

# Kapitel 3

## Start des Entwicklungsprojekts

*Dieses Kapitel beschreibt die ersten Schritte auf dem Weg zu einer CAP-basierten Anwendung, die Sie am Ende auf die SAP Cloud Platform deployen werden. Ich stelle Ihnen das Modell eines Produktkatalogs vor und zeige Ihnen, wie Sie das Entwicklungsprojekt dazu aufsetzen. Erste Basisteile des Anwendungsmodells werden in CDS überführt.*

In Teil I dieses Buches haben Sie die Grundlagen über CAP gelernt und als kleine Übungsaufgabe einen einfachen »Hello World«-Service entwickelt. Nun möchte ich Ihnen anhand einer Beispielanwendung zeigen, wie sie einen Produktkatalog implementieren, in dem Benutzer\*innen nach Produkten suchen und Produkte bestellen können. In einer klassischen Dreischichten-Architektur werden Sie die Persistenzschicht und die Service-schicht mittels CAP sowie eine Benutzeroberfläche basierend auf SAP Fiori implementieren.

CAP macht es Ihnen leicht, ein Entwicklungsprojekt zu beginnen und dann im Grow-as-you-go-Stil schrittweise auszubauen. Vor dem ersten Kontakt mit CDS werden Sie in Abschnitt 3.1 dieses Kapitels ein Entitätenmodell und ein Verhaltensmodell der Anwendung entwerfen. In Abschnitt 3.2 legen Sie das eigentliche Entwicklungsprojekt an, indem Sie das Entitätenmodell in Abschnitt 3.3 direkt in ein CDS-Modell überführen. Nach dieser datenbezogenen Arbeit behandelt Abschnitt 3.4 die Serviceschicht der Anwendung, und in Abschnitt 3.5 führen Sie schon komplexe Datenabfragen durch, ohne ein Stück Programmcode geschrieben zu haben. In Abschnitt 3.6 zeige ich Ihnen, wie Sie CAP-Services mit einem HTTP-Client systematisch testen, bevor Sie in Abschnitt 3.7 zum ersten Mal Programmcodes beisteuern und debuggen.

Vom Modell zum Code

### 3.1 Modell eines Produktkatalogs

Bevor es an die Entwicklung selbst geht, sollten Sie ein Domänenmodell des Produktkatalogs erstellen, das alle Entitäten und Beziehungen abbildet. Abbildung 3.1 zeigt das Domänenmodell unseres Katalogs, mit den invol-

vierten Entitäten und ihren gegenseitigen Beziehungen mit den jeweiligen Kardinalitäten.

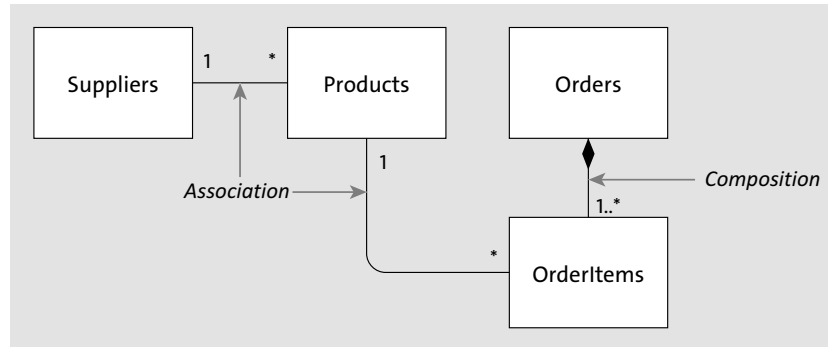


Abbildung 3.1 Domänenmodell des Produktkatalogs

**Domänenmodell** Jeder Kasten stellt eine Entität dar. Als wesentliche Entitäten neben dem eigentlichen Produkt (*Products*) gibt es den Lieferanten (*Suppliers*) und die Bestellung. Die Bestellung ist eine strukturierte Entität, bestehend aus einem Bestellkopf (*Orders*) und den Bestellpositionen (*OrderItems*). Die Entität *Orders* ist also eine *Komposition*. Zu einer Bestellung gehört mindestens eine Bestellposition. Außerdem können die Positionen nicht alleine, sondern nur in Bezug zu einem Bestellkopf existieren.

**Assoziation und Komposition** Eine Beziehung zwischen den Entitäten kann aber nicht nur eine *Komposition*, sondern auch eine *Assoziation* sein. Assoziierte Entitäten können unabhängig voneinander existieren. Beispielsweise existiert ein Lieferant unabhängig davon, ob eines seiner Produkte im Katalog enthalten ist. Zahlen repräsentieren Kardinalitäten von Beziehungen. Das Sternchen \* steht für eine beliebige positive Zahl oder Null. Ein Lieferant ist deswegen zu keinem, einem, oder mehreren Produkten assoziiert.

Das Diagramm des Domänenmodells zeigt nicht die innere Struktur der jeweiligen Entitäten, diese werden Sie aber gleich in dem korrespondierenden CDS-Modell spezifizieren.

**Rollen und Verhalten** Neben dem statischen Domänenmodell ist es natürlich ebenso wichtig, das Verhalten der Anwendung festzulegen. Wie in Abbildung 3.2 gezeigt, soll der Produktkatalog zwei Anwenderrollen unterstützen: den Administrator und den Katalognutzer. Beide unterscheiden sich durch die Daten, auf welche sie zugreifen und die Operationen, die sie mit diesen Daten ausführen dürfen.

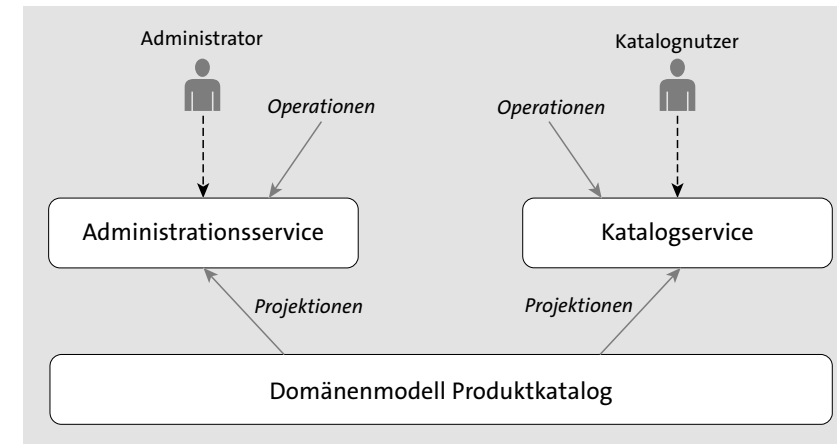


Abbildung 3.2 Anwenderrollen und zugeordnete Services

Jeder dieser Rollen wird ein CDS-Service zugeordnet, der die jeweils relevanten Sichten des Domänenmodells exponiert und mit den für die Rolle definierten Operationen ausgestattet ist. Die Datenstrukturen der Services sind dabei Projektionen aus dem gemeinsamen Domänenmodell.

Zuordnung von Rollen zu Services

#### Zielgerichtete CDS-Services

Erstellen Sie keinen »allmächtigen« CDS-Service, der alle Daten exponiert und der möglichst viele Operationen ausführen kann. Erstellen Sie stattdessen mehrere zielgerichtete Services, jeweils für eine Anwenderrolle oder einen Anwendungsfall (auch *Use Case* genannt). Diese Services exponieren gezielt nur relevanten Aspekte des Domänenmodells. So kann der Service auch mit rollengerechten Operationen ausgestattet werden. Das an Rollen ausgerichtete Autorisierungskonzept ist deshalb einfacher zu implementieren. CDS-Services können als Grundlage für Benutzeroberflächen dienen, die damit automatisch rollenspezifisch werden.

Eine Ausnahme von dieser Empfehlung gilt für Services, die zum Zweck der Stammdatenreplikation erstellt werden. Hier kann es von Vorteil sein, große und tiefe Strukturen zu exponieren, um viele Daten in wenigen Anfragen zu transportieren.

Für den Produktkatalog beschränken wir uns auf die in Tabelle 3.1 aufgeführten Serviceoperationen.

Anwenderrolle	Operation
Administrator	Produkte anlegen, lesen, ändern, löschen
	Lieferanten anlegen, lesen, ändern, löschen
	Lesen aller Bestellungen mit Suchkriterien
	Bestellungen lesen und löschen
Katalogbenutzer	Lesen (Auflisten) aller Produkte mit Suchkriterien
	Bestellungen anlegen
	eigene Bestellungen lesen

Tabelle 3.1 Serviceoperationen pro Anwenderrolle

## Sicherheit durch Design

Nach dem Prinzip der rollenspezifischen Services werden Sie einen Service pro Anwenderrolle definieren: einen Administratorservice und einen Katalogservice. Wie Sie in Tabelle 3.1 sehen, exponiert der Katalogservice beispielsweise keine Lieferanteninformationen, oder Operationen auf Lieferanten. Damit haben die Katalognutzer keine Möglichkeit, auf diese Entität zuzugreifen. Deswegen wird die Projektion aus dem Domänenmodell heraus für den Katalogservice von vorneherein keine Lieferanteninformation enthalten. Wie Sie in Abschnitt 5.4, »Authentifizierung und Autorisierung«, sehen werden, können Sie eine ähnliche Einschränkung auch mittels Berechtigungen erreichen. Wenn ein Service allerdings nur konsumentengerechte Informationen und Operationen anbietet, ist die Wahrscheinlichkeit wesentlich geringer, dass Sie durch Unachtsamkeit Sicherheitslücken verursachen.

