

## Reading Sample

*This chapter covers different web standards and data formats you can use for configuration files and data transfer over the internet. You'll start by walking through application programming interfaces (APIs), representational state transfers (REST), and Open Data (OData). Then, you'll learn about programming languages used for web development, including JavaScript Object Notation (JSON) and Yet Another Markup Language (YAML). You'll also review the twelve-factor app principles for building and designing resilient applications. As with all other chapters in the book, this chapter contains practice questions with detailed answers to help you better understand the topic.*



**"Web Development Standards"**



**Contents**



**Index**



**The Author**

Krishna Kishor Kammaje, Mahesh Kumar Palavalli

### **SAP Extension Suite Certification Guide: Development Associate Exam**

307 pages, 2023, \$79.95

ISBN 978-1-4932-2239-1



[www.sap-press.com/5490](http://www.sap-press.com/5490)

# Chapter 2

## Web Development Standards



### Techniques You'll Master

- Understanding web development standards
- Using APIs and the importance of REST and OData
- Working with JSON and YAML data formats.
- Building a twelve-factor app

Web technologies are evolving at a fast pace. Web development standards are established by various independent bodies specifying how the developers must build products, how they interact with other standards, and how usability and accessibility features are used. This chapter covers different web standards and data formats you use for configuration files and data transfer over the internet. You'll also learn about the twelve-factor app principles for building and designing resilient applications.

Real-World Scenario

You're developing a modern, enterprise-grade resilient web application and need to deploy this application to an industry-standard cloud provider. You then need to understand which types of application programming interfaces (APIs) are required and the guidelines to develop resilient applications.

2.1 Objectives of This Portion of the Test

This portion of the test checks your understanding of the web development standards. Application programming interfaces (APIs) play a significant role in moving data to and from user interfaces (UIs). You're expected to know about representational state transfer (REST) principles in API design and development. You'll also have to learn about Open Data Protocol (OData), a REST-based protocol for building and consuming RESTful APIs. You'll learn about Yet Another Markup Language (YAML) and JavaScript Object Notation (JSON), which are useful for defining configuration files, as well as the best practices for developing resilient applications.



Note

Web development standards topics make up less than 8% of the total exam.

2.2 Application Programming Interface, Representational State Transfer, and Open Data

For different systems to communicate with each other to exchange data, you need an interface called an API. There are various ways to architect the APIs, and REST, Simple Object Access Protocol (SOAP), and OData are the most popular formats. We'll discuss these in detail in the following sections.

2.2.1 API

APIs enable companies to securely connect their internal applications or external partners to transfer data both ways. An API will have a set of protocols that defines

the rules of how it's accessed. Consumers don't need to know how an API is implemented; they only need the documentation that is explicitly provided or comes automatically with the API.

The role of an API is to act as an interface or an intermediate layer that transfers the data between multiple partners, as shown in Figure 2.1. The actors and their responsibilities in Figure 2.1 are as follows:

- Consumers, who can be the end users and third-party companies, request the API to access the data from their applications.
- API producers take care of creating, hosting, and managing the API.
- The API gets the request from the consumers, validates it, and queries the database to fetch the data.
- Data is processed and sent back to the consumer applications.

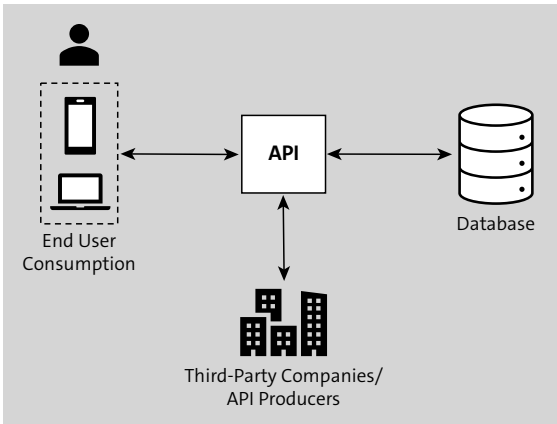


Figure 2.1 High-Level API Design

APIs should not expose all the data of the database or the server; they should only show the required and authorized information when they are accessed. For instance, A bank application only lets you access your account details; it should not show other users' accounts. So it's recommended to include robust security protocols while designing an API.

There are different ways to architect the APIs, and the widely used architectures are SOAP and REST. SOAP is a stricter protocol with defined security rules and is standardized by the World Wide Web Consortium (W3C), which uses XML payload to transfer the data between systems. Although SOAP used to be popular, the REST protocol is now used mainly for developing APIs. In the next section, we'll cover REST in detail, which is the most used architecture today.

