



Fullstack-Entwicklung mit SAP®

- › Fullstack-Anwendungen mit CDS, SAP Fiori Elements, SAPUI5 und Co. entwickeln
- › Programmiermodelle, Protokolle und Frameworks zur Integration von Front- und Backend
- › Fehlerbehandlung, Testen, Bereitstellung u. v. m.

Glavanovits · Koch
Kranicz · Olzinger

 **Rheinwerk**
Publishing

Kapitel 7

Entwicklung von OData-V4-Services mit dem SAP Cloud Application Programming Model

Mit dem SAP Cloud Application Programming Model können OData-Services relativ einfach erstellt werden. Diese Entwicklung kann auch lokal erfolgen und erfordert daher nicht viel Software. Zudem sind Sie relativ schnell bei einem testbaren OData-Service. Und bei Bedarf können spezifische Anforderungen auch durch eigene Logik umgesetzt werden.

Das *SAP Cloud Application Programming Model* stellt die fachlichen Anforderungen an die Anwendungsentwicklung in den Vordergrund. Wie diese Entwicklung umgesetzt wird, also die zugrunde liegenden Technologien, wird durch das Programmiermodell abstrahiert. Dadurch wird auch die Zusammenarbeit zwischen den Experten aus den Fachbereichen und den Entwicklern erleichtert. Eine wesentliche Rolle spielt dabei die Infrastruktur der Core Data Services (CDS) als leistungsfähige Sprache zur Erfassung von bereichspezifischen Datenmodellen.

Mit der sich schnell verändernden Welt der Cloud-Technologien und -Plattformen Schritt zu halten, ist eine große Herausforderung – sowohl für Hersteller als auch für Kunden. Heute aktuelle Technologien können übermorgen schon veraltet sein. SAP vermeidet eine einseitige Fokussierung im SAP Cloud Application Programming Model durch übergeordnete Konzepte und APIs. Letztendlich haben die Entwicklungs- und Architekturteams die Kontrolle darüber, welche Tools oder Technologien sie wählen oder welchen Architekturmustern sie folgen.

Das SAP Cloud Application Programming Model ist ein Framework aus Programmier- und Designsprachen, Bibliotheken und Werkzeugen für die Entwicklung von Services und Anwendungen im professionellen Unternehmensumfeld. Es gibt Ihnen Best Practices an die Hand und stellt eine Vielzahl sofort einsetzbarer Lösungen für wiederkehrende Aufgaben bereit. Es ermöglicht Ihnen die Entwicklung von Fullstack-Applikationen. Darunter versteht man alle Komponenten einer Applikation von der Persistenzschicht über die Geschäftslogik und Servicebereitstellung bis hin zur Benutzeroberfläche. Das Framework besteht aus einer Mischung aus bewährten sowie

weit verbreiteten Open-Source-Technologien und SAP-Technologien. Mit diesem Programmiermodell entwickelte Anwendungen werden entweder in JavaScript oder in Java entwickelt – beides Open-Source-Frameworks. Bei der JavaScript-Entwicklung wird auf Node.js in Kombination mit dem Express Framework gesetzt. Node.js ist der De-facto-Standard für die clientseitige JavaScript-Entwicklung. Das Express Framework ermöglicht eine einfache Entwicklung und Bereitstellung von Anwendungen auf Basis von Node.js.

In diesem Kapitel betrachten wir die Entwicklung von OData-V4-Services mit dem SAP Cloud Application Programming Model. Dazu wird der OData-Service, den wir bereits in Kapitel 6, »Entwicklung von OData-V4-Services mit dem ABAP RESTful Application Programming Model (Managed Scenario)«, mit dem ABAP RESTful Application Programming Model implementiert haben, hier erneut mit dem SAP Cloud Application Programming Model implementiert.

In Abschnitt 7.1 werden wir uns zunächst untersuchen, wie eine Domäne modelliert werden kann, d. h., welche Entitäten, Typen und Schlüssel uns zur Verfügung stehen. Außerdem werden wir uns die verschiedenen Arten von Beziehungen und Kompositionen ansehen. Da die Lokalisierung bei der Entwicklung von Diensten eine wesentliche Rolle spielt, werden wir in diesem Abschnitt auch darauf näher eingehen. In Abschnitt 7.2 lernen wir bestimmte Annotationen kennen, die es uns ermöglichen, Eingaben zu validieren und Benutzeroberflächen zu implementieren, und wir erfahren, wie benutzerdefinierte Logik in solchen Services implementiert werden kann. In Abschnitt 7.3 sehen wir, wie eine Anwendung mit dem SAP Cloud Application Programming Model eingerichtet und implementiert werden kann. In Abschnitt 7.4 sehen wir schließlich, wie Berechtigungsprüfungen auf Services angewendet werden können, die mit SAP Cloud Application Programming Model entwickelt wurden.

7.1 Domänenmodellierung

Domänenmodelle sind ein wichtiger Bestandteil bei der Entwicklung von Softwaresystemen, da sie die statischen Aspekte eines Problemfelds strukturiert und übersichtlich darstellen. Sie bilden die Grundlage für alle weiteren Schritte bei der Umsetzung einer Softwarelösung, angefangen von der Konzeptionierung bis hin zur Implementierung. Ein Domänenmodell beschreibt die Entitäten eines Problemfelds sowie deren Beziehungen untereinander. Dabei werden auch die Attribute und Eigenschaften der verschiedenen Entitäten erfasst, wodurch ein umfassendes Bild des Problemfelds erstellt werden kann. Da Domänenmodelle eine einheitliche Sprache zur Beschreibung von Problemstellungen zur Verfügung stellen, vereinfacht dies die Entwicklung von Softwaresystemen erheblich. Außerdem stellen Domänenmodelle die Basis für Persistenzmodelle, die in Datenbanken implementiert werden.

Neben der Verwendung von Domänenmodellen in der Erstellung von Persistenzmodellen werden diese zugleich auch für die Erstellung von Services verwendet. Services sind dafür zuständig, eine Schnittstelle zwischen dem Benutzer und dem Softwaresystem zu schaffen. Im Falle des SAP Cloud Application Programming Models leiten sich anhand der Domänenmodelle die entsprechenden Services ab.

Ähnlich wie beim Domain-driven Design zielt auch die Domänenmodellierung darauf ab, den Fokus von Projekten auf das Problemfeld zu legen. Hierbei wird eine enge Zusammenarbeit zwischen Entwicklern und Experten gefördert, um das Wissen über das Problemfeld laufend zu verfeinern.

Ein wichtiger Grundsatz im SAP Cloud Application Programming Model ist das KISS-Prinzip, »Keep it simple (and) stupid«.

Deshalb ist es bei der Modellierung von Entitäten wichtig, auf folgende Aspekte zu achten:

- **Sauberkeit**

Domänen sollen nicht mit technischen Details zugemüllt sein.

- **Prägnanz**

Es sollen kurze, aber sprechende Bezeichner verwendet werden. Es sollen möglichst flache Strukturen geschaffen und verwendet werden.

- **Nachvollziehbarkeit**

Es ist wichtig, dass nicht nur die Entwickler, sondern auch die Nutzer verstehen, warum eine Domäne auf eine gewisse Art und Weise erstellt wurde.

7.1.1 Namenskonventionen

Im Hinblick auf die Lesbarkeit und Verständlichkeit spielt es auch eine wichtige Rolle, Namenskonventionen einzuhalten. Das SAP Cloud Application Programming Model gibt diese zwar nicht vor, es wird allerdings empfohlen, trotzdem welche zu verwenden. Empfohlen wird die Verwendung von Konventionen, die auch in anderen Communities, wie z. B. Java, JavaScript, C, SQL usw. genutzt werden.

Hier könnte man beispielsweise auf folgende Konventionen setzen:

- zwischen Entitäten, Typen und Elementen unterscheiden
- Namensräume verwenden
- bevorzugen von Namensräumen gegenüber Top Level Contexts

Zum ersten Punkt: Die Bezeichner für Entitäten und Typen sollten immer mit Großbuchstaben beginnen, während Attribute in Kleinbuchstaben gehalten werden sollen. Weiterhin sollte für Entitäten der Plural verwendet werden. Typen sollten im Singular benannt werden. Daraus würden sich beispielsweise folgende Bezeichnungen ergeben: »Alben« für eine Entität, »Genre« für einen Typen und »Titel« für ein Attribut.

Außerdem sollte man soweit möglich einfache Bezeichner, die aus einem Wort bestehen, vergeben.

Um eindeutige Namen zu erhalten, ohne auf vollqualifizierte Namen zurückzugreifen, kann man *Namensräume* verwenden. Grundsätzlich sind diese lediglich ein Präfix, das sonst keine besondere Bedeutung hat. Dieses Präfix wird automatisch auf sämtliche relevanten Namen in der Datei angewandt.

Ein Beispiel hierfür sehen Sie in Listing 7.1:

```
namespace demo;  
entity TestOne {}  
entity TestTwo : TestOne {}
```

Listing 7.1 Verwendung eines Namensraums

Wie die inhaltlich gleiche Definition ohne die Verwendung von Namensräumen aussehen würde, sehen Sie in Listing 7.2.

```
entity demo.TestOne {}  
entity demo.TestTwo : demo.TestOne {}
```

Listing 7.2 Beispiel ohne Namensraum

Anhand der beiden vorangegangenen Beispiele wird ersichtlich, dass durch die Verwendung von Namensräumen der Code schlanker und besser lesbar wird.

Grundsätzlich gibt es für die Verwendung von Namensräumen folgende Empfehlungen:

- Werden die Modelle auch in anderen Projekten verwendet, sollte man Namensräume verwenden.
- Ist dies nicht der Fall, kann darauf verzichtet werden.
- Für die Definition eines Namensraums eignet sich der Ansatz reverse-domain-name gut. Das heißt, wenn die Unternehmensdomain »clouddna.at« lautet, wäre der korrelierende Namensraum »at.clouddna«.
- Man sollte möglichst versuchen, langlebige Namen zu vergeben, um nicht Gefahr zu laufen, diese Bezeichner in Zukunft ändern zu müssen.
- Weiterhin sollten die Namen kurz und knackig sein. Auf lange zusammenhängende Akronyme sollte verzichtet werden.



Namensräume sind optional

Namensräume sind optional und müssen daher nicht zwingend verwendet werden. Man kann genauso auch direkt eindeutige Bezeichner vergeben, wodurch auf Namensräume verzichtet werden kann.

Neben Namensräumen gibt es auch noch die Möglichkeit, *Kontexte* zu verwenden. Diese sind im Grunde ähnlich wie Namensräume. In älteren Verwendungen von CDS in SAP HANA wurden man gezwungen, sämtliche Definitionen umschlossen von einem Top-Level-Kontext zu definieren. Diese Einschränkung gibt es nun nicht mehr und sollte auch nicht verwendet werden. In der Regel sollten Namensräume immer vorrangig gegenüber eines Top-Level-Kontexts verwendet werden.

In Listing 7.3 sehen wir einen Vergleich zwischen der Verwendung eines Namensraums und eines Top-Level-Kontexts.

```
// Usage of namespace
namespace demo;
entity TestOne {}
entity TestTwo {}

// usage of top-level-context
context demo {
    entity TestOne {}
    entity TestTwo {}
}
```

Listing 7.3 Verwendung eines Namensraums vs. Verwendung eines Top-Level-Kontexts

7.1.2 Entitäten, Typen und Schlüssel

Für das Mapping eines Datenmodells sind Entitäten, Typen sowie Schlüssel unumgänglich. Deshalb sehen wir uns in den folgenden Abschnitten diese genauer an.

Entitäten

Im SAP Cloud Application Programming Model stellen Entitäten *Domänendaten* dar. Wenn Sie einfache Anforderungen in natürlicher Sprache niederschreiben und dann Substantive, Verben, Präpositionen und Schlüsselwörter extrahieren, können Sie daraus oft die Struktur eines Datenmodells ableiten.

Betrachten Sie das folgende Beispiel: Wir möchten eine Musik-Streaming-Plattform erstellen, auf der die Benutzer *Alben* und *Interpreten* durchsuchen und zwischen den beiden navigieren können. Die Alben werden nach *Genre* sortiert. Betrachtet man diese konkrete Anforderung, so ergeben sich daraus die Entitäten *Album* und *Interpret* und ein speziell definierter Typ *Genre*. Außerdem stehen die beiden Entitäten in einer Beziehung zueinander. In Listing 7.4 sehen wir, wie dies im SAP Cloud Application Programming Model aussehen könnte.

```
entity Album {  
    key ID: UUID;  
    title: String;  
    year: Integer;  
    genre: Genre;  
    interpret: Association to Interpret;  
}  
  
entity Interpret {  
    key ID: UUID;  
    name: String;  
    albums: Association to many Album on albums.interpret = $self;  
}  
  
type Genre : String enum {  
    Rock, Pop, Rap  
}
```

Listing 7.4 Beispielentitäten in CDS

Oftmals ergeben sich Anwendungsfälle, in denen mehrerer Tabellen gewisse Überschneidungen haben, also die gleichen Attribute vorweisen. Um hier die Implementierung und spätere Wartung zu vereinfachen, kann auf das Konzept der Vererbung zurückgegriffen werden. Dazu muss bei der Definition der Entität nach der Bezeichnung dieser nur jener Typ angegeben werden, von welchem geerbt werden soll. Wie dies praktisch aussieht, sehen Sie in Listing 7.5.

```
entity Demo : Demo2{  
    key id : UUID;  
    name : String;  
}  
  
type Demo2 {  
    description : String;  
}
```

Listing 7.5 Entität mit Vererbung

Typen

Wie am vorangegangenen Beispiel zu sehen ist, werden für die Definition von Entitäten verschiedene Typen benötigt. CDS kommt standardmäßig mit einem kleinen Set an generellen vordefinierten Typen. Diese Typen können verwendet werden, um Elementen einen Typ zuzuordnen oder aber auch um eigens definierte Typen von bereits bestehenden abzuleiten.