## 3.2 Entwicklungsprojekt anlegen

Legen Sie auf Ihrem Dateisystem einen Projektordner mit dem Namen **prod-cat** an. Öffnen Sie diesen Order mit Visual Studio Code. Öffnen Sie außerdem wieder ein Terminal in Visual Studio Code und überprüfen Sie mit dem Kommando `pwd`, ob das Terminal Ihren Projektordner als aktuelles Arbeitsverzeichnis gesetzt hat.

## Generierung von Projektdateien

Führen Sie dann das Kommando `cds init` aus. Eine Reihe von Dateien und Ordnern wird in Ihrem Projektverzeichnis erzeugt. Ihr Projekt sollte danach die in Abbildung 3.3 gezeigte Struktur haben. Es handelt sich um eine von CAP vorgeschlagene Struktur, von der Sie zwar abweichen können, was aber

zusätzlichen Konfigurationsaufwand bedeutet. Wenn Sie keine guten Gründe haben, sollten Sie bei dieser vorgeschlagenen Struktur bleiben.

The screenshot shows the Visual Studio Code interface. The Explorer view on the left displays the project structure for 'PROD-CAT' with folders 'app', 'db', and 'srv', and files '.cdsrc.json', '.eslintrc', 'package.json', and 'README.md'. The main editor shows the 'package.json' file with the following content:

```

{
  "name": "prod-cat",
  "version": "1.0.0",
  "description": "A simple CAP project.",
  "repository": "<Add your repository here>",
  "license": "UNLICENSED",
  "private": true,
  "dependencies": {
    "@sap/cds": "^4",
    "express": "^4"
  },
  "devDependencies": {
    "sqlite3": "^4"
  },
  "scripts": {
    "start": "npx cds run"
  }
}

```

The Terminal view at the bottom shows the output of the `cds init` command:

```

>> cds init
[cds] - creating new project in current folder
done.

Find samples on https://github.com/SAP-samples/cloud-cap-samples
Learn about next steps at https://cap.cloud.sap/docs/get-started
>>


```

Abbildung 3.3 Empfohlene Projektstruktur mit dem Projektdeskriptor »package.json«

Jeder dieser Ordner hat seine und jede Datei ihre spezifische Bedeutung:

## Projektstruktur und Dateien

- Der Ordnername **app** steht für »Applikation«. Dort werden Artefakte abgelegt, die für die Benutzeroberfläche relevant sind. Initial ist dieser Ordner leer. Wenn Sie ein Projekt durchführen, das keine Benutzeroberfläche verlangt, können Sie diesen Ordner natürlich löschen.
- Der Ordnername **db** steht für »Datenbank«. Genauer gesagt soll er die Heimat für das Domänenmodell der Anwendung sein. Da dies aber eng an die Persistenz gekoppelt ist, entsteht der enge Bezug zur Datenbank. Initial ist dieser Ordner leer.

- Der Ordner `srv` wird Artefakte enthalten, welche die Serviceschicht der Anwendung implementieren. Initial ist auch dieser Ordner leer.
- Die Datei `.cdsrc.json` kann neben `package.json` für CAP-Konfigurationsdaten genutzt werden.
- Die Datei `.eslintrc` ist eine Konfigurationsdatei für das Werkzeug *ESLint* (siehe <https://eslint.org/>), das statische Codeprüfungen für JavaScript-Code durchführt. ESLint ist ein Quasi-Standard in der JavaScript-Entwicklung und wird von den meisten JavaScript-Editoren genutzt. Wenn Sie die Datei öffnen, sehen Sie, welche Konfigurationen CAP vornimmt. Bestimmte globale Variablen wie `SELECT` werden z. B. bekannt gemacht, um gewisse Warnungen zu unterdrücken.
- Die Datei `package.json` ist der zentrale Projektdeskriptor für npm-Projekte (siehe <http://s-prs.de/v765105>). CAP nutzt diese Datei auch für seine spezifischen Konfigurationen. Der wichtigste Eintrag ist im Moment der `dependencies`-Abschnitt, der festlegt, welche npm-Pakete in welcher Version im Projekt installiert werden sollen. Sie sehen hier das `cds`-Paket von SAP mit der Major-Version 4 sowie das `express`-Paket (siehe <https://expressjs.com>). Letzteres beinhaltet das Web-Framework *Express*, das CAP für Node.js nutzt, um die HTTP-Endpunkte der Services bereitzustellen.
- Die technische Dokumentation zu Ihrem Projekt legen Sie in der Datei `README.md` ab. Sie nutzen dabei das *Markdown*-Format (siehe <http://s-prs.de/v765106>). Visual Studio Code ermöglicht eine parallele Ausgabeansicht, wenn Sie eine Markdown-Datei geöffnet haben. Klicken Sie dazu auf die Datei `README.md` im Dateieexplorer und danach auf das Symbol  im rechten oberen Bereich von Visual Studio Code. Ein zweites Editor-Fenster öffnet sich mit der Ausgabeansicht der Markdown-Datei.



#### Sichere CAP-Anwendungen mit dem Express-Web-Framework

CAP für Node.js nutzt das quelloffene Web-Framework *Express*, um HTTP-Endpunkte in einem Netzwerk (z. B. dem Internet) zu exponieren. Damit sind auch CAP-basierte Anwendungen potenziell das Ziel bössartiger Attacken, die über das Internet erfolgen. Beachten Sie deshalb bei der Erstellung produktiver Anwendungen die Empfehlungen zur sicheren Nutzung von *Express* unter <http://s-prs.de/v765107>. Zur Umsetzung dieser Empfehlungen helfen Ihnen die Informationen zur Sicherung von CAP-basierten Anwendungen auf <http://s-prs.de/v765108>.

Konfiguration für Visual Studio Code

Neben diesen direkt sichtbaren Dateien hat `cds init` noch den Ordner `.vscode/` erzeugt. Dort werden die Einstellungen Ihrer Visual-Studio-Code-

IDE projektspezifisch abgelegt. Dieser Ordner ist im Dateieexplorer nicht sichtbar, da es nicht notwendig ist, die darin enthaltenen Dateien mit einem gewöhnlichen Editor zu bearbeiten. Visual Studio Code bietet dazu den Menüeintrag **File • Preferences • Settings**.

Damit Ihr Projekt vollständig und selbstkonsistent wird, sollten Sie nun alle in der Datei `package.json` definierten Abhängigkeiten installieren. Damit vermeiden Sie Abhängigkeiten von eventuell global auf Ihrem Rechner installierten Paketen. Ihr Projekt können Sie so leichter mit anderen austauschen, z. B. über ein Versionskontrollsystem. Bevor Sie dies durchführen, sollten Sie für diese Entwicklungsphase noch eine npm-Konfiguration vornehmen, um zu verhindern, dass die Installation eine Datei `package-lock.json` generiert, die Versionen von Abhängigkeiten für ein produktives Deployment festschreibt. Geben Sie dazu im Terminal das Kommando `npm set package-lock=false` ein. Sie können die gesamte npm-Konfiguration jederzeit durch das Kommando `npm config ls` einsehen.

Für die Installation der Abhängigkeiten stellt npm das Kommando `npm install` bereit. Nach dessen Ausführung sehen Sie im Dateieexplorer einen neuen Ordner mit dem Namen `node_modules/`, der die angeforderten Abhängigkeiten (und wiederum deren Abhängigkeiten) enthält. Wenn Sie die Deklaration der Abhängigkeiten in der Datei `package.json` ändern, sollten Sie anschließend wieder `npm install` ausführen.

Nach diesen Vorbereitungen steigen Sie nun in die eigentliche Entwicklungsarbeit ein.

#### Download der Dateien zum Beispielprojekt

In diesem Buch finden Sie eine Reihe von Listings, die den Inhalt der Dateien zeigen, die Sie für Ihr Anwendungsprojekt benötigen. Mit der Zeit werden diese Inhalte natürlich komplexer, sodass eine manuelle Übertragung in Ihr Projekt zeitaufwendig, aber vor allem fehleranfällig wird. Deswegen werden die Projektdateien auf einem Server des Rheinwerk Verlags für Sie zum Download bereitgestellt.

Sie erreichen den Server unter <https://www.rheinwerk-verlag.de/5110>. Laden Sie dort das unter dem Reiter **Materialien** hinterlegte ZIP-Archiv herunter, und extrahieren Sie es in einem Ordner Ihrer Wahl auf Ihrem Rechner. Der Inhalt gliedert sich in mehrere weitere ZIP-Archive, in denen Codebeispiele abgelegt sind. Die ZIP-Dateien sind nach den Kapiteln bzw. den Abschnitten benannt, zu denen die jeweilige Ausbaustufe des Projekts gehört. Die Datei `Kapitel_03.zip` enthält z. B. den Stand des Projekts am Ende von

Abhängigkeiten installieren

Das Verzeichnis »node\_modules/«



Kapitel 3. Das Archiv **Kapitel\_05.02.zip** enthält den Stand des Projekts am Ende von Abschnitt 5.2.

Der Ordner **node\_modules/** ist nicht Bestandteil der ZIP-Archive. Dies hat den Zweck, dass Sie immer möglichst aktuelle Versionen der Projektabhängigkeiten nutzen. Wenn Sie ein Vorlageprojekt kopieren, denken Sie deswegen daran, danach immer `npm install` auszuführen, um alle Projektabhängigkeiten, die in dem Projektdeskriptor **package.json** deklariert sind, im Ordner **node\_modules/** zu installieren.

Seien Sie aber nicht zu kopierfreudig. Machen Sie sich zumindest zu Beginn die Mühe und übertragen Sie die Listings manuell. Sie lernen dadurch den Umgang mit der IDE und vor allem die Fähigkeiten der CDS-Sprachunterstützung im Editor kennen. Des Weiteren lernen und verinnerlichen Sie nur so die Sprachsyntax von CDS.