2.2.2 REST

REST is a famous web API architecture that is mainly used for lightweight web services and mobile applications. Unlike SOAP, which has stricter protocols, REST

comes with guiding principles, and the APIs that implement these guidelines are referred to as RESTful APIs. Following are the guiding principles and constraints that a RESTful API should have:

- **Client-server architecture**  
The client-server architecture enforces the separation of client and server, which allows them to be updated and evolve independently. The client will only request the server for the data using an API, and the server can only return the data via HTTP requests without interacting with the clients.  
For example, consider a mobile bank application (client), where only the application is available in the user’s mobile phone, but not the database or the business logic of the bank process. Users can only see their account details via the API provided by the bank. The business logic and database reside in the server, where the requests coming from the API are authenticated and processed.
- **Statelessness**  
Statelessness requires that all the requests originated from the client be unconnected, and each request should have all the information to complete the request.
- **Cacheability**  
Cache ability suggests that, when required, resources should be cacheable on the client side or server side to increase the performance of applications.
- **Layered system**  
The layered system in a server takes care of the request in different stages (firewall, load balancing, security, etc.). The server should hide all these processes from the client and only respond to the request by returning the data.
- **Uniform interface**  
Uniform interface mandates that all API requests should have a standard format. The following multiple constraints should be applied while designing an API:
  - All the resources requested from the client must be uniquely identifiable.
  - Server responses should have uniform representations, including metadata, which API consumers can use to modify the resources in the server.
  - Response messages from the server should be informative enough for the client to process them.
  - Hyperlinks to different actions should be available in the server response to discover the available resources the API needs.
- **Code on demand**  
Code on demand is the ability to send the executable code to the client to customize different functionalities. For instance, the client can use Java applets or JavaScript code to extend its functionality.

2.2.3 OData

OData is a REST-based protocol approved by International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) and used for building and consuming RESTful APIs. It helps you focus on business logic without needing to worry about the different API approaches in defining your request, response, status codes, URL conventions, query options, and so on. OData supports XML-based AtomPub and JSON formats.

OData protocol follows these design principles to achieve the uniformity of both the data and the data model:

- Follow the REST principles as OData is the best way to REST.
- Support the API extension without breaking the client’s functionality.
- Keep it simple by addressing the common functionalities and providing the extensions wherever required.
- Prefer mechanisms that work on a variety of data sources. Don’t assume a relational data model.
- Build incrementally—a basic, compliant service should be easy to build, and additional work should be necessary only to support additional capabilities.

In SAP, the OData protocol is primarily used to generate the APIs. For example, SAP S/4HANA uses the ABAP RESTful Application Programming Model (RAP) to generate OData V2 and V4 services, and in SAP BTP, the SAP Cloud Application Programming model is used for generating the OData services. Even the UI, SAPUI5, has built-in libraries to consume OData V2 and V4 models, which use metadata and annotations to construct the UI dynamically. A typical OData service looks like Figure 2.2.

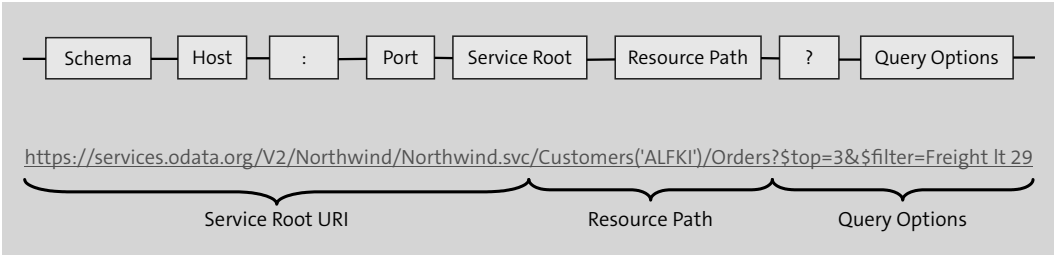


Figure 2.2 OData URI Components

Let’s look at the OData request URL and its components in detail:

- **Service root URI**  
The service root URI is the root of an OData service through which different resources can be accessed and created. OData services provide two types of metadata documents, as follows:



- A *service document* has all top-level feeds so clients can discover them and access only those required (see Figure 2.3). The service document is available by accessing the service root URI directly.

```
<service xml:base="https://services.odata.org/V2/Northwind/Northwind.svc/">
  <workspace>
    <atom:title>Default</atom:title>
    <collection href="Categories">
      <atom:title>Categories</atom:title>
    </collection>
    <collection href="CustomerDemographics">
      <atom:title>CustomerDemographics</atom:title>
    </collection>
    <collection href="Customers">
      <atom:title>Customers</atom:title>
    </collection>
    <collection href="Employees">
      <atom:title>Employees</atom:title>
    </collection>
  </workspace>
</service>
```

Figure 2.3 Service Document



Example

To see an example of an OData service, go to: <http://s-prs.co/v540902>.