In Tabelle 7.1 sehen Sie eine Auflistung bereits vordefinierter Typen, die wir in einem SAP-Cloud-Application-Programming-Model-Projekt verwenden können.

CDS-Typ	Argumente	Beispiel	SQL
UUID	opaque 36 characters string	123e4567-e89b-12d3-a456-426614174000	NVARCHAR(36)
Boolean		true false	BOOLEAN
UInt8		133	TINYINT
Int16		1337	SMALLINT
Int32		1337	INTEGER
Integer		1337	INTEGER
Int64		1337	BIGINT
Integer64		1337	BIGINT
Decimal	(precision, scale)	15.2	DECIMAL
Double		15.2	DOUBLE
Date		'2023-03-28'	DATE
Time		'22:15:00'	TIME
DateTime	sec precision	'2023-03-28T22:15:00Z'	TIMESTAMP
Timestamp	0.1µs precision	'2023-03-28T22:15:00.000Z'	TIMESTAMP
String	(length)	'Hello World!'	NVARCHAR
Binary	(length)		VARBINARY
LargeBinary			BLOB
LargeString		'Hello World!'	NCLOB

Tabelle 7.1 Verfügbare Datentypen in CDS

Neben den aufgezählten Datentypen bietet SAP im Namensraum `@sap/cds/common` weitere Typen an, die für die Wiederverwendung gedacht sind. Man sollte, so gut es geht, bereits bestehende Datentypen verwenden und diese nicht neu definieren. Dadurch profitiert man in folgenden Belangen:

- prägnante und verständliche Datenmodelle
- bessere Interoperabilität zwischen verschiedenen Anwendungen, da die gleichen Typen verwendet werden

- bewährte Best Practices aus echten Applikationen
- optimierte Implementierung und Performance
- Localization wird out-of-the-box unterstützt

Als Beispiel kann man hier den Typ »Country« heranziehen. Dieser Typ kann verwendet werden, um beispielsweise den zweistelligen ISO-Code eines Landes zu speichern. Weiterhin gibt es noch Typen wie z. B.:

- Currency
- Language
- Locale

Neben den Typen spielen auch Schlüsselfelder eine wichtige Rolle, um ein Datenmodell abzubilden. Hierbei unterscheidet man zwischen zwei verschiedenen Arten von Schlüsseln:

- Primärschlüssel
- Fremdschlüssel

Wofür diese benötigt werden und inwieweit sie sich voneinander unterscheiden, sehen wir uns im Folgenden an.

Primärschlüssel

Primärschlüssel werden verwendet, um Entitäten eindeutig identifizierbar zu machen. Durch diesen Schlüssel erhält somit jeder Datensatz diese Entität ein eindeutiges Merkmal, worüber man dann auf einen speziellen Datensatz zugreifen kann. Um in SAP Cloud Application Programming Model ein Feld als Primärschlüssel zu definieren, wird das Schlüsselwort `key` verwendet. In Listing 7.6 sehen wir eine beispielhafte Entität, die als Primärschlüssel ein Feld vom Typ `UUID` hat.

```
entity Demo {  
    key id: UUID;  
    description: string;  
}
```

Listing 7.6 Entität mit Primärschlüssel

Es ist ebenso möglich, in einer Entität mehrere Felder zum Primärschlüssel zu machen, hierbei spricht man dann von einem zusammengesetzten Primärschlüssel. In Listing 7.7 sehen Sie, wie die Definition einer solchen Entität aussehen könnte. Im Grunde unterscheidet sich diese Entität nicht stark von der vorangegangenen. Sie weist lediglich mehrere Felder, die mit dem Schlüsselwort `key` versehen sind, auf.

```
entity Demo {
    key id: UUID;
    key id2: UUID;
    description: string;
}
```

Listing 7.7 Entität mit zusammengesetztem Primärschlüssel

Primärschlüssel können grundsätzlich die folgenden Merkmale aufweisen:

■ **Einfach**

Dies bedeutet, dass sie aus einem einzelnen Feld oder aus mehreren Feldern bestehen.

■ **Technisch**

Dies bedeutet, dass sie nicht zwingend eine semantische Bedeutung vorweisen müssen. Diese Art von Schlüssel wird in den meisten Fällen automatisch generiert, beispielsweise durch eine Datenbanksequenz.

■ **Unveränderlich**

Dies bedeutet, dass die Schlüsselfelder nach initialer Erstellung nicht mehr verändert werden können. Eine Veränderung des Primärschlüssel könnte möglicherweise zu Inkonsistenzen oder Fehlern in der Datenbank führen, weshalb eine nachträgliche Änderung nicht erlaubt ist.

Best Practises für Primärschlüssel

SAP Cloud Application Programming Model unterstützt alle genannten Punkte, der Empfehlung nach sollte so gut und oft es geht, ein einfacher, technischer, unveränderlicher Primärschlüssel zum Einsatz kommen.

Die Verwendung von Binärdaten als Primärschlüssel ist dringend zu vermeiden. Aufgrund der Tatsache, dass Binärdaten (z. B. Dokumente, Bilder, Video, Audio) meist relativ groß und schwer zu vergleichen sind, eignen sich kurze und einfach vergleichbare Werte besser.



Des Weiteren wird empfohlen, kanonische Primärschlüssel zu verwenden. Kanonische Primärschlüssel sind nichts anderes als Schlüssel, die bestimmten Konventionen folgen. Dies verbessert die Lesbarkeit, Verständlichkeit und Wartbarkeit. Im Grunde haben diese Schlüssel eindeutige, aber beschreibende Bezeichnungen. Durch die beschreibenden Bezeichnungen haben Entwickler einen besseren Überblick und können etwaige Probleme rascher lösen.

Wenn eine Entität `Customer` vorliegt, könnte der kanonische Schlüssel beispielweise `customer_id` lauten. So wird durch die Bezeichnung sofort ersichtlich, welche Entität

damit referenziert wird. Natürlich kann hier auch jeder beliebige andere Name verwendet werden, schließlich ist die Verwendung von kanonischen Bezeichnern lediglich eine Empfehlung und keine Einschränkung.

Außerdem können bei Bedarf Basisentitäten definiert werden, die dann im Projekt bei den einzelnen Entitäten über Vererbung verwendet werden können. Dies hat den Vorteil, dass Sie Teile, die immer gleich sind, an zentraler Stelle einmalig definieren können.

Eine weitere Empfehlung ist, für technische Primärschlüssel den Typ `UUID` zu wählen. `UUIDs` bringen zwar einen gewissen Overhead und somit auch Performanceeinbußen mit sich, allerdings werden diese Nachteile durch eine Reihe an Vorteilen eliminiert:

- **UUIDs sind universal.**

`UUIDs` sind über alle Systeme der Welt hinweg eindeutig. Im Vergleich dazu sind Datenbanksequenzen nur bis zu den Systemgrenzen eindeutig. Möchten Sie Daten mit anderen Systemen austauschen, haben Sie bei der Verwendung von `UUIDs` somit kein Problem. Im Falle von Datenbanksequenzen muss aber unbedingt darauf geachtet werden, die Eindeutigkeit aufrechtzuerhalten.

- **UUIDs erlauben die Verteilung von Seed-Werten.**

Betrachtet man Sequenzen, wird immer ein zentraler Service benötigt, der diese verwaltet. Dies könnte beispielsweise eine Datenbankinstanz oder ein Schema sein. In verteilten Systemlandschaften wird dies zu einem Problem. `UUIDs` hingegen benötigen keine zentrale Instanz.

- **Datenbanksequenzen sind schwer zu erraten.**

Möchte man innerhalb einer Transaktion Einträge in einer Entität anlegen, die einen Fremdschlüssel auf eine übergeordnete Entität beinhaltet, müsste der Werte aus der Datenbanksequenz explizit ausgelesen werden. Dafür gäbe es zwar Funktionen, allerdings würden diese einem bei der Verwendung von `UUIDs` erspart bleiben.

- **Primärschlüssel sind autogeneriert.**

Primärschlüssel vom Typ `UUID` werden bei Insert-Statements automatisch generiert und können so weggelassen werden.

Die Verwendung von Integer-Sequenzen als Primärschlüssel bietet sich dann an, wenn man mit großen Datenmengen zu tun hat, andernfalls sollten – wenn möglich – `UUIDs` eingesetzt werden. Oftmals hat man auch semantische Primärschlüssel, die z. B. aus zusammengesetzten Daten wie Datum oder Uhrzeit bestehen. In diesem Fall kann je nach Belieben eine Sequenz oder ein `UUID` verwendet werden.



Universal Unique Identifier

UUID ist die Abkürzung für Universal Unique Identifier. Häufig wird dafür auch Globally Unique Identifier (GUID) verwendet. Er besteht aus einer 128-Bit-Zeichenkette, die nach einem bestimmten Algorithmus generiert wird. Diese besteht aus mehreren Komponenten:

- 60-Bit-Zeitstempel-Komponente
- 48-Bit-Knoten-ID
- 6-Bit-Zufallszahl

Fremdschlüssel

Neben den Primärschlüsseln gibt es eine zweite Art von Schlüsseln: Fremdschlüssel. Dabei handelt es sich im Wesentlichen um Attribute, die sich auf den Primärschlüssel einer anderen Tabelle beziehen. Das heißt, Fremdschlüssel werden verwendet, um Beziehungen zwischen Entitäten darzustellen. Der Fremdschlüssel ist also immer in der abhängigen Entität zu finden. In den meisten Fällen wird der Name der anderen Entität oder ihr Primärschlüsselname als Bezeichner des Fremdschlüssels verwendet.

Darüber hinaus wird beim Einfügen oder Ändern eines Datensatzes mit einem Fremdschlüssel geprüft, ob dieser Schlüssel überhaupt existiert. Wenn diese Prüfung fehlschlägt, kann der Datensatz nicht eingefügt werden. Darüber hinaus wird beim Löschen von Datensätzen geprüft, ob der zu löschen Datensatz von anderen Tabellen referenziert wird. Ist dies der Fall, müssen zuerst die abhängigen Datensätze entfernt werden, bevor der referenzierte Eintrag gelöscht werden kann. Diese Vorkehrungen dienen der Wahrung der Datenkonsistenz und der Datenintegrität, um Fehler in der Datenbank zu vermeiden.

7.1.3 Kompositionen und Assoziationen

Kompositionen und Assoziationen sind ebenfalls ein wichtiger Bestandteil für die Erstellung von Datenmodellen. Was diese genau bewirken, werden wir in den folgenden Abschnitten sehen.

Kompositionen

Kompositionen werden verwendet, um eine »Enthalten in«-Beziehung darzustellen. Das Besondere an dieser Beziehung ist, dass die abhängige Entität nicht ohne die übergeordnete Entität existieren kann. Das bedeutet, dass die übergeordnete Entität praktisch aus der abhängigen Entität besteht. In den meisten Fällen enthält die übergeordnete Entität Metadaten, während die abhängige Entität letztlich bestimmte Daten speichert.

Ein klassisches Beispiel für eine solche Konstellation sind die Entitäten `Invoice` und `InvoiceItem`. Eine Rechnung (engl. `invoice`) besteht bekanntlich aus mehreren Positionen (engl. `item`), und eine Position gehört zu genau einer Rechnung. Außerdem kann ein Rechnungsosten nicht ohne eine Rechnung existieren.

Um eine Eigenschaft innerhalb einer Entität als Komposition zu deklarieren, wird das Schlüsselwort `Composition of` verwendet. Zusätzlich kann `of many` verwendet werden, um anzugeben, dass es sich um eine 1:n-Komposition handelt. In der `on`-Klausel muss schließlich angegeben werden, über welche Eigenschaft die abhängige Entität auf die übergeordnete Entität abgebildet wird.

In Listing 7.8 sehen wir ein Beispiel für eine Komposition in Form der Entitäten `Invoice` und `InvoiceItem`. Natürlich können die Entitäten um beliebige andere Felder erweitert werden.

```
entity Invoice {  
    key id : UUID;  
    invoiceItems : Composition of many InvoiceItems on invoiceItems.parent =  
$self;  
}  
  
entity InvoiceItems {  
    key id : UUID;  
    key invoiceId : Association to Invoice;  
    amount : Float;  
}
```

Listing 7.8 Beispiel einer Komposition

Mit dem SAP Cloud Application Programming Model steht eine Reihe von Funktionen standardmäßig zur Verfügung, z. B. die folgenden:

- Eigenschaften, die als Kompositionen enthalten sind, werden automatisch im Service exponiert.
- Deep Insert, Update und Upsert (ein Portmanteau (Kofferwort) aus Update und Insert) werden automatisch unterstützt.
- Die *Cascade-Delete-Strategie* wird standardmäßig für Löschungen verwendet.



Cascade-Delete-Strategie

Bei Löschvorgängen, die sich auf einen Datensatz einer übergeordneten Entität beziehen, spielt die *Cascade-Delete-Strategie* eine Rolle. Dabei wird berücksichtigt, dass nicht nur die übergeordnete Entität, sondern auch die von ihr abhängigen Datensätze gelöscht werden. Dadurch werden Inkonsistenzen in der Datenbank vermieden.