### 3.3 Domänenmodell anlegen

Nachdem Sie das Entwicklungsprojekt für Ihren Produktkatalog angelegt haben, sollten Sie als Nächstes das Domänenmodell anlegen.

Domänenmodell  
in CDS

Das Domänenmodell des Produktkatalogs, das ich zu Beginn des Kapitels in Abbildung 3.1 vorgestellt habe, lässt sich direkt in ein CDS-Modell übersetzen. Legen Sie dazu die Datei **schema.cds** im Ordner **db/** an, mit dem Inhalt, den Listing 3.1 zeigt.

```
namespace my.domain;
entity Products {
  key ID      : Integer;
  name       : String(100) not null;
  stock      : Integer;
  price      : Decimal(9, 2);
  retail     : Decimal(9, 2);
  currency_code: String(3);
  supplier   : Association to Suppliers;
}
entity Suppliers {
  key ID      : Integer;
  name       : String(100);
  priority   : Integer;
  products  : Association to many Products on products.supplier =
$self;
}
```

```
entity Orders {
  key ID      : UUID;
  orderNo    : String;
  currency_code: String(3);
  Items      : Composition of many OrderItems on Items.parent =
$self;
}
entity OrderItems {
  key ID      : UUID;
  parent     : Association to Orders;
  product    : Association to Products;
  amount     : Integer;
}
```

Listing 3.1 Basis-Domänenmodell des Produktkatalogs

Falls Sie sich wundern: Es ist eine Konvention, Pluralnamen für die Entitäten zu verwenden (`Products` anstelle von `Product`). Neben den Entitäten und deren Elementen werden auch die Assoziationen und Kompositionen direkt in CDS ausgedrückt. Für die Rückbeziehungen, z. B. von `Suppliers` nach `Products`, wird ein Join-Kriterium angegeben.

Die Festlegung eines Namensraums mit der `namespace`-Deklaration gilt für alle Entitäten in der CDS-Datei. Die Entität `Products` besitzt damit in einem globalen Kontext (über die einzelne Datei hinaus) den voll qualifizierten Namen `my.domain.Products`.

Namensräume

Speichern Sie das Domänenmodell und stellen Sie sicher, dass der CDS-Editor keine Syntaxfehler anzeigt. Im Verlaufe dieses Buches werden Sie dieses Domänenmodell weiter verfeinern. Die Bedeutung der einzelnen Elemente der Entitäten ist in Tabelle 3.2 erläutert.

Entität	Element	Bedeutung
Products	ID	technisches Schlüsselfeld
	name	Produktname für eine Benutzeroberfläche. Der Zusatz <code>not null</code> macht dieses Element obligatorisch.
	stock	Bestand eines Produkts
	price	Einkaufspreis mit 9 Dezimalen, davon 2 Nachkommastellen

Tabelle 3.2 Semantik der Elemente der Entitäten des Domänenmodells

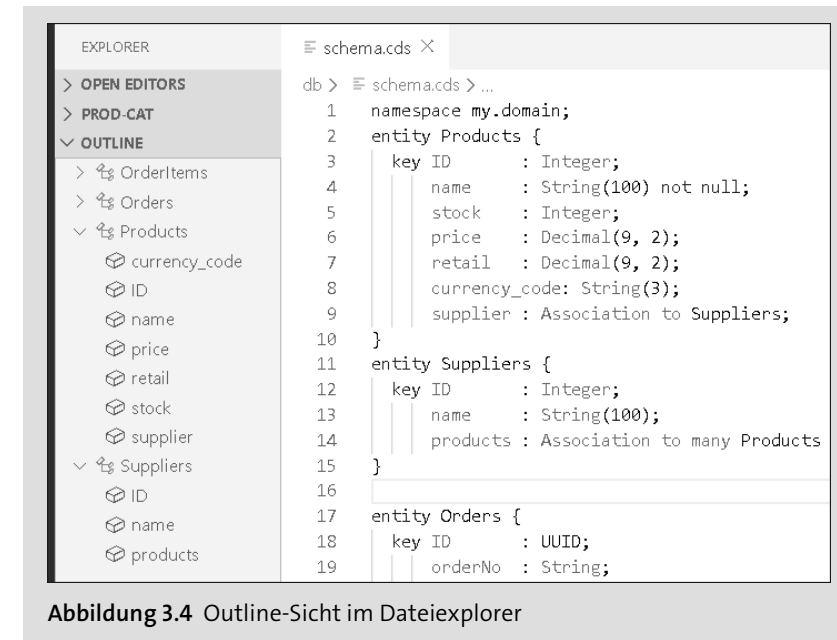
Entität	Element	Bedeutung
Products (Forts.)	retail	Verkaufspreis mit 9 Dezimalen, davon 2 Nachkommastellen
	currency_code	Währungscode von drei Zeichen Länge
	supplier	Assoziation zu genau einem Lieferanten
Suppliers	ID	technisches Schlüsselfeld
	name	Lieferantenname für die Benutzeroberfläche
	priority	Priorität des Lieferanten
	products	Assoziation zu mehreren Produkten
Orders	ID	Schlüsselfeld vom Typ <i>Universially Unique Identifier</i> (UUID). Die Werte solcher Schlüsselfelder werden von CAP automatisch generiert.
	orderNo	Bestellnummer zur Anzeige an der Benutzeroberfläche
	currency_code	Währungscode für den Gesamtwert
	Items	Deklaration für die Komposition aus OrderItems
OrderItems	ID	Schlüsselfeld vom Typ UUID
	parent	Zeiger auf die Kopfentität Orders
	product	Assoziation zu genau einem Produkt
	amount	Bestellmenge in Stück

**Tabelle 3.2** Semantik der Elemente der Entitäten des Domänenmodells (Forts.)



#### Die Outline-Sicht für CDS in Visual Studio Code

Die CDS-Sprachunterstützung für Visual Studio Code erzeugt eine sogenannte *Outline-Sicht*. Sie sehen die Outline-Sicht im Datei-Explorer unterhalb der Ordnerstruktur. Wenn Sie eine CDS-Datei im Editor geöffnet haben, zeigt Ihnen die Outline-Sicht eine hierarchische Übersicht der Entitäten und Services mit ihren untergeordneten Elementen (siehe Abbildung 3.4).



**Abbildung 3.4** Outline-Sicht im Dateixplorer

### 3.4 Die Serviceschicht

Im Sinne von "Grow as you go" sollen Sie schnelle, testbare Fortschritte mit Ihrem Projekt erzielen. Deswegen definieren Sie nun den Service für den Administrator mit den in Abschnitt 3.1 beschriebenen Eigenschaften. Legen Sie dazu im Ordner `srv/` die Datei `admin-service.cds` mit dem in Listing 3.2 gezeigten Inhalt an.

```
using my.domain from '../db/schema';
service AdminService {
    entity Products as projection on domain.Products;
    entity Suppliers as projection on domain.Suppliers;
    entity Orders as projection on domain.Orders;
}
```

**Listing 3.2** Definition des Administratorservices

Mit der ersten Zeile wird der Namensraum des Domänenmodells importiert. Die Pfadangabe ist dabei eine relative Referenz auf die Datei `schema.cds` im Ordner `db/`. Im Nachfolgenden können dadurch alle Entitäten des Domänenmodells referenziert werden. Dazu muss nur das letzte

Der Administrator-service

Import des Domänenmodells

Segment des Namensraums der Entität vorangestellt werden, wie z. B. in `domain.Products`. Die Definition des `Administratorservices` besagt, dass die drei gelisteten Entitäten zu exponieren sind, die einfach als 1:1-Projektionen der jeweiligen Entitäten aus dem Domänenmodell anzubieten sind.



#### Die »using«-Direktive

In Ihrem Entwicklungsprojekt nutzen Sie mehrere Dateien, um Ihr CDS-Modell zu definieren. Diese Aufspaltung dient zum einen einer semantischen Strukturierung des Projekts (z. B. Trennung von Domänen- und Servicemodellen). Zum anderen kann die Trennung aber auch, im Falle der Wiederverwendung von Modellen, die in anderen Projekten erstellt wurden, notwendig sein. Die `using`-Direktive sorgt dafür, dass die getrennten Modellartefakte zur Laufzeit zu einem konsistenten Gesamtmodell zusammengeführt werden. Je mehr Sie das Gesamtmodell fragmentieren, desto mehr müssen Sie Fragmente zusammenbinden. Untergliedern Sie Ihr Projekt deswegen nicht zu stark. Nutzen Sie die Semantik von Modellen und auch Verantwortlichkeiten, um eine maßvolle Strukturierung vorzunehmen. Die `using`-Direktive wird im Detail in Abschnitt 4.8, »Referenzen auf Modelle«, vorgestellt.

#### Testen des Services

Wie Sie in Kapitel 2, »Erste Schritte zur eigenen Anwendung«, gesehen haben, ist es jetzt sehr einfach, den Service zu testen. Starten Sie dazu in Ihrem Terminal `cds watch` und danach Ihren Browser über den angebotenen Link auf `http://localhost:4004/`. Auf der Willkommenseite sehen Sie die Struktur des `Administratorservices` wie in Abbildung 3.5 gezeigt.