- A *service metadata document* is usually referred to as metadata. This document describes the Entity Data Model (EDM) for a given service. You can get all the information about different resources, associations, and properties from the EDM.



Example

To access the metadata of an OData service, you must add *\$metadata* to the end of the service as shown in this URL: [https://services.odata.org/V2/Northwind/Northwind.svc/\\$metadata](https://services.odata.org/V2/Northwind/Northwind.svc/$metadata).

As you can see in Figure 2.4, the metadata document provides all the entities, properties, and relations (navigation properties). Along with the properties of the individual entity, it also returns the key properties. For instance, the customer entity has *CustomerID* as the key property. An *entity* is a single resource; it can be a customer, order, and so on. *Entity type* describes an entity (properties, keys, navigations, etc.), and *entity set* is a collection of entities. For instance, an entity set can be customers (multiple), orders, and so on.



Note

The EDM is a set of concepts that describe the structure of data, regardless of its stored form.

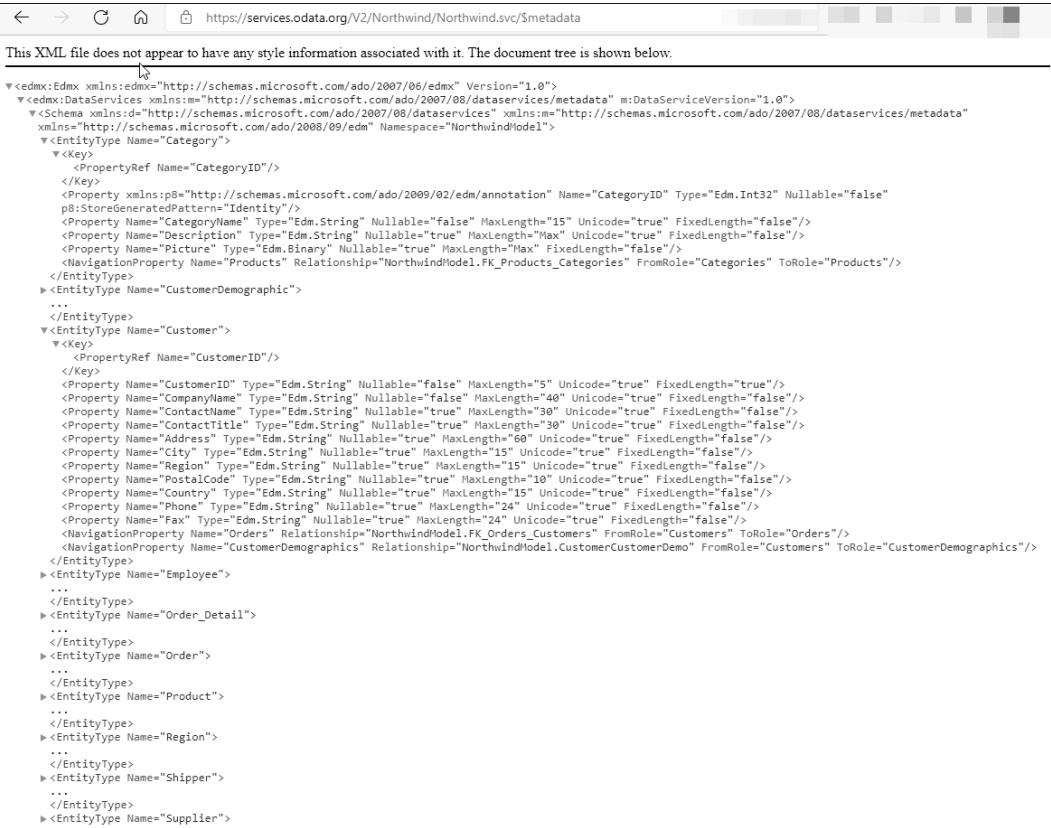


Figure 2.4 Metadata of an OData Service

■ Resource path

The resource path URI identifies the resources of an OData service such as customers, a single customer, or orders related to a single customer. We'll look at each of them here:

- **Requesting resources (entity set)**

These collections are called entity sets. Accessing each entity set will provide the data related to it. For instance, the categories entity set will return all the categories, and the customers entity set will give all the customers. In the following example, you can see the entity set *Customers* is added at the end of the URL to fetch the list of customers.