In Listing 7.8 wurde für jede der über- und untergeordneten Entitäten eine eigene Entität erstellt. Mithilfe von aspects kann hier Code eingespart werden. Die Definition einer Komposition mit Aspekten ist in Listing 7.9 dargestellt.

```
entity Invoice {
    key id : UUID;
    invoiceItems : Composition of many InvoiceItems;
}

aspect InvoiceItems {
    key id : UUID;
    amount : Float;
}
```

Listing 7.9 Komposition mit aspects

Wie in diesem Beispiel gezeigt, werden bei dieser Art der Verwendung die To-parent-Beziehung und die on-Klausel weggelassen.

Alternativ können Sie auch auf eine Definition mit anonymen Inline-Typen zurückgreifen (siehe Listing 7.10).

```
entity Invoice {
    key id : UUID;
    invoiceItems : Composition of many {
        key id : UUID;
        amount : Float;
    }
}
```

Listing 7.10 Komposition mit anonymen Inline-Typen

Auch wenn die Definition der beiden Varianten kürzer und knackiger ist als die der ersten, führen alle drei Varianten zum gleichen Ergebnis.

Mithilfe von Kompositionen können natürlich auch Many-to-many-Beziehungen abgebildet werden. Dazu ist allerdings eine Zwischentabelle mit zwei One-to-many-Beziehungen erforderlich.

Assoziationen

Assoziationen können auch verwendet werden, um Beziehungen zwischen verschiedenen Entitäten abzubilden. Im Gegensatz zu Kompositionen müssen die Einträge in der übergeordneten Entität nicht unbedingt vorhanden sein, d. h., der Fremdschlüssel muss nicht gefüllt sein. Darüber hinaus kann man über Assoziationen zwischen

den einzelnen Entitäten navigieren. Es werden drei Arten von Beziehungen unterschieden:

■ **One-to-one-Beziehung**

One-to-one-Beziehungen deklarieren eine Abhängigkeit, bei der jeder Eintrag der übergeordneten Entität mit einem Eintrag der abhängigen Entität verbunden ist.

■ **One-to-many-Beziehung**

Bei One-to-many-Beziehungen werden null oder mehr Instanzen einer anderen Entität der Instanz einer Entität zugeordnet.

■ **Many-to-many-Beziehung**

Eine Many-to-many-Beziehung liegt vor, wenn mehrere Instanzen einer anderen Entität einer Instanz zugeordnet sind. Das Gleiche gilt in beide Richtungen.

Dies wird oft als *Kardinalität* bezeichnet, die bestimmt, wie viele Instanzen einer Entität von einer zweiten Entität abhängig sind.

Navigationseigenschaften ermöglichen die Navigation zwischen abhängigen Entitäten. Wenn Sie beispielsweise eine Entität haben, die eine One-to-many-Beziehung mit einer anderen Entität hat, gibt es eine Eigenschaft auf der ersten Entität, die Sie verwenden können, um zu den entsprechenden Datensätzen in der zweiten Entität zu springen. Das SAP Cloud Application Programming Model unterstützt auch die referenzielle Integrität. Dadurch wird sichergestellt, dass die Abhängigkeiten zwischen Entitäten konsistent sind. Dies vermeidet das Auftreten von Fehlern in der Datenbank und fördert auch die Leistung.

Um eine Beziehung zwischen Entitäten zu deklarieren, wird das Schlüsselwort `Association to` verwendet. Für One-to-many-Beziehungen müssen zusätzlich der `many`-Qualifier und die `on`-Klausel angegeben werden. In Listing 7.11 sehen wir ein Beispiel dafür, wie eine Beziehung zwischen zwei Entitäten definiert werden kann.

```
entity Demo {  
    key id : UUID;  
    name : String;  
    items : Association to many DemoItem on items.demo = $self;  
}  
  
entity DemoItem {  
    key id : UUID;  
    demo : Association to Demo;  
}
```

Listing 7.11 Beispiel einer Assoziation

Wie in diesem Beispiel gezeigt, gibt es für jeden Eintrag in der Entität Demo mehrere Einträge in der Entität DemoItem. Zusätzlich wird ein Rückverweis in die Entität DemoItem eingefügt. Dadurch ist es möglich, die Beziehung in beide Richtungen aufzulösen. Many-to-many-Beziehungen werden grundsätzlich genauso definiert wie One-to-many-Beziehungen, jedoch muss zusätzlich eine Zwischentabelle angelegt werden. Diese Zwischentabelle enthält jeweils einen Fremdschlüssel zu den beiden Tabellen, die mit einer Many-to-many-Kardinalität verknüpft werden sollen. Diese beiden Fremdschlüssel werden auch als zusammengesetzter Primärschlüssel in dieser Tabelle verwendet. Natürlich können dieser Zwischentabelle weitere Felder hinzugefügt werden. In Listing 7.12 sehen wir ein Beispiel für eine solche Beziehung mit einer Zwischentabelle.

```
entity Demo1 {  
    key id : UUID;  
    items : Association to many Demo1_to_Demo2 on items.demo1 = $self;  
    ...  
}  
  
entity Demo2 {  
    key id : UUID;  
    items : Association to many Demo1_to_Demo2 on items.demo2 = $self;  
    ...  
}  
  
entity Demo1_to_Demo2 {  
    key demo1 : Assocation to Demo1;  
    key demo2 : Assocation to Demo2;  
}
```

Listing 7.12 Beispiel einer Many-to-many-Assoziation

Wie in Kompositionen können auch in Assoziationen aspects oder anonyme Inline-Typen verwendet werden. Die Definitionen dieser Typen sind grundsätzlich identisch.

7.1.4 Lokalisierte Daten

Im SAP Cloud Application Programming Model ist es auch möglich, die Lokalisierung zu nutzen. Das bedeutet, dass mehrsprachige Texttabellen out of the box implementiert werden können. Darüber hinaus werden diese mehrsprachigen Texte automatisch und korrekt gelesen. Die Sprache wird dabei durch die vom Benutzer angegebene Sprache bestimmt. Existieren die Texte nicht in der gewünschten Sprache, wird

eine Standardsprache verwendet. Die Sprachschlüssel basieren auf dem ISO-639-1-Format.

Um bestimmte Felder in Entitäten als mehrsprachig zu definieren, ist das Schlüsselwort `localized` zu verwenden. Die einzige Einschränkung hierbei ist, dass die Schlüsselfelder der Entität keine Assoziationen sein dürfen. Listing 7.13 enthält ein Beispiel für die Definition einer solchen Entität.

```
entity Demo {  
    key id : UUID;  
    name : localized String;  
    ...  
}
```

Listing 7.13 Entität mit einer lokalisierten Eigenschaft

Im Prinzip genügt der Zusatz zur Spezifikation der Eigenschaften; den Rest erledigt der CDS-Compiler für uns. Zu diesem Zweck legt der Compiler eine eigene Tabelle an. Außerdem verwendet der Compiler die Konzepte der Assoziationen und Kompositionen, deren Namen von der jeweiligen Entität abgeleitet sind. Außerdem wird das Suffix `.texts` hinzugefügt. In unserem Fall hieße diese Tabelle `Demo.texts`.

Neben dem Schlüssel aus der Hauptentität und dem Sprachschlüssel enthält diese Entität alle Felder, die als lokalisiert gekennzeichnet sind. In Listing 7.14 sehen wir, wie diese Texttabelle in unserem Fall aussehen würde.

```
entity Demo.texts {  
    key locale : sap.common.Locale;  
    key ID : UUID; // Primary key from table "Demo"  
    name : String;  
}
```

Listing 7.14 Texttabelle für Demo-Entität

Darüber hinaus wird die eigentliche Entität `Demo` um eine Assoziation und Komposition zur Texttabelle erweitert (siehe Listing 7.15).

```
extend entity Demo with {  
    texts : Composition of many Demo.texts on texts.ID = ID;  
    localized : Association to Demo.texts on localized.IID = ID and  
    localized.locale = $user.locale;  
}
```

Listing 7.15 Entitätserweiterung für Lokalisierung

Die Komposition `texts` enthält einen Verweis auf alle Übersetzungen der jeweiligen Entität. Die Assoziation `localized` hingegen verweist auf alle Übersetzungen, eingeschränkt auf die Sprache des Benutzers. Diese Sprache kann aus dem Request-Objekt ausgelesen werden.

Letztendlich wird ein View generiert, der das Auslesen der Übersetzungen mit einem entsprechenden Fallback ermöglicht. Dies ist in Listing 7.16 zu sehen.

```
entity localized.Demo as SELECT from Demo {*,  
    coalesce (localized.name, name) as name  
}
```

Listing 7.16 View für lokalisierte Tabelle

Pseudovariable für User-Sprache



Wie bereits erwähnt, wird die Sprache des Benutzers anhand der Pseudovariablen `$_user.locale` in der Anfrage ermittelt, die dann zum Lesen der Übersetzungen verwendet wird.

Es gibt eine Reihe von Regeln für die Einstellung dieser Sprache, die in der folgenden Reihenfolge abgearbeitet werden:

1. Wenn vorhanden, wird die Sprache aus dem URL-Parameter `sap-locale` gelesen.
2. Wenn vorhanden, wird die Sprache aus dem URL-Parameter `sap-language` gelesen.
3. Der erste Eintrag stammt aus dem Header `Accept-Language`.
4. Ansonsten wird die Standardsprache verwendet, die auf Anwendungsebene definiert werden kann.

7.2 Service-Bereitstellung

In diesem Abschnitt lernen wir Annotationen kennen, die es uns ermöglichen, Eingaben zu validieren und Benutzeroberflächen zu implementieren, und wir erfahren, wie benutzerdefinierte Logik in solche Dienste implementiert werden kann.

7.2.1 Input-Validierung

Um von vornherein sicherzustellen, dass nur gültige Einträge gespeichert werden können, gibt es im SAP Cloud Application Programming Model eine Reihe von Anmerkungen, die für die Validierung der Einträge zuständig sind, wie folgt:

- `@readonly`
- `@mandatory`

- `@assert.unique`
- `@assert.integrity`
- `@assert.target`
- `@assert.format`
- `@assert.range`
- `@assert.notNull`

In den folgenden Abschnitten werden wir diese im Detail betrachten.

@readonly-Felder

Wenn einzelne Felder mit der `@readonly`-Annotation gekennzeichnet sind, sind sie vor nachfolgenden Schreiboperationen geschützt, z. B.:

```
@readonly field1 : String;
```

@mandatory-Felder

Als `@mandatory` gekennzeichnete Felder werden bei der Erstellung oder Aktualisierung auf leeren Inhalt geprüft. Diese müssen ausgefüllt werden. Im Falle von Null-Werten oder leeren Zeichenfolgen tritt ebenfalls ein Fehler auf, z. B.:

```
@mandatory field1 : String;
```

@assert.unique-Einschränkung

Um festzulegen, dass die Datensätze einer Entität in Bezug auf bestimmte Felder eindeutig sein müssen, kann die Einschränkung `@assert.unique` verwendet werden. Dazu wird die Einschränkung einfach über die Entität mit den entsprechenden Feldern definiert. Die Syntax lautet `@assert.unique.<Name>`. Ein Beispiel sehen Sie hier:

```
@assert.unique : {  
    demo : [field1, field2, ...]  
}
```

Primärschlüsselfelder müssen nicht mit der Eindeutigkeitsbeschränkung versehen werden, sie sind automatisch eindeutig.

@assert.integrity-Einschränkung

Assoziationen und Kompositionen mit einer To-one-Kardinalität können automatisch auf referentielle Integrität geprüft werden. Wenn dies nicht gewünscht ist, kann die Einschränkung `@assert.integrity` verwendet werden, um dies für Assoziationen, Entitäten oder Dienste zu kontrollieren. Ein Beispiel könnte so aussehen:

```
@assert.integrity : false or true
```

@assert.target-Einschränkung

Die Einschränkung `@assert.target` kann verwendet werden, um sicherzustellen, dass der entsprechende Datensatz, auf den verwiesen wird, bei Schreiboperationen von Fremdschlüsseln tatsächlich existiert. Diese Einschränkung kann auch auf mehrere Fremdschlüssel in einer Entität gleichzeitig angewendet werden. Allerdings wirkt sich diese Einschränkung nur auf Schreiboperationen aus, nicht auf Löschoperationen. Schließlich dient sie der Überprüfung von Benutzereingaben und nicht der Gewährleistung der referenziellen Integrität. Daher wirkt diese Beschränkung gleichzeitig mit der `@assert.integrity`-Beschränkung. Wenn das `@assert.target`-Constraint verletzt wird, d. h. ein Fremdschlüssel übergeben wird, der nicht existiert, wird eine entsprechende Fehlermeldung ausgelöst. Ein Beispiel sieht aus wie folgt:

```
texts : Association to Text @assert.target;
```

@assert.format-Einschränkung

Wenn eine Anforderung verlangt, dass die Werte bestimmter Felder einem bestimmten Muster folgen müssen, kann dies über Regelausdrücke und die Einschränkung `@assert.format` eingeschränkt werden. Zum Beispiel:

```
demo : String @assert.format: '[a-z]';
```

@assert.range-Einschränkung

Um festzulegen, dass eine Eigenschaft nur Werte innerhalb eines bestimmten Bereichs annehmen kann, kann z. B. die Einschränkung `@assert.range` verwendet werden. Hier muss nur eine untere und obere Grenze angegeben werden, gegen die die Validierung erfolgt. Die beiden Grenzen werden als ein geschlossenes Intervall betrachtet. Zum Beispiel:

```
demo : Integer @assert.range : [1, 10];  
demo2 : String @assert.range enum { m, w };
```

@assert.notNull

Um die Nicht-Null-Prüfung zu umgehen, können einzelne Attribute mit `@assert.notNull` annotiert werden, was notwendig sein kann, wenn dieses Feld automatisch gefüllt wird. Zum Beispiel:

```
demo : String not null @assert.notNull : false;
```

7.2.2 OData-Annotationen

Besonders in SAP-Fiori-Elements-Anwendungen oder SAPUI5-Anwendungen, die Smart Controls verwenden, ist es sehr schwierig, ohne Annotationen auszukommen.