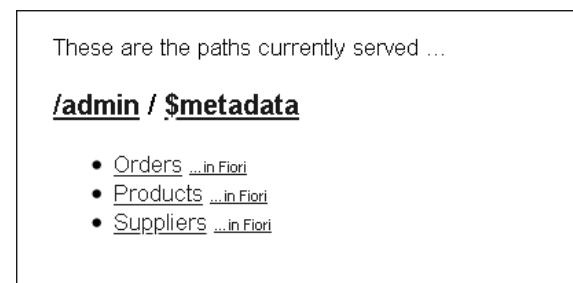


Abbildung 3.5 Browserseite mit der Spezifikation des `Administratorservices`

#### OData-Metadaten

Klicken Sie auf den Pfad `/admin`, um das sogenannte *OData-Servicedokument* im JSON-Format zu sehen. Unter dem Pfad `/admin/$metadata` erreichen Sie das *OData-Metadatenokument* des Services. Dies ist ein XML-Dokument, das die detaillierte Struktur und gegebenenfalls auch die Opera-

tionen eines Services beschreibt. Beide OData-Dokumente werden aus den CDS-Modellen generiert und sind für Sie deshalb nur abgeleitete Artefakte, die als technisches Kommunikationsprotokoll genutzt werden. Es besteht aktuell kein Bedarf für Sie, zusätzlich zu CDS den OData-Standard im Detail kennenlernen zu müssen.

#### Ableitung von Service-Endpunkten

Der Pfad eines Service-Endpunkts wird nach den folgenden Regeln bestimmt:

- Der Servicenamen im CDS-Modell wird in Kleinbuchstaben überführt. In den erzeugten Namen werden Bindestriche zur Trennung eingesetzt, wenn im Servicenamen ein Großbuchstabe vor einem Kleinbuchstaben steht. »Camel-Case-Notation« (`»camelCase«`) wird also in »Kebab-Case-Notation« (`»kebab-case«`) überführt.
- Endet der Servicenamen mit »Service«, wird dieser Teil bei der Bildung des URL-Pfads für den Endpunkt ignoriert.
- Durch die Angabe der Annotation `@path` kann ein beliebiger Pfad festgelegt werden, unabhängig vom Servicenamen. Das Konzept der Annotationen wird in Kapitel 4, »Core Data Services für CAP im Detail«, erläutert.

Wie in Abbildung 3.5 gezeigt, werden Ihnen auch Hyperlinks zu den durch den Service exponierten Entitäten angeboten. Klicken Sie z. B. auf **Products**, um die vorhandenen Produktdaten innerhalb des JSON-Arrays `value` zu sehen. Da noch keine Daten vorhanden sind, ist dieses Array leer. Wie Sie in Kapitel 2, »Erste Schritte zur eigenen Anwendung«, gelernt haben, können Sie einfach mit einer `.csv`-Datei einige Testdaten erzeugen. Legen Sie dazu die Datei `my.domain-Products.csv` in einem neuen Ordner `data/` unterhalb des `db/`-Ordners an. Beachten Sie, dass der Namensraum Teil des `.csv`-Dateinamens sein muss, um eine Zuordnung zur Entität zu ermöglichen. Erstellen Sie einen beliebigen Testinhalt an Produktdatensätzen, z. B. wie in Listing 3.3 gezeigt.

```
ID;name;stock;price;retail;currency_code;supplier_ID
101;Flat Screen TV 32 inch;2;199.98;219.00;EUR;301
102;MyVoice Fitness Headphone blue;207;5.75;11.99;EUR;303
```

Listing 3.3 Inhalt einer »csv«-Datei mit zwei Produkttestdatensätzen

Beachten Sie das Feld `supplier_ID`. Da das Produkt eine Assoziation auf die Lieferantenentität besitzt, wird die Produkttabelle ein Fremdschlüsselfeld



#### Testdaten einfügen

für den Lieferanten enthalten. Vermeiden Sie das Einfügen von Leerzeichen zwischen den Semikola, denn dies kann bei der Übertragung in eine Datenbank zu Problemen führen. Sobald Sie die `.csv`-Datei speichern, wird der laufende `cds watch`-Prozess diese Änderung detektieren und die Testdaten in die In-Memory-Datenbank SQLite deployen. Durch erneutes Klicken auf den **Products**-Link sollten die Testdaten nun sichtbar sein.

### 3.5 Datenabfragen

#### OData-Query-Optionen

Mit den bisher implementierten CDS-Modellen lassen sich über Ihren Browser schon komplexe Datenabfragen (*Queries*) an den Administratorservice senden. Da dieser Service eine OData-Service ist, unterstützt er automatisch die meisten der in der OData-Spezifikation beschriebenen *Query-Optionen*. Diese werden als URL-Parameter übergeben und beginnen alle mit dem `$`-Zeichen. Details dazu, welche Optionen OData spezifiziert, finden Sie unter <http://s-prs.de/v765109>. Suchen Sie dort im Inhaltsverzeichnis nach dem Kapitel »Query Options«. CAP unterstützt die neueste Version 4 der OData-Spezifikation.

#### Serviceantwort im Browser

Die Antwortseite eines Browsers auf eine Datenabfrage ist beispielhaft in Abbildung 3.6 gezeigt. Die Ausgabe ist als strukturiertes JSON-Dokument gezeigt. Je nachdem, welchen Browser mit welchen Erweiterungen Sie verwenden, kann die Ausgabe im Detail etwas vom gezeigten Beispiel abweichen.



#### Vollständige Testdaten

Damit Sie beim Testen der Query-Optionen aussagekräftige Resultate sehen, empfiehlt es sich für alle Entitäten, `.csv`-Dateien mit mehreren Einträgen bereitzustellen. Wenn Sie solche `.csv`-Dateien nicht selbst erstellen möchten, können Sie auf die Dateien zurückgreifen, die über die Downloadseite (zu finden unter <https://www.rheinwerk-verlag.de/5110>) zu diesem Buch zu beziehen sind. Für dieses Kapitel finden Sie passende `.csv`-Dateien im Archiv `Kapitel_03.zip` im Ordner `db/data/`.

#### Nützliche OData-Queryys

Im Rahmen dieses Buches kann ich keine vollständige Übersicht über alle OData-URL-Parameter geben. Tabelle 3.3 zeigt Ihnen aber eine Sammlung nützlicher und häufig genutzter Query-Optionen anhand des Administratorservices.

```

{
  "@odata.context": "$metadata#Orders(orderNo, ID, Items())",
  "@odata.metadataEtag": "W/\ "/QsriG98dJva5e04xpcHrmPUL6xi9LKxlR2p8/5abPA=\ ",
  "value": [
    {
      "ID": "4c1ea77c-ce95-439e-92d6-6f14cd35797b",
      "orderNo": "MX50001",
      "Items": [
        {
          "ID": "0caa520e-6758-4513-887e-090036d8bc99",
          "amount": 1,
          "parent_ID": "4c1ea77c-ce95-439e-92d6-6f14cd35797b",
          "product_ID": 103
        },
        {
          "ID": "cd5900d9-b1f6-43ec-a543-a0760679fc31",
          "amount": 2,
          "parent_ID": "4c1ea77c-ce95-439e-92d6-6f14cd35797b",
          "product_ID": 101
        }
      ]
    },
    {
      "ID": "377404df-4951-4cb3-afe9-ae2bfa1db5e",
      "orderNo": "ZZ50003",
      "Items": [
        {
          "ID": "9e1f5362-a76a-48fb-b23d-e46c17603cc4",
          "amount": 2,
          "parent_ID": "377404df-4951-4cb3-afe9-ae2bfa1db5e",
          "product_ID": 121
        }
      ]
    }
  ]
}

```

Abbildung 3.6 Ausgabe im Browser für die Datenabfrage »/admin/Orders?\$select=orderNo&\$expand=Items«

Der dort aufgeführten URL mit Parametern müssen Sie jeweils noch den Teil `http://localhost:4004/admin` voranstellen, um die vollständige Query-URL für den Administratorservice zu erhalten, die Sie in Ihrem Browser eingeben. Probieren Sie selbst neue Querys aus, indem Sie verschiedene Optionen kombinieren. Denken Sie daran, dass der Prozess `cds watch` laufen muss, damit der Administratorservice für den Browser verfügbar ist.



URL mit Parametern	Rückgabemenge
<code>/Products(103)</code>	alle Elemente des Produkts mit ID = 103
<code>/Suppliers(303)/products</code>	alle Produkte, die dem Lieferanten mit ID = 303 zugeordnet sind
<code>/Products(101)/supplier</code>	der Lieferant, der dem Produkt mit ID = 101 zugeordnet ist
<code>/Products?\$select=name,price</code>	die Elemente ID, name, price für alle Produkte
<code>/Products?\$search=Phone</code>	Alle Produkte, die die Zeichenkette »Phone« in Elementen vom Typ String beinhalten. Groß- und Kleinbuchstaben werden nicht unterschieden.
<code>/Suppliers?\$expand=products</code>	Alle Lieferanten. Pro Lieferant alle zugeordneten Produkte
<code>/Suppliers?\$expand=products(\$select=name,price)</code>	Alle Lieferanten. Pro Lieferant alle zugeordneten Produkte, aber nur die Elemente ID, name, price
<code>/Orders?\$expand=Items</code>	alle Bestellungen mit ihren zugehörigen Bestellpositionen
<code>/Suppliers/\$count</code>	Anzahl der Lieferanten
<code>/Suppliers?\$count=true</code>	alle Lieferanten sowie das zusätzliche OData-Metadatum @odata.count
<code>/Products?\$orderby=stock</code>	alle Produkte, aufsteigend sortiert nach ihrem Bestand
<code>/Products?\$orderby=stock&amp;\$search=Phone</code>	alle Produkte, die dem Suchkriterium »Phone« genügen, sortiert nach ihrem Bestand
<code>/Products?\$filter=stock lt 40</code>	alle Produkte, deren Bestand kleiner als 40 ist
<code>/Products?\$skip=1&amp;\$top=2</code>	das zweite und dritte Produkt (überspringe das erste und zeige die nächsten zwei)

Tabelle 3.3 Nützliche Query-Optionen für den Administratorservice

### 3.6 Testen mit einem HTTP-Client

Die Testseite, die unter `http://localhost:4004/` angeboten wird, bietet Ihnen schon einige Möglichkeiten, die Korrektheit Ihrer Domänen- und Service-Modelle zu testen. Insbesondere lassen sich Datenabfragen senden und

Service-Operation testen. Allerdings lassen sich auf diese Art und Weise nur Anfragen an den Service senden, die vorhandene Daten nach gewissen Kriterien selektieren oder berechnen und dann an den Browser zurückschicken. Es ist nicht möglich, aktiv neue Daten zu erzeugen oder vorhandene Daten zu ändern. Die Daten, die Sie bisher in die Datenbank geschrieben haben, stammen aus `.csv`-Dateien, die das CDS Development Kit zu Testzwecken in die Datenbank importiert hat. In einer produktiven Umgebung werden solche neuen Daten oder Datenänderungen auch über eine Servicekommunikation des HTTP-Protokolls übermittelt werden müssen.

