesem ApiGateway ist ein Authorizer des Services Cognito angebracht. Dieser Authorizer überprüft das JWT-Token aus dem Header und stellt fest, dass der Nutzer ein angemeldeter Nutzer aus dem Userpool ist. Daraufhin triggered ApiGateway seine integrierte Lambdafunktion an diesem Endpunkt. Die Funktion liest den Body aus und stellt fest, dass beide Werte richtig übergeben wurden. In der Funktion wird eine Transaktion mittels des AWS Softwaredevelopmentkits (SDK) für DynamoDB erstellt. Darin werden die Ids in die entsprechenden Todos und Listen als Abhängigkeit eingetragen. Nachdem dies erfolgreich geschah wird von der Lambdafunktion eine Erfolgsnachricht zurückgegeben. Da die Lambdafunktion eine Lambdaintegration hat, wird das Resultat der Funktion so über ApiGateway an das Frontend weitergegeben, ohne noch einmal von ApiGateway überarbeitet zu werden. Das ist der Grund dafür, dass die Lambdas Headers mit Cors im Rückgabewert haben.

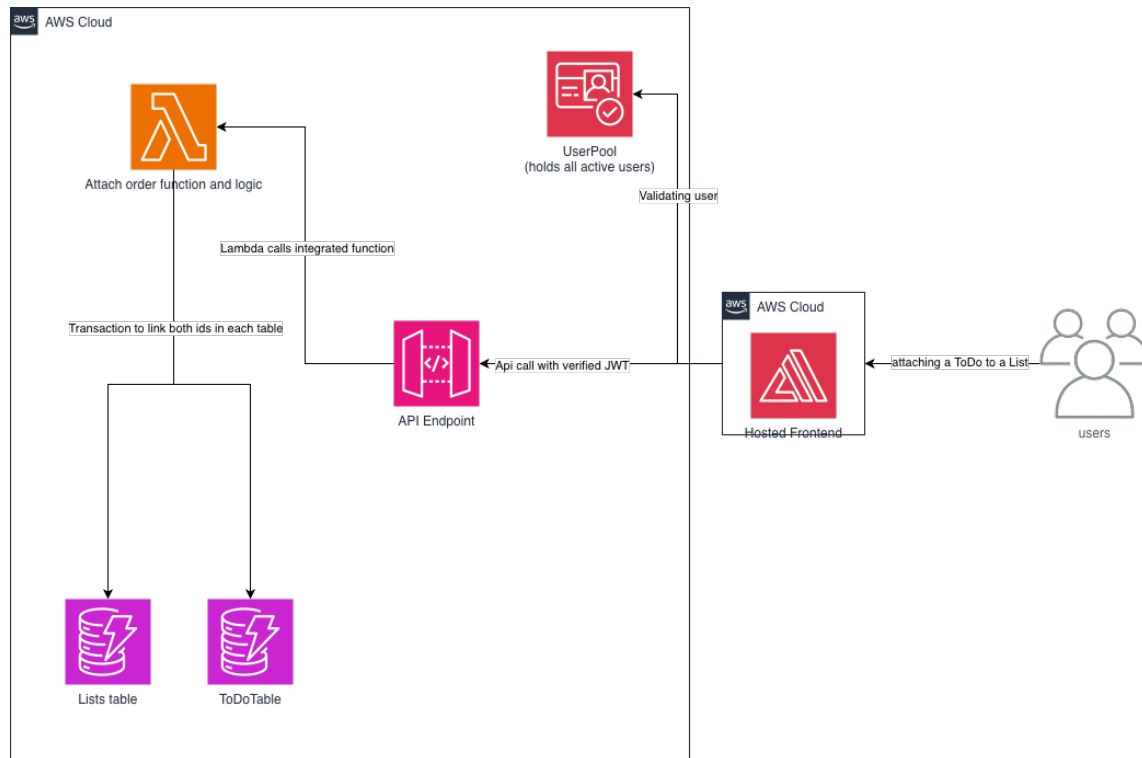


Abbildung 4.2: Ablauf des Hinzufügens eines Todos zu einer Liste

4.4 Tests

In dem Projekt existieren eine Vielzahl an verschiedenen Tests. Dazu gehören Unittests und Integrationstests auf Lambda-Ebene, Integrationstests auf CDK-Ebene (mit verschiedenen Tools) und Snapshottests. Leider sind nur die Unittests und Integrationstests auf Lambda-Ebene aktuell funktionsfähig. Bei diesen handelt es sich um Tests mit dem JS-Testframework Jest. Die Tests überprüfen, ob eine Todo richtig geupdatet wird. Sprich den Fall eines Put-Requests auf den Endpunkt `/Todo/{id}` mit einer entsprechenden Id. Auch gibt es Tests für Post auf ein neues Todo.

4.5 Frontend



Kapitel 5

Building

Da das Projekt für AWS ausgelegt ist, ist lokales Building nicht empfehlenswert. Was ist benötigt, um dieses Projekt zu nutzen?

Ein AWS Account mit CLI-Zugriff. Diesen kann man auf der Webseite von AWS erstellen. Jedoch gelangt man vorerst in den Root-Account. Es ist zu empfehlen, einen Nutzeraccount beim Service IAM zu erstellen. Über diesen besteht die Möglichkeit, Access-Token für die CLI zu kreieren. Mittels des Tokens kann über das *AWS CLI Tool* (download via npm) Zugriff auf AWS Ressourcen erstellt werden.

Da mit AWS CDK gearbeitet wird muss auch dieses als CLI Tool installiert werden. Das Kommando *cdk deploy* kann anschließend genutzt werden, um den Stack auf AWS zu deployen.

Das Frontend kann ohne ein eigenes Backend verwendet werden. Hierfür müssen die Environment-Variablen gesetzt werden. Diese sind: `VITE_API_URL` und `VITE_REDIRECT_URL`. Mit *npm run dev* wird eine lokale Instanz erstellt, welche mit dem Backend kommunizieren kann.



Kapitel 6

Fazit



Literatur

- [1] Amazon Web Services, *AWS CDK Constructs*, <https://docs.aws.amazon.com/cdk/v2/guide/constructs.html>, Accessed: 2026-01-01, 2026.