Obwohl diese auch direkt lokal in der Anwendung eingefügt werden könnten, ist es ratsam, sie in der Backend-Anwendung, also in der SAP-Cloud-Application-Programming-Model-Anwendung, zu speichern. Im Folgenden sehen wir uns an, wie solche OData-Annotationen in unserer SAP-Cloud-Application-Programming-Model-Anwendung angewendet werden können. In den folgenden Abschnitten werden wir einige der wichtigsten Annotationen im Detail betrachten.

Terms und Eigenschaften

OData definiert eine zweistufige Schlüsselstruktur für die Annotationen, die aus folgenden Elementen besteht:

- Vocabulary
- Term

In der Praxis sieht das folgendermaßen aus: @<Vocabulary>.<Term>.

Der Wert, der einer Annotation zugewiesen wird, kann verschiedene Arten annehmen, wie z. B. die folgenden:

- primitiver Typ
- Record
- Collection

Wenn wir z. B. unserer Entität Book eine Bezeichnung geben wollen, können wir dies wie in Listing 7.17 gezeigt tun.

```
@Common.Label: 'Books'  
entity Book { /* ... */ }
```

Listing 7.17 Entität mit Label annotiert

Qualifizierte Annotationen

OData ermöglicht auch die Zuweisung von qualifizierten Annotationen. Dies bedeutet nichts anderes, als ein und denselben Eigenschaft mehrere Werte zuzuweisen. Dazu muss nach der Deklaration der Annotation ein # gefolgt von einem Schlüssel angegeben werden. Dies kann in der SAPUI5-Anwendung abgefragt werden, wodurch die richtigen Werte angezeigt werden. In Listing 7.18 können Sie sehen, wie dies beispielweise aussehen könnte.

```
@Common.Label: 'BOOK'  
@Common.Label#Teacher: 'Book'
```

Listing 7.18 Qualifizierte Annotationen

Primitive Typen als Annotationen

Das SAP Cloud Application Programming Model kann auch primitive Annotationen verarbeiten. Wenn primitive Annotationswerte auftreten, werden sie auf die entsprechende OData-Annotation abgebildet. Die Annotationen in Listing 7.19 würden auf die Annotationen in Listing 7.20 gemappt werden. Beachten Sie, dass die Annotation Any nicht existiert, sie wird hier nur zur Veranschaulichung des Mappings verwendet.

```
@Any.Integer: 8.2
@Any.Boolean: false
@Any.String: 'test'
```

Listing 7.19 Primitive Annotationen im SAP Cloud Application Programming Model

```
<Annotation Term="Any.Integer" Int="8.2"/>
<Annotation Term="Any.Boolean" Bool="false"/>
<Annotation Term="Any.String" String="test"/>
```

Listing 7.20 Primitive Annotation in OData

Records als Annotationen

Strukturen, die einem Datensatz ähneln, werden den entsprechenden `<Record>`-Knoten in einer **EDMX**-Datei zugeordnet. Die primitiven Typen werden jedoch wie im vorherigen Abschnitt beschrieben abgebildet. In Listing 7.21 und Listing 7.22 sehen wir noch einmal anhand eines Beispiels, wie das Mapping des SAP Cloud Application Programming Model in OData aussieht.

```
@Any.Record: {
    Integer: 8.2,
    Boolean: false
    String: 'test'
}
```

Listing 7.21 Record-Annotation im SAP Cloud Application Programming Model

```
<Annotation Term="Any.Record">
<Record>
    <PropertyValue Property="Integer" Int="8.2"/>
    <PropertyValue Property="Bool" Int="false"/>
    <PropertyValue Property="String" Int="test"/>
</Record>
</Annotation>
```

Listing 7.22 Record-Annotation in OData

Collections als Annotationen

Zusätzlich zu primitiven Typen und Records können auch Collections in Annotations abgebildet werden. Das heißt, wenn ein Array angegeben wird, wird es auf einen entsprechenden Sammlungsknoten abgebildet. Die Regeln für Records und Collections werden rekursiv angewendet. Primitive Typen werden direkt als Kindknoten erstellt.

In Listing 7.23 und Listing 7.24 sehen wir wieder ein Beispiel dafür, wie ein solches Mapping aussehen könnte.

```
@Any.Collection: [  
    false, 8.2, 'test', { $Type: 'UI.DataField', Label: 'Test-Label' }  
]
```

Listing 7.23 Collection-Annotation im SAP Cloud Application Programming Model

```
<Annotation Term="Any.Collection">  
    <Collection>  
        <Bool>false</Bool>  
        <Int>8.2</Int>  
        <String>Test</String>  
        <Record Type="UI.DataField">  
            <PropertyValue Property="Label" String="Test-Label"/>  
        </Record>  
    </Collection>  
</Annotation>
```

Listing 7.24 Collection-Annotation in OData

Unterschiede im Vergleich zu den ABAP Core Data Services

Was die Annotationen betrifft, gibt es einige Unterschiede zwischen SAP Cloud Application Programming Model und ABAP CDS. Im SAP Cloud Application Programming Model wird ein generischer, isomorpher Ansatz verfolgt, was bedeutet, dass die Namen und Positionen der Annotationen identisch sind wie in den OData-Vokabularen angegeben. Daraus ergibt sich eine Reihe von Vorteilen:

- **Single Source of Truth**

Entwickler müssen keine andere Dokumentation als die offizielle OData-Dokumentation konsultieren.

- **Effizientere Geschwindigkeit**

Komplexe Mapping-Logik ist nicht erforderlich.

- **Eliminierung von Engpässen**

CDS kann in Entity Data Model XML (EDMX) übersetzt werden und umgekehrt.

- **Zeitsparend**

Das Schreiben von Derivaten des OData-Vokabulars ist nicht erforderlich.

7.2.3 Kundenspezifische Logik

Auch wenn die meisten Anforderungen mit den standardmäßig verfügbaren generischen Service-Providern umgesetzt werden können, gibt es auch Fälle, in denen auf manuelle Implementierungen, auch wenn sie mehr Aufwand bedeuten, nicht ganz verzichtet werden kann. In den meisten Fällen ist die Notwendigkeit solcher Implementierungen jedoch eher begrenzt. Zum Beispiel könnte eine solche Implementierung notwendig sein, wenn einer der folgenden Punkte zutrifft:

- Domänen spezifische programmatische Validierungen sind erforderlich.
- Ergebnismengen müssen erweitert werden, z. B. berechnete Felder, die zur Laufzeit berechnet werden.
- Eine programmatische Berechtigungsprüfung ist erforderlich.
- Der Aufruf bestimmter Folgeaktionen ist erforderlich, z. B. der Aufruf anderer Services.
- Es werden Logiken benötigt, die mit dem generischen Ansatz nicht implementiert werden können.

In den folgenden Abschnitten werden wir sehen, wie solche spezifischen Logiken aufgerufen oder implementiert werden können.

7.2.4 Kundenspezifische Implementierungen anbieten

Der einfachste Weg, die Implementierung von Services in Node.js zu definieren, besteht darin, eine JavaScript-Datei zu erstellen, in die später die Logik eingefügt wird. Es ist wichtig, dass die JavaScript-Datei denselben Namen hat wie die CDS-Datei, die die Servicedefinition enthält. Außerdem sollten sich beide Dateien im selben Verzeichnis befinden.

Diese Dateistruktur könnte z. B. so aussehen:

- `/srv`
 - `my-service.cds`
 - `my-service.js`

7.2.5 Event Handler Hooks

Es gibt verschiedene Hooks, für die Sie einen Event Handler registrieren können. Der Hook bestimmt im Grunde nur, zu welchem Zeitpunkt die Implementierung aufgerufen wird. Die folgenden Hooks sind verfügbar:

■ on

Ein on-Handler ersetzt die Standardimplementierung eines Services.

■ before

Der before-Handler wird vor der Standardimplementierung aufgerufen, oder, wenn er von on überschrieben wird, vor dem on-Handler.

■ after

Der after-Handler wird nach der Standardimplementierung aufgerufen, oder, wenn er von on überschrieben wird, nach dem On-Handler. Es kann immer nur eine gültige Implementierung des on-Handlers geben, die aufgerufen wird. Wenn mehrere before- oder after-Handler implementiert sind, werden sie alle parallel aufgerufen.

Event Handler registrieren

Nachdem wir nun die einzelnen Hooks kennengelernt haben, stellt sich natürlich die Frage, wie die Event Handler registriert werden können, damit die entsprechenden Logiken auch ausgeführt werden.

Dazu muss Code in der zuvor definierten JavaScript-Datei implementiert werden. Für jeden Hook wird hier eine Methode zur Verfügung gestellt, die aufgerufen werden kann. In dieser Methode müssen folgende Parameter übergeben werden:

- Name der Operation (CREATE, READ, UPDATE, DELETE)
- Name der Entität (z. B. Products)
- Funktion, die die Logik des Handlers beinhaltet

Für die ersten beiden Parameter (Operation, Entität) kann ein Sternchen * als Platzhalter angegeben werden. Dies würde dann »alle« bedeuten.

In Listing 7.25 sehen wir, wie der Inhalt der JavaScript-Datei aussehen könnte. Natürlich muss auch die Logik in der Funktion entsprechend implementiert werden.

```
const cds = require('@sap/cds');
module.exports = function() {
  this.on('CREATE', 'Product', (req) => {/* Custom Logic */})
  this.before('*', '*', (req) => {/* Custom Logic */})
  this.after('READ', 'Product', (req) => {/* Custom Logic */})
}
```

Listing 7.25 Kundenspezifischen Event Handler registrieren

Wie hier im Beispiel gezeigt, wird der Handler-Funktion ein Request-Message-Objekt übergeben. Dadurch ist es möglich, auf sehr wichtige Informationen in der Funktion zuzugreifen, darunter die folgenden:

- Ereignisname (kann ein Name der Methoden create, read, updated, delete [CRUD] oder ein benutzerdefinierter Name sein)
- Zielentität, falls vorhanden
- Abfrage für CRUD-Anfragen
- Daten-Payload
- der Benutzer, wenn die Anfrage authentifiziert ist
- der Mandant, der die Software-as-a-Service-Anwendung nutzt (nur wenn dies aktiviert ist)

7.3 Aufsetzen einer SAP-Cloud-Application-Programming-Model-Anwendung

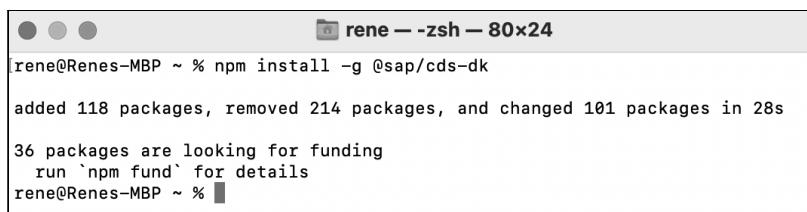
In diesem Abschnitt sehen wir uns nun an, wie wir eine Anwendung mit dem SAP Cloud Application Programming Model entwickeln können und welche Voraussetzungen dafür notwendig sind. Für das Datenmodell verwenden wir dasjenige, das bereits in Kapitel 6, »Entwicklung von OData-V4-Services mit dem ABAP RESTful Application Programming Model (Managed Scenario)«, mit dem ABAP RESTful Application Programming Model implementiert wurde.

7.3.1 Installieren der erforderlichen Software

Bevor Sie beginnen, sollten Node.js und Visual Studio Code installiert werden. Sobald dies geschehen ist, kann die Installation von cds-dk gestartet werden. Dabei handelt es sich um eine Kommandozeile, die Funktionen und Werkzeuge bereitstellt, die für die Entwicklung mit dem SAP Cloud Application Programming Model erforderlich sind.