#### 3.6.1 Kommunikation über HTTP

Das *HTTP-Protokoll* definiert eine Reihe von Anfragemethoden (*Request-Methoden*), die mögliche Aktionen bei HTTP-Anfragen beschreiben. Wenn Sie eine URL in die Adresszeile eines Webbrowsers eintippen, sendet dieser z. B. eine GET-Anfrage an einen Server, der dann mit den angefragten Daten antwortet.

Der grundsätzliche Ablauf der Kommunikation ist in Abbildung 3.7 dargestellt. Ein *HTTP-Client* (z. B. Ihr Webbrowser) sendet eine Anfrage ❶ an einen HTTP-Server. Jede Anfrage (Request) besteht aus einer Kopfinformation (*Request Header*) und den eigentlichen Daten (*Request Body*). Der Request Header beschreibt allgemeine Eigenschaften der Anfrage (z. B. bezüglich der Autorisierung) und enthält auch die in Abschnitt 3.5, »Datenabfragen«, benutzten URL-Parameter. Der Request Body kann z. B. Daten enthalten, die vom Server in einer Datenbank geändert werden sollen. Jedem Request ist eine Request-Methode zugeordnet, die beschreibt, was mit den Daten im Request Body geschehen soll. Die gängigsten Methoden sind GET, POST, PUT und PATCH, die auch für CAP eine wichtige Rolle spielen. Eine Beschreibung dieser Methoden zusammen mit der Beziehung zu CDS-Ereignissen finden Sie in Tabelle 3.4.

Nach Erhalt der Anfrage wird der Server diese verarbeiten ❷. Anschließend sendet der Server eine Antwort, die wiederum in einen Kopfteil (*Response Header*) und einen Datenteil (*Response Body*) gegliedert ist ❸. Die Antwort enthält mindestens eine Statusinformation in Form eines standardisierten Statuscodes, um dem Client mitzuteilen, ob seine Anfrage erfolgreich verarbeitet wurde, oder ob Fehler aufgetreten sind. Der Client hat dann die Möglichkeit, entsprechend zu reagieren ❹.

Ablauf einer HTTP-Kommunikation

Antwort des Servers

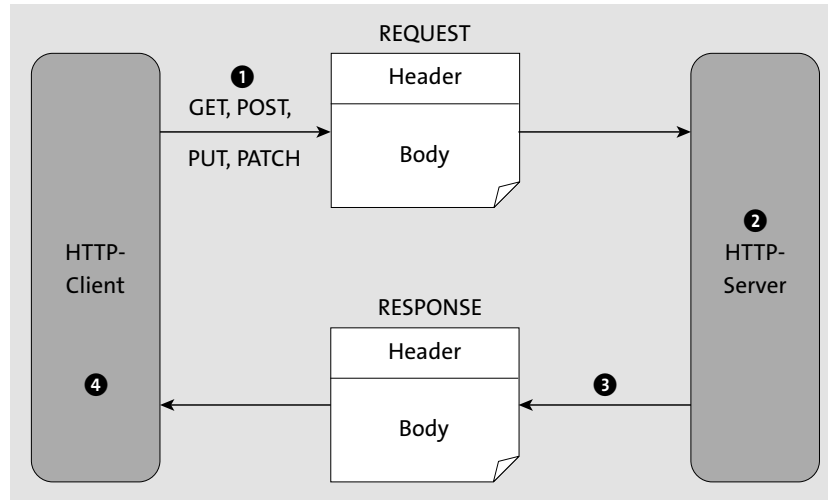


Abbildung 3.7 Ablauf der Kommunikation über das HTTP-Protokoll

Request-Methode	Bedeutung	CDS-Event
GET	Abfrage von Daten nach Kriterien, die als URL-Parameter, oder im Request Body übergeben werden	READ
POST	Erzeugung einer neuen Ressource	CREATE
PUT	Modifikation einer existierenden Ressource, oder Erzeugung einer neuen, wenn die Ressource noch nicht existiert	UPDATE
PATCH	teilweise Modifikation einer existierenden Ressource	UPDATE
DELETE	Löschen einer Ressource	DELETE

Tabelle 3.4 Die gängigsten HTTP-Methoden mit ihrer Bedeutung für CDS

### 3.6.2 Nutzung eines HTTP-Clients

In Gestalt Ihres Webbrowsers haben Sie ein einfaches Beispiel für einen HTTP-Client. Eine Einschränkung ist allerdings, dass der Browser selbst nur GET-Requests an einen Server schickt. Wenn Sie aber in Ihren Testszenarios Daten erzeugen, ändern, oder löschen wollen, so benötigen Sie einen HTTP-Client, der auch POST-, PUT-, oder DELETE-Requests schicken kann. Es gibt neben kommerziellen Angeboten eine Vielzahl freier HTTP-Clients in der Open-Source-Gemeinde, die Sie dafür nutzen können. Alle funktionieren

sehr ähnlich und werden geeignet sein, die hier gezeigten Beispiele nachzuvollziehen.

Für die abgebildeten Tests nutze ich eine Visual-Studio-Code-Erweiterung, weil diese sich optimal in die IDE integriert. Diese Erweiterung können Sie in Visual Studio Code installieren, indem sie die Webseite mit der Adresse <http://s-prs.de/v765110> besuchen. Die Installation kann auch direkt über die Extension-Sicht von Visual Studio Code initiiert werden, wenn Sie dort nach dem Begriff »REST Client« suchen.

Am einfachsten nutzen Sie die Erweiterung, indem Sie in Ihrem Projekt einen Unterordner `_requests/` erzeugen und dort Dateien mit der Endung `.http` anlegen. Ich benutze für Hilfsverzeichnisse, die keine eigentlichen Anwendungsartefakte enthalten, die Konvention des Unterstrichs als Präfix. Jede dieser Dateien kann viele Request-Daten beinhalten, der Request kann direkt aus dem Editor gestartet werden und Sie können die Antwort in einem parallelen Fenster analysieren. Sie können verschiedene `.http`-Dateien anlegen, um z. B. alle GET-Requests und alle POST-Requests jeweils in einer Datei zu bündeln. Andere für Sie sinnvolle Aufteilungen sind aber auch möglich. Eine Reihe von Request-Dateien finden Sie in den Downloadmaterialien zu diesem Buch.

Abbildung 3.8 zeigt die HTTP-Client-Extension für Visual Studio Code in Aktion. Die Datei `GET.http` ist im linken Editorfenster geöffnet. Sie enthält eine Reihe von GET-Request-Definitionen. Die Requests werden durch eine Kommentarzeile voneinander getrennt, die mit mindestens drei aufeinanderfolgenden Raute-Zeichen (`#`) beginnt. Einen Request können Sie abschicken, indem Sie die Schaltfläche `Send Request` anklicken, die der Editor automatisch zu jedem Request erzeugt (siehe Markierung in Abbildung 3.8). Beachten Sie wieder, dass der Request natürlich nur beantwortet werden kann, wenn Sie Ihren Webserver zuvor gestartet haben, z. B. mit `cds watch`.

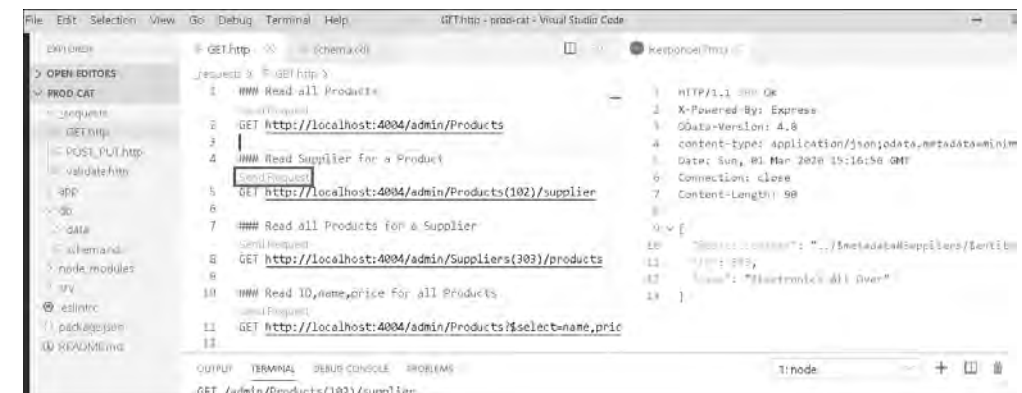


Abbildung 3.8 HTTP-Client Extension für Visual Studio Code

HTTP-Client als Visual-Studio-Code-Extension

Aufbau von Request-Dateien

Anfragen abschicken

**Antwort auf einen HTTP-Request**

Die Antwort auf einen Request wird in dem sich selbstständig öffnenden Response-Fenster zur Rechten angezeigt. Die erste wichtige Information im Response Header ist der HTTP-Statuscode mit der zugehörigen Meldung (**200 OK** in Abbildung 3.8). In diesem Fall wurde der Request erfolgreich bearbeitet. Der Response Header zeigt noch weitere beschreibende Informationen an, z. B., dass der Response Body im JSON-Format vorliegt. Der Response Body selbst entspricht für diesen GET-Request der Ausgabe, die Sie auch in Ihrem Browserfenster gesehen haben.

Mit dem HTTP-Client haben Sie nun ein mächtiges Testtool zur Verfügung, mit dessen Hilfe Sie im Folgenden auch Datenänderungen durchführen können. Die Requests, die in diesem Buch beispielhaft behandelt werden, finden Sie auch als `.http`-Dateien im Ordner `_requests/` in den Download-materialien zu diesem Buch.

**3.6.3 Datenänderungen über einen CDS-Service**

Bisher konnten Sie über Ihren Browser nur GET-Requests an den Administratorservice senden. Mithilfe des HTTP-Clients können nun auch die anderen HTTP-Request-Methoden POST, PUT, PATCH und DELETE verwendet werden.

**Anlegen neuer Datensätze**

Benutzen Sie einen Request wie in Listing 3.4, um einen neuen Lieferanten anzulegen. Neu ist, dass Sie nun einen Request Body mitschicken, in dem der neue Datensatz aufgeführt ist. Das Request-Header-Attribut `Content-Type` gibt an, dass diese Daten im JSON-Format geschickt werden. Beachten Sie die Leerzeile zwischen Header und Body.

```
### Create new Supplier
POST http://localhost:4004/admin/Suppliers
Content-Type: application/json

{
  "ID": 401,
  "name": "ACME Company"
}
```

**Listing 3.4** Request zur Anlage eines neuen Lieferanten

**JSON-Format in Service-Requests**

CAP-Services nutzen meist das JSON-Format, um Anfragen zu formatieren. Es wird wahrscheinlich auch bei Ihnen vorkommen, dass die Serverantwort manchmal einen Status `400 Bad Request` und eine Fehlermeldung wie

"Error while deserializing payload. An error occurred during deserialization of the entity. Unexpected string in JSON at position 23." beinhaltet. Der Grund ist, dass das JSON-Dokument im Ihrem Request Body nicht korrekt formatiert ist. Beachten Sie in dem Fall die Fehlermarkierungen in Ihrem Editor. Eine Einführung in das JSON-Format finden Sie beispielsweise unter <https://www.json.org/json-de.html>.

Nach dem Abschicken des Requests werden Sie als Antwort eine Erfolgsmeldung sehen (Status **201 Created**). Im Response Body erhalten Sie eine Kopie der neu angelegten Daten.

Schicken Sie den identischen Request nun erneut ab. Diesmal sollten Sie eine Antwort sehen, die auf einen Fehler hinweist (Status **400 Bad Request**). Im Response Body wird eine Fehlermeldung zurückgegeben. Dieses Verhalten entspricht der Semantik eines POST-Requests, der spezifiziert ist, neue Daten anzulegen. Vorhandene Daten können damit nicht geändert werden.

**Zurücksetzen Ihrer Daten in den Anfangszustand**

Nachdem Sie mehrere HTTP-Requests abgeschickt haben, können Sie bestimmte Tests eventuell nicht mehr erneut ausführen, weil Daten schon existieren oder nicht mehr konsistent sind. Wenn Sie mit der In-Memory-Option von SQLite arbeiten, ist ein Zurücksetzen sehr einfach: Stoppen Sie den aktuellen Serverprozess, indem Sie im Terminal `Strg` + `C` drücken und starten Sie `cds watch` anschließend wieder. Die In-Memory-Datenbank wird dadurch abgebaut und anschließend erneut aus den vorhandenen `.csv`-Dateien bestückt. Damit erhalten Sie wieder einen wohldefinierten Anfangszustand. Nutzen Sie also die `.csv`-Dateien, um diesen Anfangszustand zu definieren, und nutzen Sie die `.http`-Dateien für darauf basierende Tests. Wenn Sie nicht die In-Memory-Variante von SQLite nutzen, sondern eine über `cds deploy --to sqlite` erstellte, persistente Datenbankdatei (siehe Abschnitt 2.3.5, »Blick in die Datenbank«), so müssen Sie diese Datenbankdatei erneut mit `cds deploy --to sqlite` initialisieren, um den wohldefinierten Anfangszustand wiederherzustellen.

Mit einem POST-Request können auch tief strukturierte Daten erzeugt werden. In Ihrem Modell ist die Bestellung eine Komposition aus Bestellpositionen. Es ist deshalb sinnvoll, beide Entitäten immer gemeinsam anzulegen, denn ein Bestellkopf ohne Positionen ist nicht sehr nützlich und eine Bestellposition ohne Kopf darf nicht existieren. Sie erreichen dies mit einem POST-Request wie in Listing 3.5 gezeigt.



**Tiefe Strukturen anlegen**

```

### Create Order with autogenerated UUID key
POST http://localhost:4004/admin/Orders
Content-Type: application/json

{
  "orderNo": "ZZ100",
  "currency_code": "EUR",
  "Items": [
    {
      "amount": 1,
      "product_ID": 102
    },
    {
      "amount": 2,
      "product_ID": 101
    }
  ]
}

```

Listing 3.5 Anlegen einer Bestellung mit Bestellpositionen

Auto-Generierung  
von UUID-  
Schlüsseln

Im Request Body werden dabei keine Schlüsselfelder für den Bestellkopf oder die Bestellpositionen mitgegeben. Wenn Sie den Request mit der Entität `OrderItems` aus Ihrem Domänenmodell vergleichen, sehen Sie außerdem, dass das Beziehungsfeld `parent` ebenfalls nicht angegeben wird. Wenn Sie Schlüsselfelder vom Typ UUID modellieren, kann CAP Ihnen automatisch global eindeutige Schlüsselwerte generieren. Optional könnten Sie aber auch eine vorher erzeugte UUID für das Feld `ID` im Request mitgeben. Des Weiteren kennt das CAP Service SDK natürlich das Domänenmodell, auf dem der Administrationservice basiert. Dies definiert die Kompositionsbeziehung zwischen `Orders` und `OrderItems`. So weiß CAP, dass es sinnvoll ist, das Beziehungsfeld `parent_ID` der Items mit der für den Order-Datensatz generierten ID für Sie zu füllen. Das Resultat dieser Automatismen sehen Sie im Response Body der Antwort auf den Request (siehe Listing 3.6). Dort sehen Sie, dass die Felder `ID` und `parent_ID` konsistent gefüllt sind.

```

{
  "@odata.context": "$metadata#Orders(Items())/$entity",
  "@odata.metadataEtag": "W/\"/QSriG98dJva5e04xpcHrmPUL6xi9LKxlR2p8/5abPA=\\"",
  "ID": "423d8ca5-a2bf-475f-9e46-433b7a0d7221",
  "orderNo": "ZZ100",
  "currency_code": "EUR",

```

```

"Items": [
  {
    "ID": "d932583f-5fcf-42fe-9ce5-a0a2d9987974",
    "amount": 1,
    "parent_ID": "423d8ca5-a2bf-475f-9e46-433b7a0d7221",
    "product_ID": 102
  },
  {
    "ID": "c750a23a-8177-47d3-839e-6567d842fcd9",
    "amount": 2,
    "parent_ID": "423d8ca5-a2bf-475f-9e46-433b7a0d7221",
    "product_ID": 101
  }
]
}

```

Listing 3.6 Response Body als Antwort auf die Anlage einer Bestellung mit Bestellpositionen

Im Browser sehen Sie Bestellungen mit den zugehörigen Positionen als Resultat der Abfrage `http://localhost:4004/admin/Orders?$expand=Items`.

Existierende Datensätze können mittels PUT-Requests geändert werden, wie beispielhaft in Listing 3.7 gezeigt wird.

Ändern  
existierender  
Datensätze

```

### Update Product
PUT http://localhost:4004/admin/Products(101)
Content-Type: application/json;IEEE754Compatible=true

{
  "name": "Updated Flat Screen",
  "price": "189.51"
}

```

Listing 3.7 PUT-Request zum Ändern des Produkts mit der ID = 101

Anhand des Response Body können Sie kontrollieren, ob die Änderung erfolgreich war. Beachten Sie unbedingt den zusätzlichen Parameter `IEEE754Compatible=true` im Wert des Feldes `Content-Type`. Er ist notwendig, wenn Ihr Request Zahlen vom Datentyp `Decimal` beinhaltet. JavaScript kennt, im Gegensatz zu CDS und OData, keinen nativen Datentyp `Decimal`. Fehlt dieser Parameter, wird der Request mit dem entsprechenden Hinweis als fehlerhaft beantwortet. Beachten Sie außerdem, dass der Wert von `price` in An-

Umgang mit  
Dezimalzahlen

führungszeichen zu setzen ist, anders als Sie es z. B. für Integer-Zahlen in JSON-Dokumenten gewohnt sind.

#### Löschen von Datensätzen

Den oben angelegten Lieferanten mit der ID 401 löschen Sie mit diesem einfachen Request:

```
DELETE http://localhost:4004/admin/Suppliers(401)
```

Beachten Sie, dass der Status der Antwort **204 No Content** lautet.

Das Löschen einer Bestellung erreichen Sie ebenfalls mit einem DELETE-Request, wobei Sie als Parameter die ID einer existierenden Bestellung benutzen müssen, z. B.:

```
DELETE http://localhost:4004/admin/Orders(4c1ea77c-ce95-439e-92d6-6f14cd35797b)
```

Insbesondere löscht dieser DELETE-Request auch automatisch alle zur Bestellung gehörenden Bestellpositionen. Dies liegt wieder daran, dass `Orders` eine `Composition of many OrderItems` enthält. Die Kompositionsbeziehung bedeutet, dass die Teile der Komposition (die `OrderItems`) alleine stehend keine Existenzberechtigung haben. Deswegen bedingt die Löschung der Komposition auch die Löschung seiner Teile.