Öffnen Sie dazu eine Kommandozeilenanwendung und führen Sie den folgenden Befehl aus (siehe Abbildung 7.1):

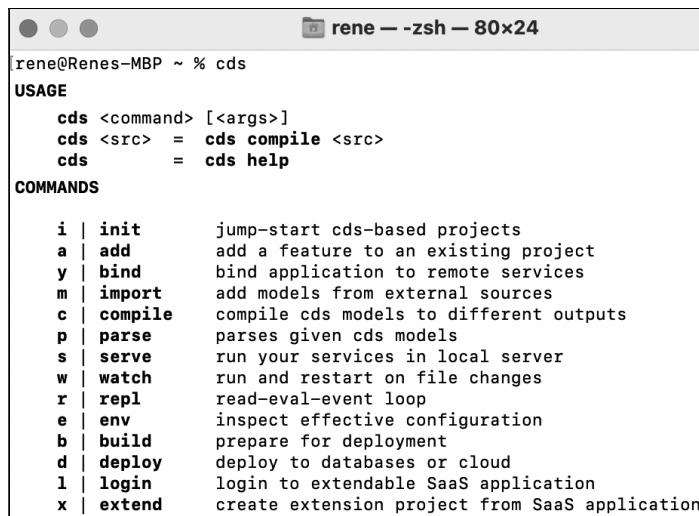
```
npm install -g @sap/cds-dk
```



```
rene -- -zsh -- 80x24
rene@Renes-MBP ~ % npm install -g @sap/cds-dk
added 118 packages, removed 214 packages, and changed 101 packages in 28s
36 packages are looking for funding
  run `npm fund` for details
rene@Renes-MBP ~ %
```

Abbildung 7.1 Tools für die Entwicklung mit dem SAP Cloud Application Programming Model installieren

Sobald die Installation abgeschlossen ist, können Sie die Installation durch Eingabe von »cds« und anschließende Bestätigung mit überprüfen. Wenn die Installation erfolgreich war, sollten Sie eine Ausgabe wie in Abbildung 7.2 sehen.



```

rene@Renes-MBP ~ % cds
USAGE
  cds <command> [<args>]
  cds <src>   = cds compile <src>
  cds        = cds help
COMMANDS
  i | init      jump-start cds-based projects
  a | add       add a feature to an existing project
  y | bind      bind application to remote services
  m | import    add models from external sources
  c | compile   compile cds models to different outputs
  p | parse     parses given cds models
  s | serve     run your services in local server
  w | watch     run and restart on file changes
  r | repl      read-eval-event loop
  e | env       inspect effective configuration
  b | build     prepare for deployment
  d | deploy    deploy to databases or cloud
  l | login     login to extendable SaaS application
  x | extend    create extension project from SaaS application

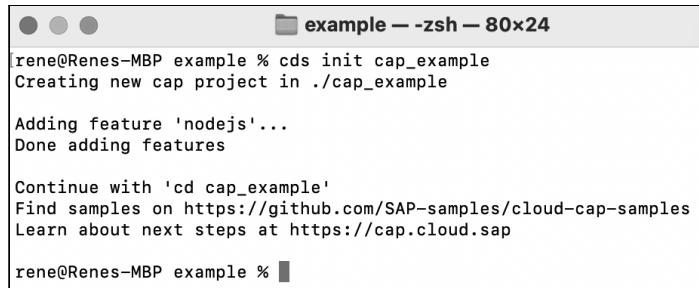
```

Abbildung 7.2 Verifizieren der Installation der Development Tools

7.3.2 Erstellen eines SAP-Cloud-Application-Programming-Model-Projekts

Nachdem nun die notwendige Software installiert wurde, können wir mit der Erstellung des SAP-Cloud-Application-Programming-Model-Projekts fortfahren. Dazu starten wir zunächst Visual Studio Code.

Um ein Projekt zu erstellen, navigieren Sie in der Befehlszeile zu einem beliebigen Verzeichnis, in dem die Anwendung erstellt werden soll. Führen Sie dann den Befehl `cds init` aus, gefolgt von dem Namen der Anwendung. In unserem Fall ist dies `cap_example`, also führen wir `cds init cap_example` aus (siehe Abbildung 7.3).



```

rene@Renes-MBP example % cds init cap_example
Creating new cap project in ./cap_example
Adding feature 'nodejs'...
Done adding features

Continue with 'cd cap_example'
Find samples on https://github.com/SAP-samples/cloud-cap-samples
Learn about next steps at https://cap.cloud.sap
rene@Renes-MBP example %

```

Abbildung 7.3 Erstellen eines SAP-Cloud-Application-Programming-Model-Projekts

Sobald dies geschehen ist, kann das Projekt in Visual Studio Code geöffnet werden. Führen Sie dann den Kommandozeilenbefehl `npm install` einmal im Stammverzeichnis des Projekts aus, um alle für das Projekt erforderlichen Dependencies zu installieren, damit das Projekt später gestartet werden kann.

Ist dies geschehen, kann die eigentliche Implementierung beginnen.

7.3.3 Erstellen des Entitätenmodells

Für unser Beispiel benötigen wir eine Entität `Books`, in der verschiedene Bücher verwaltet werden können. Um dies zu erreichen, gehen wir folgendermaßen vor: Zunächst muss im Verzeichnis `db` eine CDS-Datei namens `schema` angelegt werden, falls sie noch nicht existiert. Das Entitätenmodell wird dann in dieser Datei abgebildet.

Wie in Listing 7.26 dargestellt, hat unsere Entität eine UUID als Primärschlüssel sowie zahlreiche weitere Attribute. Darüber hinaus wird der Aspekt `managed` über Vererbung hinzugefügt. Dadurch werden weitere Felder wie letzte Änderung, Änderungsdatum, Ersteller und Erstellungsdatum hinzugefügt und automatisch ausgefüllt. Diese Metadaten können oft hilfreich sein, um nachzuvollziehen, wer etwas wann geändert hat.

```
using managed from '@sap/cds/common';
namespace at.cloudDNA;

entity Book : managed {
    key ID : UUID;
    isbn : String;
    title : String;
    description : String;
    genre : String;
    author : String;
    book_pages : Integer;
    purch_date : Date;
    finished : Boolean;
    rating : Integer;
}
```

Listing 7.26 Definition eines Schemas im SAP Cloud Application Programming Model

Nachdem das Datenmodell definiert und gespeichert wurde, starten Sie die Anwendung mit `cds watch`. Sie sollten im Terminal eine ähnliche Ausgabe wie in Abbildung 7.4 sehen.

Abbildung 7.4 Starten der SAP-Cloud-Application-Programming-Model-Anwendung

Der Standardport, auf dem die Anwendung gestartet wird, ist 4004, wie in der Konsoleausgabe zu sehen ist. Wenn Sie die Adresse `localhost:4004` aufrufen, können Sie nun auch auf die Anwendung zugreifen, aber Sie werden noch nicht viel sehen (siehe Abbildung 7.5), da eine wichtige Komponente noch fehlt: die Servicedefinition. Im nächsten Abschnitt werden Sie sehen, wie Sie diese Servicedefinition erstellen.

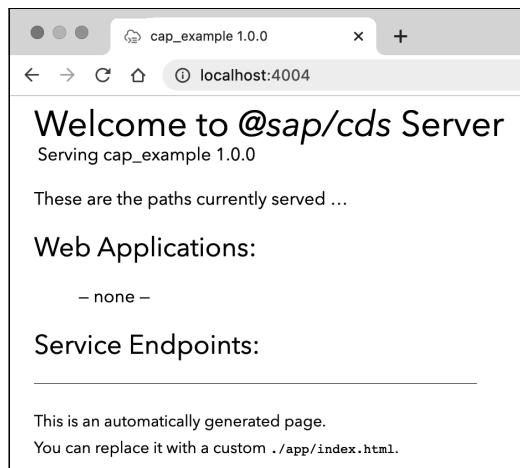


Abbildung 7.5 Einstiegsseite des gestarteten Service

7.3.4 Erstellen der Servicedefinition

Wenn Sie die Adresse öffnen, unter der die Anwendung läuft, werden Sie feststellen, dass derzeit keine Servicedefinitionen vorhanden sind. Dies wird auch in der Konsole

durch eine Ausgabe angezeigt. Um hier Abhilfe zu schaffen, werden wir in den nächsten Schritten eine Servicedefinition erstellen, die unseren Service veröffentlichen wird.

Die Servicedefinition stellt im Grunde eine Schnittstelle zwischen den Domänenmodellen und dem Endbenutzer her. Das bedeutet, dass der Benutzer über den Service mithilfe definierter Methoden und Funktionen auf die zugrunde liegenden Daten zugreifen kann. Standardmäßig wird ein OData-4.0-Service veröffentlicht. Es gibt verschiedene Möglichkeiten, wie und ob einzelne Entitäten überhaupt publiziert werden. Dies kann bei der Definition des Service festgelegt werden.

Die SAP-Cloud-Application-Programming-Model-Laufzeit übernimmt bereits einen Teil der Logik und bietet damit zahlreiche Funktionen out of the box, z. B.:

- alle CRUD-Operationen
- Deep Inserts
- Deep Updates
- Deep Deletes
- autogenerierte Primärschlüssel
- Filter- und Sortieroperationen
- Paginierung

Im SAP Cloud Application Programming Model können auch mehrere Services innerhalb eines Projekts erstellt werden. Dies hat keine negativen Auswirkungen und wird sogar empfohlen.

Generell sollte darauf geachtet werden, dass nicht ein Service definiert wird, der alle Entitäten enthält. Stattdessen sollten die Services nach Anwendungsfällen aufgeteilt werden. Darüber hinaus sollte eine 1:1-Abbildung von Services möglichst vermieden werden.

Es mag auf den ersten Blick einfacher und schneller erscheinen, einen Service zu definieren, der alle Domänenmodelle abbildet, aber man stößt dann später auf mögliche Probleme:

- Große Services, die alles enthalten, können den Nutzern Zugangspunkte bieten, die unbeabsichtigt mit wenigen Einschränkungen genutzt werden können.
- Spezifische Anwendungsfälle werden in der API möglicherweise nicht berücksichtigt.
- Zahlreiche Prüfungen, wie z. B. Berechtigungsprüfungen, müssen zu mehreren Anfragen hinzugefügt werden.

Stattdessen sollten die Services nach einzelnen Anwendungsfällen getrennt werden. Dies könnten z. B. Services sein, die nach Rolle oder Berechtigung aufgerufen wer-

den. Auf diese Weise bekämen Administratoren eigene Services mit umfangreicheren Funktionen als Endanwender. Auch wenn die gleichen Entitäten dahinter stehen, kann der Funktionsumfang unterschiedlich sein.

In unserem Fall reicht es aus, einen einzigen Service zu erstellen. Die notwendigen Schritte und wie der Service dann genutzt werden kann, wird in diesem Abschnitt gezeigt.

Um eine Servicedefinition zu erstellen, die unsere Entität zugänglich macht, muss zunächst eine CDS-Datei im Verzeichnis **srv/** angelegt werden, sofern dies noch nicht geschehen ist. Der Name dieser Datei kann frei gewählt werden, aber es ist ratsam, einen beschreibenden Namen zu verwenden, der Auskunft darüber gibt, wofür dieser Service zuständig ist.

In unserem Fall benennen wir die Datei **book-service.cds**, indem Sie die folgenden Schritte ausführen:

1. Klicken Sie mit der rechten Maustaste auf das Verzeichnis **srv**.
2. Wählen Sie **New File** (siehe Abbildung 7.6).

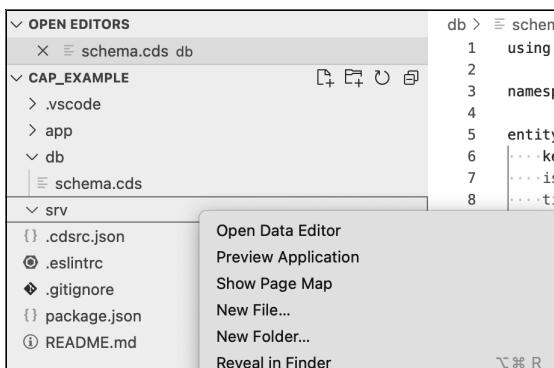


Abbildung 7.6 Erstellen einer Servicedefinition

3. Weisen Sie den Namen mit der entsprechenden **.cds**-Dateierweiterung zu.

Danach sollte die Verzeichnisstruktur wie in Abbildung 7.7 aussehen.

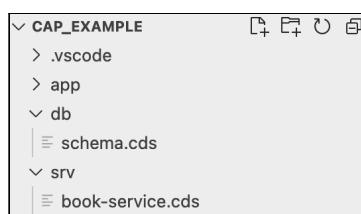


Abbildung 7.7 Projektstruktur nach Anlage der Servicedefinition

Nun öffnen wir zunächst die neu erstellte Datei und implementieren darin den eigentlichen Service. Die vereinfachte Syntax hierfür lautet wie folgt:

```
service <servicename> { /* Definitions */ }
```

Optional können einzelne Services auch mit Annotationen versehen werden, z. B. um Services nur für bestimmte Rollen oder Benutzer verfügbar zu machen. In unserem Fall kann der Service ohne Einschränkung genutzt werden.

Zunächst fügen wir in der Servicedefinition mit `using` einen Import ein, der das Datenmodellschema importiert. Dieses ist in der Datei **schema.cds** definiert. Mit dem Schlüsselwort `as` wird ein Alias definiert, damit wir nicht immer den vollen Namensraum ausschreiben müssen. Dieser Alias kann beliebig gewählt werden, etwa wie folgt:

```
using { at.clouddna as my } from '../db/schema';
```

Anschließend kann der Service definiert werden. Optional können wir einen Pfad angeben, unter dem der Service aufgerufen werden kann, was vor allem dann notwendig ist, wenn innerhalb eines Projekts verschiedene Services definiert sind. Wir verwenden `/bookservice` als Pfad. In unserem Fall sieht die Definition des Service wie folgt aus:

```
service BookService @path: '/bookservice' {}
```

Wenn wir nun die Anwendung erneut starten und die Seite `localhost:4004` aufrufen, sehen wir, dass die Servicedefinition bereits erkannt wird (siehe Abbildung 7.8). Es fehlen jedoch noch die entsprechenden Entitäten.