### 3.6.4 Validierungen von Service-Requests

Das Domänenmodell enthält viele definierte Einschränkungen und Konsistenzbeziehungen, die zur Validierung vor der Ausführung eines Requests herangezogen werden können.

#### Prüfung von Typdefinitionen

Die einfachsten Einschränkungen entstehen durch Typdefinition von Elementen. Beispielsweise ist das Element `price` der Entität `Products` vom Typ `Dec(9,2)`, also eine Dezimalzahl mit maximal neun Stellen, wovon maximal zwei Stellen Nachkommastellen sein dürfen. Einer weiteren Einschränkung unterliegt das Element `currency_code` vom Typ `String(3)`. Entsprechend sollten die Requests in Listing 3.8 alle zu einer Fehlersituation führen.

```
### Update Product with invalid price format
PATCH http://localhost:4004/admin/Products(101)
Content-Type: application/json;IEEE754Compatible=true
```

```
{
  "price": "169.5134"
}
```

```
### Update Product with invalid currency_code format
PATCH http://localhost:4004/admin/Products(101)
Content-Type: application/json
```

```
{
  "currency_code": "EURO"
}
```

#### Listing 3.8 Zwei PATCH-Requests mit fehlerhaften Datentypen

Eine weitere wichtige Prüfung basiert auf den definierten Assoziationen, die in einer Datenbank auf Fremdschlüsselbeziehungen abgebildet werden. Aufgrund dessen sollte es nicht möglich sein, ein Produkt mit einer Lieferanten-ID anzulegen, zu der es keinen Eintrag in der Lieferantenstammtabelle gibt. Eine weitere Bedingung an das Produkt ist durch den Zusatz `not null` im Domänenmodell für das Element `name` definiert worden: Sie besagt, dass es nicht möglich sein soll, ein neues Produkt anzulegen, ohne einen Produktnamen zu vergeben. Probieren Sie entsprechende Requests aus, mit denen Sie eine Fehlersituation in der Antwort hervorrufen.

Prüfung basierend auf Assoziationen

## 3.7 Debuggen einer Service-Implementierung

Die generische Verarbeitung von OData-Requests durch das CAP Service SDK bietet Ihnen viele leistungsfähige Operation, ohne dass Sie eine Zeile Programmcode beisteuern müssen. Insbesondere die Nutzung der OData Query Options lässt Sie komplexe Datenabfragen mit einem beliebigen HTTP-Client durchführen.

Natürlich werden Sie Anforderungen vorfinden, die einen programmatischen Eingriff in die Verarbeitung erfordern. Typische Anwendungsfälle sind beispielsweise:

Anwendungsfälle für Programmcode

- Verpacken einer komplexen Datenabfrage in eine einfach aufzurufende Funktion mit Parametern
- Herstellen einer transaktionalen Klammer über Operationen, die mehrere Entitäten involvieren
- Implementierung von Datenabfragen, die nicht mit OData abzubilden sind
- Komplexe Datenabfragen im Falle der Nutzung anderer Protokolle als OData (z. B. natives REST)

Sobald Sie einen eigenen Code beisteuern, werden Sie auch auf die Debugging-Fähigkeiten Ihrer IDE zurückgreifen. CAP gibt Ihnen dazu keine Einschränkungen vor, sodass Sie für Ihre IDE vorhandene Debugger für Node.js oder Java nutzen können.

#### Debug-Konfiguration in Visual Studio Code

Wenn Sie Ihr Projekt in Visual Studio Code mittels `cds init` initialisiert haben (siehe Abschnitt 3.2, »Entwicklungsprojekt anlegen«), dann ist das Projekt schon mit einer passenden Debug-Konfiguration ausgestattet. Diese Konfigurationsdaten liegen in Ihrem Projekt im Ordner `.vscode/`, den die IDE standardmäßig nicht anzeigt. Sie können den Ordner aber im gewöhnlichen Dateieexplorer Ihres Betriebssystems sehen. Am besten ändern Sie die darin enthaltenen Dateien nicht mit einem Texteditor, sondern nutzen dazu nur die Konfigurationsdialoge von Visual Studio Code. Diese erreichen Sie über **File • Preferences • Settings**.

#### Virtuelle Felder in Modellen

Bevor Sie das Debuggen mit einem Codebeispiel ausprobieren, müssen Sie zuerst eine Logik implementieren. Erweitern Sie dazu das Domänenmodell für die Entität `Products` um ein zusätzliches Element `margin` (siehe Listing 3.9). Der Zusatz `virtual` bedeutet, dass dieses Element zwar Teil der Entität wird, aber nicht auf der Datenbank gespeichert wird. Solche virtuellen Felder sollen typischerweise das Ergebnis von Berechnungen enthalten, die bei einem GET-Request ausgeführt werden. Sie werden also Code schreiben, der die Marge zu einem Produkt ausrechnet und bei Serviceanfragen mitliefert.

```
entity Products {
  key ID          : Integer;
  name           : String(100) not null;
  stock         : Integer;
  price         : Decimal(9, 2);
  retail        : Decimal(9, 2);
  virtual margin : Decimal(9, 2);
  currency_code : String(3);
  supplier      : Association to Suppliers;
}
```

**Listing 3.9** Erweitertes Produktmodell mit dem virtuellen Element »margin«

#### Service-Implementierung mit JavaScript


Erstellen Sie eine neue Datei `admin-service.js` im Ordner `srv/`. Speichern Sie den Inhalt von Listing 3.10 in der Datei.

```
module.exports = (srv)=>{
  const { Products } = srv.entities
  srv.after ('READ', Products, (each)=>{
    let num = each.retail - each.price;
```

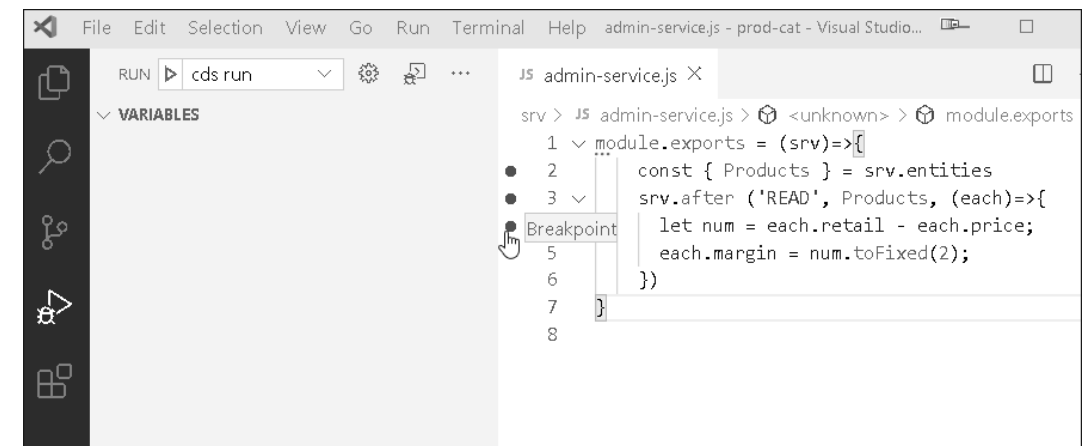
```
    each.margin = num.toFixed(2);
  })
}
```

**Listing 3.10** JavaScript-Code zur Implementierung eines »after.READ«-Ereignisbehandlers

Wenn Sie nun den bekannten GET-Request zum Lesen aller Produkte ausführen, sehen Sie das berechnete virtuelle Feld `margin` in der Antwort. Erreicht wurde dies durch die Berechnung bei der Behandlung des `after.READ`-Ereignisses für die Entität `Products`, d. h. nach dem Lesen der Produktdaten von der Datenbank.

Um nachzuvollziehen, was im Ereignisbehandler passiert, wechseln Sie zur Debugging-Sicht von Visual Studio Code, indem Sie auf das Icon  klicken. Stoppen Sie zuvor einen eventuell noch laufenden `cds watch`-Prozess. Im Editor zur Datei `admin-service.js` setzen Sie nun Breakpoints an der zweiten, dritten und vierten Zeile von Listing 3.10 (siehe auch Abbildung 3.9).

#### Breakpoints setzen



**Abbildung 3.9** Setzen von Breakpoints in der Service-Implementierung

Bewegen Sie dazu den Mauszeiger jeweils vor die gewünschte Zeilennummer, bis ein schwach roter Punkt erscheint. Klicken Sie dann mit der linken Maustaste, um den Breakpoint zu erstellen. Alle Breakpoints werden im Abschnitt **Breakpoints** links unten im Visual-Studio-Code-Fenster aufgelistet (siehe auch Abbildung 3.10).

Bevor Sie den Debugger starten, müssen Sie die CAP-Bibliotheken lokal in Ihrem Projekt installieren. Dies erreichen Sie durch Eingabe des Kommandos `npm install`. Die global installierten Bibliotheken des CDS Development Kits ist dafür nicht ausreichend. Das Kommando `npm init` zur Er-

#### Debugging durchführen

zeugung Ihres Projekts hat für Sie eine sogenannte *Run Configuration* mit dem Namen `cds run` in Visual Studio Code angelegt. Sie sehen diesen Namen im Abschnitt **RUN** links oben im Visual-Studio-Code-Fenster (siehe Abbildung 3.9). Starten Sie nun das Debugging, indem Sie auf den grünen Pfeil links neben `cds run` klicken.

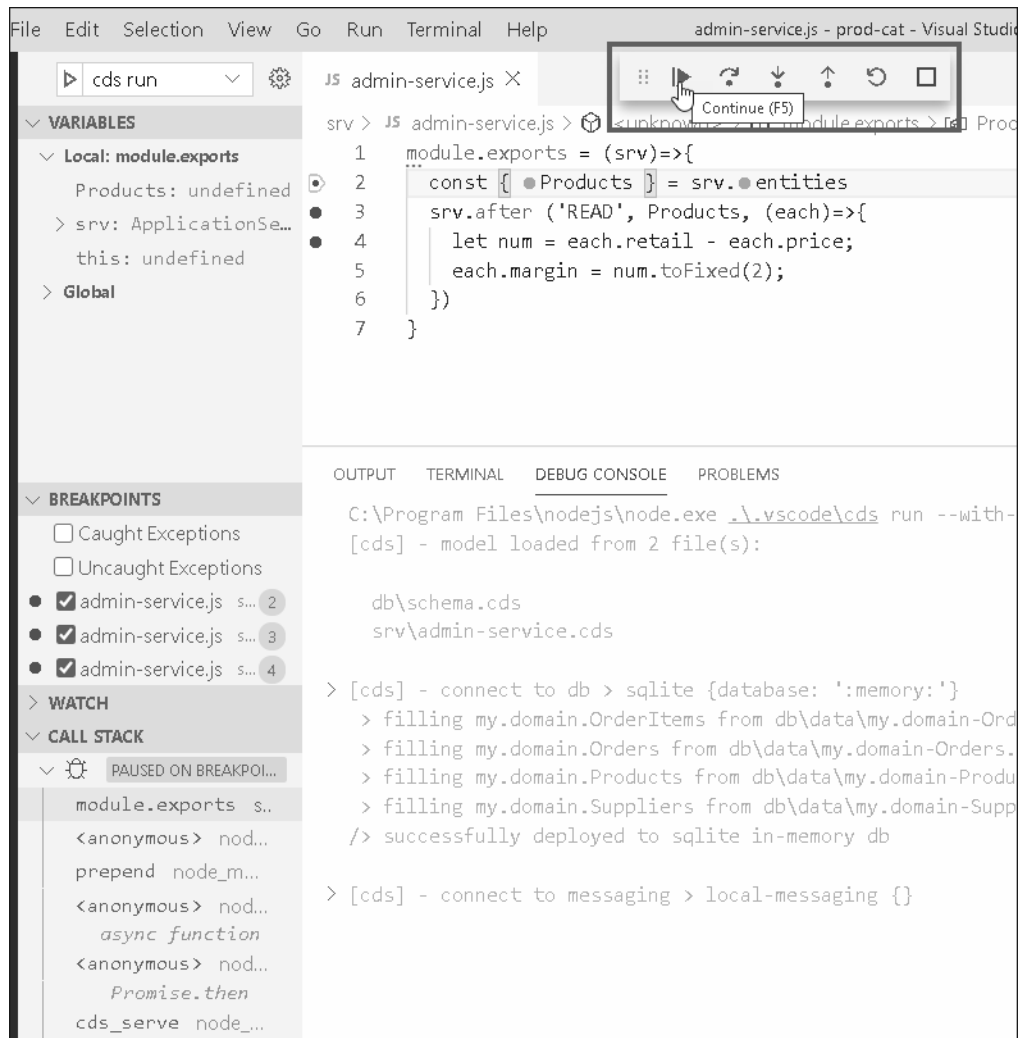


Abbildung 3.10 Aktiver Debugging-Prozess in Visual Studio Code

#### Steuerung des Debuggers

Ihr Terminalfenster wird nun durch die Debugging-Konsole ersetzt (siehe Abbildung 3.10). Dort sehen Sie mehrere Schaltflächen, mit denen Sie den Debugging-Prozess kontrollieren können. Wenn der Debugger an einem Breakpoint anhält, können Sie mit diesen Schaltflächen den Prozess entwe-

der weiterlaufen lassen, über eine Anweisung springen, in eine Unteroutine hineinspringen, eine Unteroutine verlassen, den Debugging-Prozess neu starten oder ihn beenden.

In der Debugging-Konsole in Abbildung 3.10 sehen Sie, dass `cds watch` gestartet wurde. Außerdem steht der Debugging-Prozess am ersten Breakpoint und wartet auf Ihre Aktion. Das CDS Service SDK ist also dabei, Ihren JavaScript-Code auszuführen, um die dort definierten Ereignisbehandlungsroutinen zu registrieren.

In der linken Debugging-Sicht gibt der Bereich **Variables** Auskunft über den aktuellen Zustand der Programmvariablen. Im Moment sehen Sie dort, dass die Variable `Products` noch undefiniert ist. Das können Sie ändern, indem Sie die Schaltfläche **Continue** betätigen (siehe Abbildung 3.10). Dies führt dazu, dass der Prozess weiterläuft und am nächsten Breakpoint stehen bleibt. Sie sehen, dass die Variable `Products` nun mit internen Informationen über das Produktmodell gefüllt ist. Klicken Sie den Strukturpfeil `>` vor `Products` an, um die Metadaten zu sehen.

Im Editorfenster Ihrer JavaScript-Datei sehen Sie, dass der Prozess nun vor der Funktion `srv.after` angehalten hat. Diese Funktion wird den Ereignisbehandlungler registrieren. Klicken Sie erneut auf **Continue**. Die Registrierung wird anschließend durchgeführt, und der Webserver wartet auf eingehende Anfragen eines HTTP-Clients.

Wechseln Sie zu Ihrem Browser und führen Sie eine GET-Anfrage durch, indem Sie in der Adresszeile `»http://localhost:4004/admin/Products«` eingeben. Ihr Browser wird keine Antwort anzeigen, da nun der Prozess am dritten Breakpoint angehalten hat, wie Sie durch Wechseln zum Visual-Studio-Code-Fenster sehen. Sie sind nun innerhalb des Ereignisbehandlers angelangt und können die jeweiligen Variableninhalte betrachten. Benutzen Sie die Schaltflächen **Step Over** und **Continue**. Der Ereignisbehandlungler wird entsprechend der Anzahl der Produkte in der Datenbank durchlaufen. Bei jedem Durchlauf wird die jeweilige Marge berechnet. Wenn alle Produkte durchlaufen wurden, ist der GET-Request abgearbeitet und Sie können das Ergebnis im Browser begutachten.

Variableninhalte

GET-Anfrage debuggen

#### Debugging-Modus für die Terminalausgabe

Neben dem Debuggen Ihres Programmcodes ist es zur Fehleranalyse oft auch erforderlich, die Terminalausgabe des CAP-Servers zu analysieren. Dort werden die Verarbeitungsschritte der CAP-Bibliotheken protokolliert. Es ist möglich, den Detailgrad dieses Protokolls zu erhöhen, indem Sie die Umgebungsvariable `DEBUG` im Terminal setzen. In der Windows PowerShell

nutzen Sie dazu beispielsweise das Kommando `$env:DEBUG = "true"`. Dadurch werden viele zusätzliche Informationen angezeigt (beispielsweise generierte SQL-Anweisungen), die für die Bearbeitung von Support-Anfragen nützlich sind.

### 3.8 Was sollten Sie aus diesem Kapitel mitnehmen?

Sie haben die ersten Schritte in der Implementierung Ihres Produktkatalogs gemacht. Die Spezifikation des Domänenmodells und des Verhaltens des Katalogs haben Sie direkt in CDS-Modelle für Entitäten und einen Service überführt. Neben den vielen gezeigten Details zu Modellierung, der HTTP-Request-Verarbeitung und den Testwerkzeugen, sollten sie Folgendes mitnehmen:

- **CDS-Modelle sind für Menschen gemacht**

Mit CDS modellieren Sie auf einer konzeptionellen Ebene, die eng mit einer betriebswirtschaftlichen Ebene verknüpft werden kann. Eine erste Skizze eines Domänenmodells, erstellt in einem gängigen Modellierungsformat (z. B. UML), kann direkt in ein CDS-Modell überführt werden, das über die Zeit mehr und mehr Details aufnimmt.

- **CAP-Services sind zielgerichtet**

Aus den einzelnen Rollen und Use Cases zu einer Anwendung ergeben sich auf natürliche Weise eine Anzahl von Services. Servicestrukturen sind zielgerichtete, zweckorientierte Projektionen aus dem Domänenmodell.

- **CAP-Projekte haben eine standardisierte Struktur**

Das CDS Development Kit hilft Ihnen bei der Anlage eines Projekts. Eine vorgegebene Struktur ermöglicht es Ihnen wiederum, Entwicklungswerkzeuge sofort und ohne weitere Konfiguration nutzen zu können (z. B. `cds watch`). Es ist möglich, von Strukturvorgaben abzuweichen, ohne guten Grund sollten Sie dies allerdings nicht tun.

- **CAP-Services kennen die offenen Standards OData/REST und HTTP**

HTTP ist das Kommunikationsprotokoll im Internet und in der Cloud. Sie profitieren von den vielen Werkzeugen zum Testen von HTTP-basierten Services. CAP-Services implementieren den OData-Standard, wodurch Ihnen automatisch die weit entwickelte OData-Query-Sprache und andere OData-Konzepte (wie Validierungen) für Ihre Service-Clients zur Verfügung stehen.

- **Sie nutzen die Werkzeuge Ihrer Wahl zur Entwicklung von Service-Implementierungen**

Die Implementierung von Ereignisbehandlern ermöglicht es Ihnen, spezifische Logik zu realisieren. Sie nutzen dazu die weitverbreiteten Sprachen JavaScript oder Java. CAP ist dadurch offen für die Entwicklungsumgebung Ihrer Wahl. Sie haben beispielsweise gesehen, wie Sie den Visual-Studio-Code-Debugger im CAP-Kontext nutzen können.