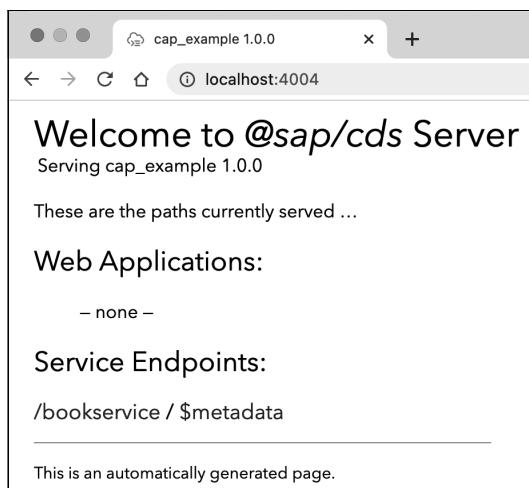


Abbildung 7.8 Veröffentlichte Servicedefinition

Im nächsten und letzten Schritt müssen nun noch die entsprechenden Entitäten im Service veröffentlicht werden. In unserem Fall ist dies nur die Entität Book. Diese veröffentlichen wir als Service-Entität mit dem gleichen Namen. Dazu gehen wir zurück zur Servicedefinition und fügen die Projektion auf die Entität an der entsprechenden Stelle ein. Die Syntax hierfür lautet wie folgt:

```
entity <service_entity_name> as projection on <entity_name>;
```

Durch die Klausel `as projection on` wird die gesamte Entität, wie sie im Datenmodell definiert ist, als Ganzes im Service veröffentlicht. Wenn wir hier bestimmte Einschränkungen erreichen wollen, wie z. B. das explizite Weglassen bestimmter Felder, kann auch eine andere Syntax verwendet werden. Dies werden wir im Folgenden sehen:

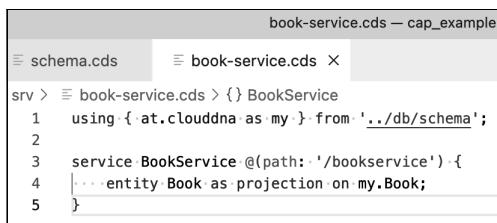
```
entity <service_entity_name> as SELECT from <entity_name> { * } excluding { <field1, ...> }
```

Anstelle der `as projection on`-Klausel kann auch eine `SELECT`-Anweisung angegeben werden. Außerdem kann eine `excluding`-Klausel verwendet werden, um anzugeben, dass bestimmte Felder nicht gelesen werden sollen.

Für unseren Fall ist die vereinfachte Syntax ausreichend. In der Datei `service.cds` fügen wir dies entsprechend ein. Wir wählen Book als Namen für die Entität in unserem Service. Die Definition sollte wie folgt aussehen:

```
entity Book as projection on my.Book;
```

Nachdem diese Definition eingefügt wurde, sollte unsere CDS-Datei wie in Abbildung 7.9 aussehen.



```
book-service.cds — cap_example
schema.cds book-service.cds

srv > book-service.cds > {} BookService
1 using { at.clouddna as my } from '../db/schema';
2
3 service BookService @path: '/bookservice' -{
4 |...entity Book as projection on my.Book;
5 }
```

Abbildung 7.9 Servicedefinition mit der veröffentlichten Book-Entität

Wenn die Anwendung nun neu gestartet wird, sollte die veröffentlichte Entität auch in ihr verfügbar sein. Dazu öffnen wir die Anwendung erneut in einem beliebigen Browser. Wie in Abbildung 7.10 zu sehen ist, ist die entsprechende Entität, die mittels einer Servicedefinition veröffentlicht wurde, nun hier sichtbar.

Außerdem wird die veröffentlichte Entität auch in den Metadaten wiedergegeben, wie in Abbildung 7.11 zu sehen ist. Diese können unter der folgenden Adresse abge-

fragt werden: `localhost:4004/<service_name>/$metadata`. In unserem Fall ist es `localhost:4004/bookservice/$metadata`.

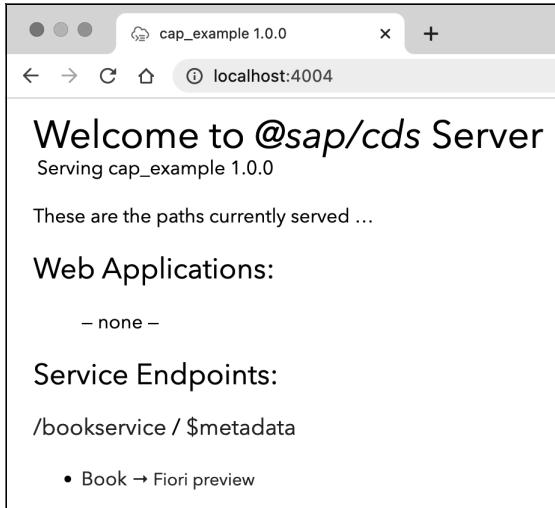


Abbildung 7.10 Veröffentlichte Entität in laufender Applikation

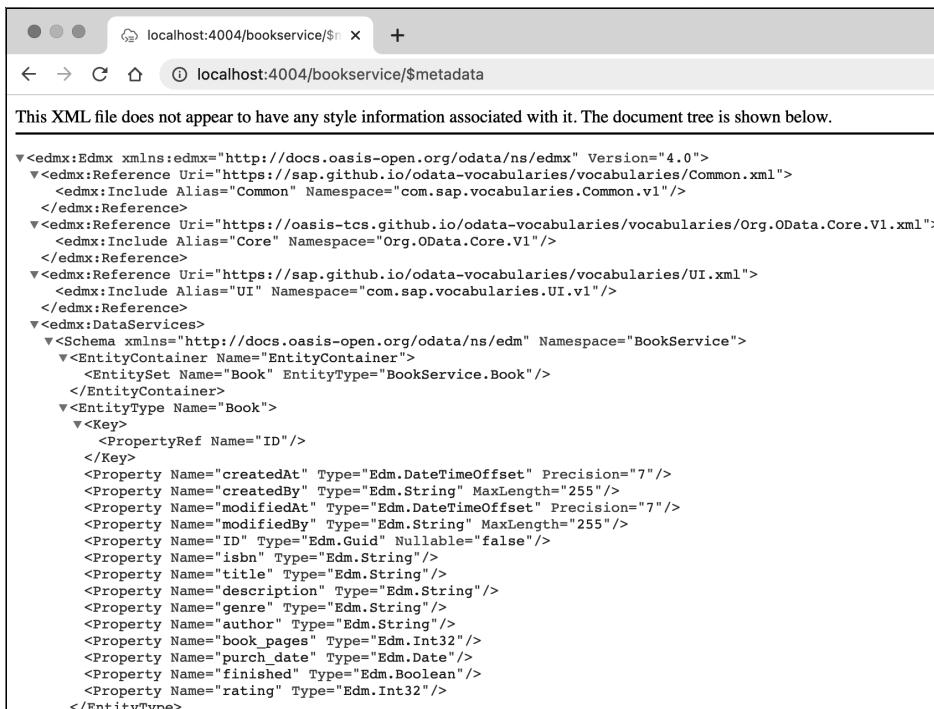


Abbildung 7.11 Metadaten des zuvor angelegten Service

7.3.5 Bereitstellen von Daten

Nachdem wir nun alle notwendigen Vorehrungen für die Nutzung des Service getroffen haben, geht es jetzt darum, die entsprechenden Daten in den Service zu bekommen. Dies kann mithilfe der CRUD-Funktionen über die OData-Schnittstelle geschehen. Wird die Anwendung lokal gestartet, befindet sich dahinter nur eine In-Memory-Datenbank. Das bedeutet, dass die Daten bei jedem Neustart verloren gehen.

Um dies zu vereinfachen, können Daten aus **.csv**-Dateien importiert werden, die dann bei jedem Neustart der Anwendung zur Verfügung stehen. Obwohl dies kein Anwendungsfall für produktive Anwendungen ist, eignet sich diese Art der Datenbereitstellung gut zum Testen der Anwendung.

Um Daten automatisch zu importieren, sind nur wenige Schritte erforderlich. Zunächst muss innerhalb des **db**-Verzeichnisses ein Unterverzeichnis namens **data** angelegt werden. In diesem Verzeichnis können später die gewünschten **.csv**-Dateien abgelegt werden. Folgen Sie diesen Schritten:

1. Klicken Sie mit der rechten Maustaste auf **db**, wählen Sie **New Folder** und erstellen Sie einen neuen Ordner mit dem Namen »data« (siehe Abbildung 7.12).

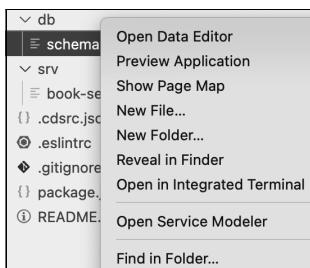


Abbildung 7.12 Erstellen eines neuen Ordners für die ».csv«-Dateien

Nachdem dieses Verzeichnis erstellt wurde, sollte die Ordnerstruktur in etwa wie in Abbildung 7.13 aussehen.

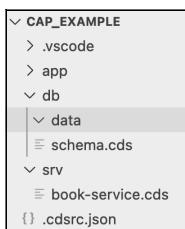


Abbildung 7.13 Ordnerstruktur nach Anlage des Ordners für die ».csv«-Dateien

Anschließend müssen die entsprechenden **.csv**-Dateien in dem neu erstellten Verzeichnis angelegt werden. Dabei gilt die folgende Namenskonvention: **<Name>**

space>-<Entitätsname>.csv. Es ist wichtig, dass für jede gewünschte Entität eine eigene Datei erstellt wird. In unserem Fall muss die Datei also wie folgt lauten: **at.clouddna-Book.csv**.

- Um diese Datei zu erstellen, klicken Sie mit der rechten Maustaste auf das Verzeichnis **data** (siehe Abbildung 7.14). Anschließend muss mit **New File** eine neue **.csv**-Datei mit dem zuvor angegebenen Namen erstellt werden.

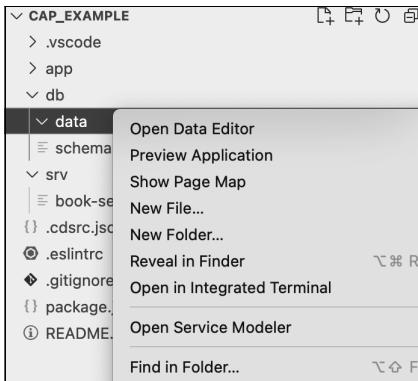


Abbildung 7.14 Erstellen einer ».csv«-Datei für Beispieldaten

Nachdem diese Datei erstellt wurde, sollte die Projektstruktur wie in Abbildung 7.15 aussehen. Der Namensraum und der Entitätsname im Dateinamen sind hier wichtig.

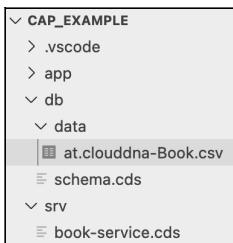


Abbildung 7.15 Projektstruktur nach Anlage der ».csv«-Datei

Nachdem die Datei erstellt wurde, können die Beispieldaten in der Datei gepflegt werden. Die Eigenschaftnamen müssen als Kopfzeile definiert werden. Es müssen jedoch nicht alle vorhandenen Attribute angegeben werden, da fehlende mit den jeweiligen Standardwerten des dahinter stehenden Datentyps aufgefüllt werden.

Außerdem kann in unserem Fall die ID weggelassen werden, da wir eine UUID verwenden, die automatisch generiert wird. In Abbildung 7.16 sehen Sie, wie eine Beispieldatendefinition aussehen könnte. In unserem Fall werden wir nur die folgenden Spalten füllen: `isbn`, `title`, `description`, `author` und `book_pages`.

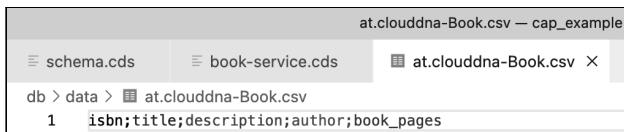


Abbildung 7.16 Definition der Header in der ».csv«-Datei

- Alle nachfolgenden Zeilen enthalten die Daten, die in die Entität eingefügt werden sollen. In unserem Fall fügen wir zwei Beispieldatensätze ein, wie in Abbildung 7.17 dargestellt. Achten Sie bei der Definition der Beispieldaten darauf, dass die Datentypen der Werte mit den jeweiligen Spalten übereinstimmen, da sonst ein Fehler auftritt.

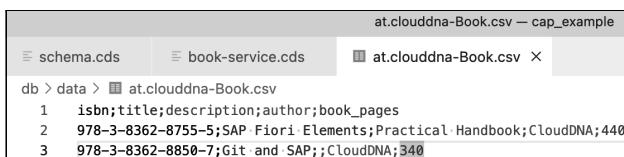


Abbildung 7.17 Definition der ».csv«-Datei mit Beispieldaten

Nachdem die Beispieldaten gepflegt worden sind, kann die Anwendung neu gestartet werden. In der Konsole sollte angezeigt werden, dass die Daten aus der .csv-Datei geladen werden.

- Dazu starten Sie die Anwendung erneut mit `cds watch`. In der Konsole sollte eine Ausgabe sichtbar werden, dass die Daten aus der .csv-Datei geladen wurden. Dies ist in der Regel `init from db/data/<Name_der_csv-Datei>`. Ein solches Beispiel sehen Sie in Abbildung 7.18.

```

PROBLEMS TERMINAL OUTPUT DEBUG CONSOLE SQL CONSOLE
reno@Renes-MBP cap_example % cds watch
cds serve all --with-mocks --in-memory?
live reload enabled for browsers

[cds] - loaded model from 3 file(s):
db/schema.cds
srv/book-service.cds
../../../../usr/local/lib/node_modules/@sap/cds-dk/node_modules/@sap/cds/common.cds

[cds] - connect using bindings from: { registry: '~/.cds-services.json' }
[cds] - connect to db > sqlite { database: ':memory:' }
> init from db/data/at.clouddna-Book.csv
/> successfully deployed to in-memory database.

[cds] - serving BookService { path: '/bookservice' }

[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at 4/11/2023, 5:38:55 PM, version: 6.7.0, in: 1.199s
[cds] - [ terminate with ^C ]

```

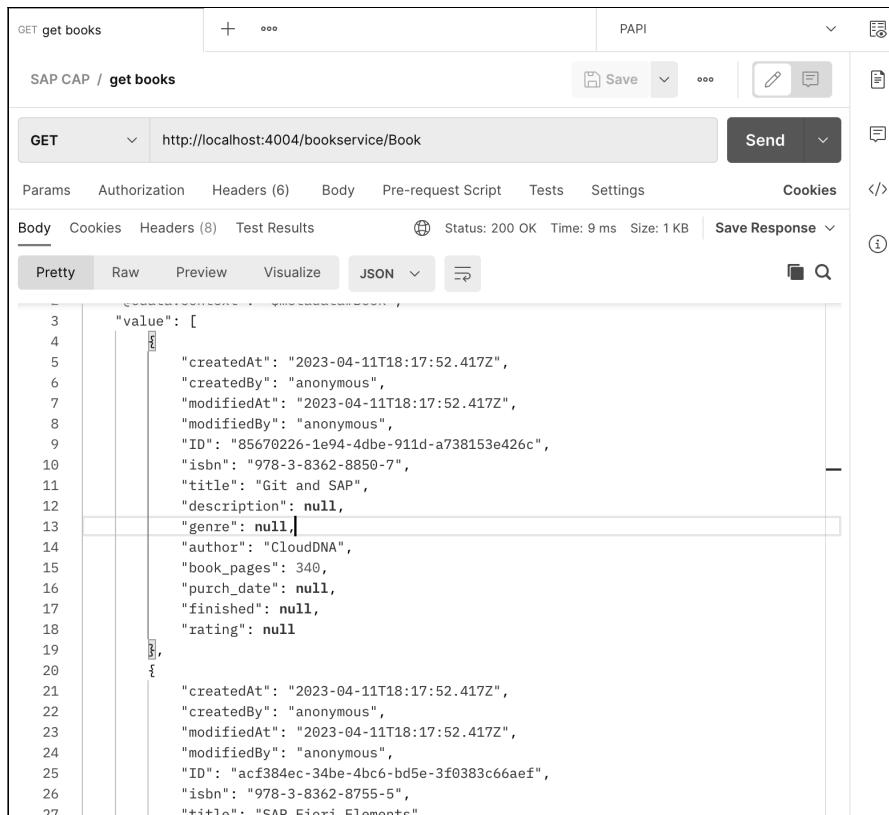
Abbildung 7.18 Konsolenausgabe nach erfolgreichem Laden der Beispieldaten

7.3.6 Anzeige von Daten

Um tatsächlich zu überprüfen, ob die Daten geladen wurden, können Sie den OData-Service oder die SAP-Fiori-Vorschau aufrufen. Im Folgenden werden wir uns beide Möglichkeiten ansehen.

Daten über den OData-Service anzeigen

OData-Services werden in der Regel in SAPUI5-Anwendungen verwendet, um Daten aus dem Backend abzurufen oder Daten an das Backend zu übermitteln. Alles was wir brauchen, ist der Aufruf des entsprechenden Entity-Sets. In unserem Fall kann dieses über die URL <http://localhost:4004/bookservice/Book> mithilfe eines REST-Clients abgefragt werden. Wie in Abbildung 7.19 zu sehen ist, werden die beiden Einträge, die wir über die .csv-Datei importiert haben, korrekt zurückgegeben.



```

3   "value": [
4     {
5       "createdAt": "2023-04-11T18:17:52.417Z",
6       "createdBy": "anonymous",
7       "modifiedAt": "2023-04-11T18:17:52.417Z",
8       "modifiedBy": "anonymous",
9       "ID": "85670226-1e94-4dbe-911d-a738153e426c",
10      "isbn": "978-3-8362-8850-7",
11      "title": "Git and SAP",
12      "description": null,
13      "genre": null,
14      "author": "CloudDNA",
15      "book_pages": 340,
16      "purch_date": null,
17      "finished": null,
18      "rating": null
19    },
20    {
21      "createdAt": "2023-04-11T18:17:52.417Z",
22      "createdBy": "anonymous",
23      "modifiedAt": "2023-04-11T18:17:52.417Z",
24      "modifiedBy": "anonymous",
25      "ID": "acf384ec-34be-4bc6-bd5e-3f0383c66aef",
26      "isbn": "978-3-8362-8755-5",
27      "title": "SAP Fiori Elemente"
    }
  ]

```

Abbildung 7.19 Abfrage von Daten über den OData-Service

Daten mittels der SAP-Fiori-Vorschau anzeigen

Eine weitere Möglichkeit, die Daten aus den jeweiligen Diensten anzuzeigen, bietet die SAP-Fiori-Vorschau. Diese wird von SAP Cloud Application Programming Model

standardmäßig bereitgestellt. Es handelt sich im Grunde um eine tabellarische Darstellung der Service-Entität. Hier können auch Sortierungen und Filter angewendet werden.

Um diese Vorschau zu öffnen, navigieren Sie zunächst zur Root-Adresse der SAP-Cloud-Application-Programming-Model-Anwendung, d. h. zu <http://localhost:4004>. Die verfügbaren Service-Endpunkte werden hier angezeigt, mit dem Link **Fiori preview** neben jedem Service-Endpunkt (siehe Abbildung 7.20).

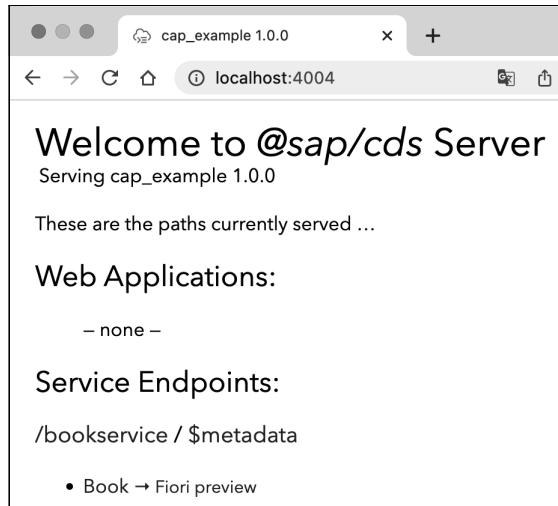


Abbildung 7.20 SAP-Fiori-Preview-Link für das Entity-Set

Wenn Sie auf den Link klicken, können Sie dorthin springen. Hier werden zunächst keine Spalten angezeigt, da diese erst eingeblendet werden müssen. Wenn die gewünschten Spalten zum ersten Mal angezeigt werden, werden auch die vorhandenen Datensätze angezeigt (siehe Abbildung 7.21).

Angelegt am	Angelegt von	Löschen	85670226-1e94-4dbe-911d-a738153e426c	978-3-8362-8850-7
11.04.2023, 20:17:52	anonymous	CloudDNA	acf384ec-34be-4bc6-bd5e-3f0383c66aef	978-3-8362-8755-5
11.04.2023, 20:17:52	anonymous	CloudDNA		

Abbildung 7.21 SAP Fiori Preview der Books

7.4 Autorisierungsprüfungen

Autorisierung bedeutet, dass der Zugriff auf Daten entsprechend den Berechtigungen des Benutzers eingeschränkt wird. Im SAP Cloud Application Programming Model kann dies über Annotationen gesteuert werden, die in der Servicedefinition angewendet werden. Standardmäßig gibt es keine Einschränkung, d. h., jeder kann alles sehen.

Mit der entsprechenden Annotationen kann festgelegt werden, dass nur Benutzer mit der entsprechenden Rolle oder dem entsprechenden Scope auf den Service zugreifen dürfen. Dafür stehen die folgenden Annotationen zur Verfügung:

- `@requires`
- `@restrict`

7.4.1 `@requires`

Mit der `@requires`-Annotation kann festgelegt werden, dass nur Benutzer mit bestimmten Rollen auf den Service zugreifen dürfen. Diese Annotation wird in der Regel direkt auf dem Service angegeben. Es folgt ein Anwendungsbeispiel für diese Annotation:

```
annotate Bookservice with @({requires : 'Enduser'});
```

Die vorstehende Anweisung würde dazu führen, dass nur Benutzer mit der Rolle **Enduser** auf den Service zugreifen dürfen.

Metadaten veröffentlichen

Wenn Sie einen ganzen Dienst mit der `@requires`-Annotation einschränken, werden auch die Metadaten automatisch durch diese Annotation eingeschränkt. Dies ist nicht immer erwünscht, da die Metadaten oft unabhängig geladen werden können sollten.

Um dies zu verhindern, können Sie die Konfigurationseigenschaft `cds.env.odata.protectMetadata` auf `false` setzen, um die Metadaten unabhängig von der `@requires`-Anmerkung zu veröffentlichen.



7.4.2 `@restrict`

Neben der `@requires`-Annotation ist auch die `@restrict`-Annotation verfügbar. Diese Annotation ermöglicht es, granulare Berechtigungen zu definieren. Grundsätzlich können damit alle Arten von Einschränkungen ausgedrückt werden. Diese Annota-

tion wird normalerweise auf den einzelnen Entitäten in der Servicedefinition angegeben. Die Syntax dieser Regeln lautet:

```
{ grant: <event>, to: <roles>, where: <filter_condition> }
```

Die einzelnen Attribute lauten wie folgt:

■ **grant**

Hier können ein oder mehrere Events angegeben werden, für die die Einschränkung wirksam wird. Mögliche Werte für diese Eigenschaft sind READ, CREATE, UPDATE und DELETE. Außerdem kann das virtuelle Element WRITE angegeben werden, das grundsätzlich alle Schreiboperationen abdeckt. Wenn Sie alle Ereignisse abfangen wollen, können Sie den Platzhalter * verwenden.

■ **to (optional)**

Hier können eine oder mehrere Rollen angegeben werden, für die die Einschränkung wirksam wird. Welche Rollen hier angegeben werden, bleibt grundsätzlich dem Entwickler überlassen. Wenn keine Rolle angegeben wird, wird der Standardwert any verwendet.

■ **where (optional)**

Hier kann eine Filterbedingung angegeben werden, die die Ergebnismenge einschränkt. Diese Klausel kann einen booleschen Ausdruck enthalten. Wenn Sie dynamische Berechtigungen auf die Ergebnismenge anwenden wollen, ist diese Klausel die richtige Wahl.

Eine definierte Berechtigung ist nur dann erfüllt, wenn alle vorhandenen Klauseln tatsächlich erfüllt sind. Angenommen, wir haben ein Beispiel wie in Listing 7.27. Hier soll die Entität Book nur von Benutzern mit der Rolle Admin gelesen werden können. Außerdem sollen nur Einträge zurückgegeben werden, bei denen die Eigenschaft valid den Wert true enthält.

```
entity Book @restrict: [grant: 'READ', to: 'Admin', where: 'valid = true']) {  
    /* ... */  
}
```

Listing 7.27 Beispiel einer beschränkten Entität

Solche Rechte können natürlich auch erweitert werden, z. B. könnte man mehrere Rollen angeben. In diesem Fall würde es jedoch ausreichen, wenn der Benutzer nur eine der definierten Rollen hat (d. h., diese sind miteinander verknüpft).

7.5 Zusammenfassung

In diesem Kapitel haben wir einen ausführlichen Blick auf das SAP Cloud Application Programming Model geworfen. Beginnend mit den Grundlagen, wie man eine Entität definiert, welche Typen verfügbar sind und wie man Primär- und Fremdschlüssel versteht und verwendet, haben wir in den folgenden Abschnitten auch fortgeschrittenere Themen erforscht. Wir haben uns auch angeschaut, wie Übersetzungstabellen implementiert und sofort verwendet werden können. Anschließend haben wir Möglichkeiten zur Validierung von Benutzereingaben erörtert, bevor sie in die Datenbank geschrieben werden.

Weitere wichtige Punkte sind die Integration von benutzerdefinierter Logik und die Implementierung von verschiedenen Berechtigungsprüfungen. Außerdem haben wir anhand eines praktischen Beispiels einen einfachen OData-Dienst mit dem SAP Cloud Application Programming Model installiert und implementiert. Dieser Service könnte natürlich um weitere Funktionalitäten, wie z. B. Berechtigungsprüfungen oder Eingabevalidierungen, erweitert werden.

Zusammenfassend lässt sich sagen, dass es relativ einfach ist, einen OData-Service mit dem SAP Cloud Application Programming Model einzurichten. Die Anwendung ist schnell fertig, man hat früh einen testbaren Status und die Anwendung bleibt schlank.

Nachdem wir uns in diesem und dem vorangegangenen Kapitel mit Technologien beschäftigt haben, die die Entwicklung von OData-Services ermöglichen, werden wir uns in den nächsten Kapiteln mit der Implementierung entsprechender Benutzeroberflächen mit SAPUI5 beschäftigen.

Inhalt

Einleitung	17
------------------	----

TEIL I Einführung

1 Einführung in die Fullstack-Entwicklung 27

1.1 Was ist ein Fullstack-Entwickler?	28
1.2 SAP-Programmiermodelle	33
1.2.1 ABAP-Programmiermodell für SAP Fiori	34
1.2.2 ABAP RESTful Application Programming Model	37
1.2.3 SAP Cloud Application Programming Model	45
1.3 HTTP-Grundlagen	52
1.4 Zusammenfassung	56

2 Einführung in SAPUI5 und SAP Fiori 57

2.1 SAPUI5 versus SAP Fiori	57
2.2 SAP Fiori Launchpad	67
2.3 SAP Fiori Design Guidelines	73
2.4 Entwicklungswerzeuge	74
2.4.1 SAP Business Application Studio	75
2.4.2 Visual Studio Code	79
2.5 Zusammenfassung	81

3 Einführung in OData 83

3.1 Einführung	84
3.1.1 Grundlagen von OData	85
3.1.2 CRUDQ-Methoden	89
3.1.3 OData-Query-Optionen	97

3.2	OData-Services entwickeln	102
3.2.1	SAP-Gateway-Service-Builder-Projekte	103
3.2.2	ABAP RESTful Application Programming Model	111
3.3	OData testen	116
3.3.1	Teststrategie	117
3.3.2	Testen von OData-Services mit SAP Gateway Client	120
3.3.3	Testen von OData-Services mit dem Postman-REST-Client	125
3.4	Zusammenfassung	131

4 Einführung in CDS

4.1	Entwicklung von Core Data Services	134
4.1.1	Einfache CDS-Views	135
4.1.2	Joins und Projektionen	139
4.1.3	Berechnete Felder	141
4.1.4	Aggregationsfunktionen	143
4.1.5	Kaskadierende Views	145
4.2	Assoziationen und Navigationseigenschaften	146
4.3	CDS-View-Erweiterung	150
4.4	Berechtigungen	153
4.5	Zusammenfassung	157

5 Einführung in TypeScript

5.1	TypeScript-Grundlagen	159
5.1.1	JavaScript	160
5.1.2	TypeScript	161
5.1.3	Zweck von TypeScript	162
5.1.4	Verwendung von TypeScript	162
5.1.5	Datentypen	165
5.2	TypeScript der nächsten Generation	165
5.2.1	Anonyme Funktionen	166
5.2.2	Arrow Functions	166
5.2.3	Rest-Parameter	167
5.2.4	Funktionsüberladung	168

5.2.5	Statische Mitglieder	168
5.2.6	Intersection- und Union-Typen	169
5.2.7	Variablen Deklaration	170
5.2.8	Vererbung	171
5.2.9	Abstrakte Klassen	173
5.3	Klassen und Interfaces	174
5.3.1	Klassen	174
5.3.2	Interfaces	177
5.4	Generics	182
5.4.1	Beispiel für Generics	182
5.4.2	Methoden von generischen Typen	184
5.4.3	Generic Constraints	184
5.4.4	Generische Interfaces	185
5.4.5	Generische Klassen	186
5.5	Modul Entwicklung	188
5.5.1	Was sind Module?	188
5.5.2	Warum Module verwenden?	188
5.5.3	Export-Module	189
5.5.4	Import-Module	189
5.6	Zusammenfassung	190

TEIL II Entwicklung

6 Entwicklung von OData-V4-Services mit dem ABAP RESTful Application Programming Model (Managed Scenario)

193

6.1	Entwicklung von Business-Objekten	194
6.1.1	Datenmodell	196
6.1.2	Core Data Services: Verhaltensdefinition	203
6.1.3	Projection-Views	205
6.1.4	Servicedefinition und Service-Binding	208
6.1.5	Draft Handling	215
6.2	Aktionen	224
6.3	Erweiterte Funktionen	228
6.3.1	Wertehilfen	229

6.3.2	Validierungen	231
6.3.3	Determinationen	233
6.3.4	Feature Control	235
6.4	Berechtigungsprüfung	237
6.4.1	CDS-Zugriffskontrollen	238
6.4.2	Implementierung der Authorization-Methode	239
6.5	Verschachtelte Business-Objekte	240
6.6	Zusammenfassung	242

7 Entwicklung von OData-V4-Services mit dem SAP Cloud Application Programming Model

7.1	Domänenmodellierung	244
7.1.1	Namenskonventionen	245
7.1.2	Entitäten, Typen und Schlüssel	247
7.1.3	Kompositionen und Assoziationen	253
7.1.4	Lokalisierte Daten	257
7.2	Service-Bereitstellung	259
7.2.1	Input-Validierung	259
7.2.2	OData-Annotationen	261
7.2.3	Kundenspezifische Logik	265
7.2.4	Kundenspezifische Implementierungen anbieten	265
7.2.5	Event Handler Hooks	265
7.3	Aufsetzen einer SAP-Cloud-Application-Programming-Model-Anwendung	267
7.3.1	Installieren der erforderlichen Software	267
7.3.2	Erstellen eines SAP-Cloud-Application-Programming-Model-Projekts	268
7.3.3	Erstellen des Entitätenmodells	269
7.3.4	Erstellen der Servicedefinition	270
7.3.5	Bereitstellen von Daten	276
7.3.6	Anzeige von Daten	279
7.4	Autorisierungsprüfungen	281
7.4.1	@requires	281
7.4.2	@restrict	281
7.5	Zusammenfassung	283

8 Entwicklung von OData-V2-Services mit dem SAP Gateway Service Builder	285
8.1 Servicemodellierung in der Transaktion SEGW	286
8.1.1 SEGW-Projekt erstellen	288
8.1.2 Datenmodellierung vornehmen	290
8.1.3 OData-Service registrieren	295
8.2 Create, Read, Update, Delete und Query	299
8.2.1 Read/Query	299
8.2.2 Create	310
8.2.3 Update	314
8.2.4 Delete	318
8.2.5 Assoziation implementieren	321
8.3 SADL-basierte OData-Services	328
8.3.1 Data Source Mapping	328
8.3.2 CDS-View-Entitäten als Datenquellen direkt referenzieren	334
8.4 Zusammenfassung	336
9 Entwicklung von SAPUI5-Freestyle-Anwendungen	337
9.1 SAPUI5 (Basic-)Projekt anlegen	339
9.1.1 Freestyle-Anwendung im SAP Business Application Studio anlegen	339
9.1.2 SAP Fiori Application Generator in Visual Studio Code	347
9.2 View aufbauen und UI-Komponenten verwenden	350
9.3 Applikationslogik, Formatter und Event Handler implementieren	362
9.4 Datenbindung	367
9.4.1 Binding Mode bei Datenbindung	370
9.4.2 Syntax bei Datenbindung	371
9.4.3 Arten von Datenbindung	372
9.5 Navigation und Routing	386
9.5.1 Hard-Coded Routing	388
9.5.2 Routing mit optionalen und Pflichtparametern	391
9.5.3 Navigation in Kombination mit Element Binding	394
9.6 Mehrsprachigkeit/Internationalization (i18n)	396

9.7 Deployment	398
9.8 Zusammenfassung	403

10 Entwicklung von SAPUI5-Freestyle-Anwendungen mit OData-V4-Services

10.1 Projekt anlegen	407
10.2 OData-Model-V4	412
10.2.1 OData-Model instanzieren	412
10.2.2 CRUDQ-Methoden	419
10.2.3 Datenbindung	423
10.2.4 Komfortfunktionen	423
10.3 Startseite implementieren	425
10.4 Navigation einbauen	432
10.5 Detailseite implementieren	434
10.6 Fragmente	435
10.6.1 Detailseite vorbereiten	436
10.6.2 Fragmente direkt einbetten	438
10.6.3 Fragmente aus dem Controller laden	440
10.7 SAPUI5-Projekte von einem Git Repository klonen	447
10.8 Zusammenfassung	450

11 Entwicklung von SAPUI5-Freestyle-Anwendungen mit OData-V2-Services

11.1 Projekt anlegen	455
11.2 OData-Model-V2	460
11.2.1 OData-Model instanzieren	460
11.2.2 CRUDQ-Methoden	467
11.2.3 Datenbindung	470
11.2.4 Komfortfunktionen	471
11.3 Startseite implementieren	472
11.3.1 Tabelle manuell aufbauen	473

11.3.2	Filterung und Sortierung	478
11.3.3	Smart Table verwenden	479
11.4	Navigation einbauen	481
11.5	Detailseite implementieren	484
11.5.1	Formular manuell aufbauen	484
11.5.2	Smart Form verwenden	486
11.6	Fragmente	488
11.6.1	Detailseite vorbereiten	488
11.6.2	Fragmente direkt einbetten	490
11.6.3	Fragmente aus dem Controller laden	492
11.7	SAPUI5-Projekte von einem Git Repository klonen	500
11.8	Zusammenfassung	503

12 Entwicklung von Benutzeroberflächen mit SAP Fiori Elements

505

12.1	List Report	506
12.1.1	Anwendungsfälle	508
12.1.2	Annotationen des List Reports	508
12.1.3	Implementierung eines List Reports	509
12.2	Object Page	514
12.2.1	Anwendungsfälle	516
12.2.2	Annotationen der Object Page	516
12.2.3	Implementierung einer Object Page	519
12.3	Overview Page	527
12.3.1	Cards	528
12.3.2	Anwendungsfälle	529
12.3.3	Best Practices für das Design der Pages	530
12.3.4	Annotationen der Overview Page	531
12.3.5	Implementierung einer Overview Page	532
12.4	Generische Annotationen	540
12.4.1	UI.HeaderInfo	541
12.4.2	UI.selectionField	541
12.4.3	UI.lineItem	542
12.4.4	UI.hidden	542
12.5	Zusammenfassung	543

13 Entwicklung von Benutzeroberflächen mit dem Flexible Programming Model	545
13.1 Einleitung	545
13.2 Programming Model Explorer	546
13.2.1 Erweiterungspunkte	547
13.2.2 Building Blocks	550
13.2.3 Controller-Erweiterung	551
13.2.4 SAP Fiori Elements: erweiterte Funktionen	552
13.3 Building Blocks	553
13.3.1 Field	554
13.3.2 Form	554
13.3.3 FormElement	555
13.3.4 Table	556
13.3.5 FilterBar	557
13.3.6 FilterField	558
13.3.7 MicroChart	558
13.3.8 Chart	559
13.3.9 FlexibleColumnLayoutActions	560
13.3.10 Share	560
13.3.11 Paginator	561
13.4 Implementieren des Flexible Programming Model	561
13.4.1 Das Flexible Programming Model in Freestyle-Applikationen	561
13.4.2 Neue Flexible-Programming-Model-Applikationen	562
13.4.3 Page Map	570
13.4.4 Controller-Erweiterung	576
13.5 Zusammenfassung	579

TEIL III Fortgeschrittene Themen

14 Datei-Upload	583
14.1 Media Entitys	583
14.2 OData V2	584
14.2.1 On-Premise-Grundlagen	584
14.2.2 OData-Service	587

14.2.3	Datei-Upload	593
14.2.4	Datei-Download	596
14.3	OData V4	597
14.3.1	Erweitern der Datenbanktabellen	598
14.3.2	Felder im ABAP RESTful Application Programming Model verfügbar machen	598
14.3.3	Projizieren der Felder	599
14.3.4	Vorschau der Applikation	600
14.4	Zusammenfassung	600

15 Formularvalidierung und Error Handling 601

15.1	Formularvalidierung	601
15.1.1	Validierungstypen	602
15.1.2	Empfehlungen der SAP Fiori Design Guidelines	605
15.1.3	Implementierung	607
15.2	Fehlerbehandlung	619
15.2.1	Empfehlungen der SAP Fiori Design Guidelines	619
15.2.2	Serverseitige OData-Implementierung	619
15.2.3	SAP-Fiori-Elements-Implementierung	625
15.3	Zusammenfassung	626

16 Entity Tags und Soft State 627

16.1	Soft-State-Aktivierung	627
16.1.1	DEFINE-Methode redefinieren	628
16.1.2	Instanzattribut anlegen	628
16.1.3	OPERATION_START-Methode redefinieren	629
16.1.4	OPERATION_END-Methode redefinieren	630
16.1.5	Session-Timeout setzen	630
16.2	ETags	631
16.2.1	ETags im ABAP RESTful Application Programming Model	632
16.2.2	ETags in SEGW-basierten OData-Services	632
16.3	Zusammenfassung	633

17 Erweiterbarkeit	635
17.1 Adaptation Projects	635
17.1.1 Erstellen eines Adaptation Projects	637
17.1.2 Erstellen von Erweiterungen im grafischen Editor	639
17.1.3 Deployen eines Adaptation Projects	645
17.2 Key-User-Erweiterbarkeit	647
17.3 Zusammenfassung	653
18 Bereitstellung	655
18.1 Continuous Integration, Continuous Delivery und Continuous Deployment	656
18.1.1 CTS und CTS+	658
18.1.2 SAP Continuous Integration and Delivery Service	661
18.2 Bereitstellung in SAP S/4HANA	672
18.3 Bereitstellung in der SAP Business Technology Platform	676
18.4 Zusammenfassung	680
Das Autorenteam	681
Index	683

Frontend und Backend entwickeln mit SAP

Effektive End-to-End-Entwicklung mit SAP

Von den Grundlagen bis zu fortgeschrittenen Techniken: Die Autoren begleiten Sie anhand vieler Beispiele durch den gesamten Entwicklungsprozess für moderne, ansprechende SAP-Anwendungen.

Frontend, Backend, Datenmodell – alles im Blick

Verknüpfen Sie die ABAP-Plattform oder SAP BTP mit SAP-Fiori-Frontends. Lernen Sie alle relevanten Technologien kennen, von OData und Core Data Services bis zu SAP Fiori, SAPUI5 und TypeScript.

Praxiserprobte Entwicklungsszenarien

Lernen Sie anhand detaillierter Anleitungen, wie Sie Daten über OData-Services bereitstellen, UI-Elemente implementieren, Formulare validieren, Fehler behandeln, Anwendungen deployen und vieles mehr.

»Werden Sie zum Allrounder in der SAP-Entwicklung!«

Auf einen Blick

- Core Data Services (CDS)
- SAPUI5
- SAP Fiori Elements
- OData V4 und OData V2
- TypeScript
- SAP Cloud Application Programming Model
- ABAP RESTful Application Programming Model
- SAP Gateway Service Builder
- Entwicklung von Benutzeroberflächen
- Flexible Programming Model
- Erweiterbarkeit und Bereitstellung



Die Autoren

Das Autorenteam der CloudDNA GmbH, bestehend aus Rene Glavanovits, Martin Koch, Daniel Krancz und Maximilian Olzinger, hat viele Jahre Erfahrung im SAP-Umfeld gesammelt. Als Consultants, Entwickler und Trainer sind die Autoren Experten für moderne SAP-Technologien wie SAP Fiori, SAPUI5 und die SAP BTP.