Example

To request an entity set, you need to access the URL as shown in the example at: <http://s-prs.co/v540903>.



As you can see in Figure 2.5, the response resources are in XML format because, by default, OData V2 output is in XML format. But in the case of

OData V4, the default response format will be in JSON. You can learn more about this format in Section 2.3.1.

– Requesting an individual resource

If you want to get details of a single customer in the system, you pass the key property *CustomerID* in the URL (see Figure 2.6 for an example).



Example

Visit the following link to see another example of passing the key property in the URL: <http://s-prs.co/v540904>.

Alternatively, if there is only one key property in the entity, it's not required to pass the property name.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xmlns:base="https://services.odata.org/V2/Northwind/Northwind.svc/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Customers</title>
  <id>https://services.odata.org/V2/Northwind/Northwind.svc/Customers</id>
  <updated>2022-03-05T16:48:12Z</updated>
  <link rel="self" title="Customers" href="Customers" />
  <entry>
    <id>https://services.odata.org/V2/Northwind/Northwind.svc/Customers('ALFKI')</id>
    <title type="text"></title>
    <updated>2022-03-05T16:48:12Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Customer" href="Customers('ALFKI')"/>
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Orders" type="application/atom+xml;type=feed" title="Orders" href="Customers('ALFKI')/Orders" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/CustomerDemographics" type="application/atom+xml;type=feed" title="CustomerDemographics" href="Customers('ALFKI')/CustomerDemographics" />
    <category term="NorthwindModel.Customer" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:CustomerID m:type="Edm.String">ALFKI</d:CustomerID>
        <d:CompanyName m:type="Edm.String">Alfreds Futterkiste</d:CompanyName>
        <d:ContactName m:type="Edm.String">Maria Anders</d:ContactName>
        <d:ContactTitle m:type="Edm.String">Sales Representative</d:ContactTitle>
        <d:Address m:type="Edm.String">Obere Str. 57</d:Address>
        <d:City m:type="Edm.String">Berlin</d:City>
        <d:Region m:type="Edm.String" m:null="true" />
        <d:PostalCode m:type="Edm.String">12209</d:PostalCode>
        <d:Country m:type="Edm.String">Germany</d:Country>
        <d:Phone m:type="Edm.String">030-0074321</d:Phone>
        <d:Fax m:type="Edm.String">030-0076545</d:Fax>
      </m:properties>
    </content>
  </entry>
```

Figure 2.5 OData Customers Entity Set Output

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xml:base="https://services.odata.org/V2/Northwind/Northwind.svc/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <id>https://services.odata.org/V2/Northwind/Northwind.svc/Customers('ALFKI')</id>
  <title type="text"></title>
  <updated>2022-03-08T15:15:09Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="Customer" href="Customers('ALFKI')"/>
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Orders" type="application/atom+xml;type=feed" title="Orders" href="Customers('ALFKI')/Orders" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/CustomerDemographics" type="application/atom+xml;type=feed" title="CustomerDemographics" href="Customers('ALFKI')/CustomerDemographics" />
  <category term="NorthwindModel.Customer" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:CustomerID m:type="Edm.String">ALFKI</d:CustomerID>
      <d:CompanyName m:type="Edm.String">Alfreds Futterkiste</d:CompanyName>
      <d:ContactName m:type="Edm.String">Maria Anders</d:ContactName>
      <d:ContactTitle m:type="Edm.String">Sales Representative</d:ContactTitle>
      <d:Address m:type="Edm.String">Obere Str. 57</d:Address>
      <d:City m:type="Edm.String">Berlin</d:City>
      <d:Region m:type="Edm.String" m:null="true" />
      <d:PostalCode m:type="Edm.String">12209</d:PostalCode>
      <d:Country m:type="Edm.String">Germany</d:Country>
      <d:Phone m:type="Edm.String">030-0074321</d:Phone>
      <d:Fax m:type="Edm.String">030-0076545</d:Fax>
    </m:properties>
  </content>
</entry>
```

Figure 2.6 Accessing an Individual Resource

Example

You see an example of the preceding scenario at the following URL: <http://s-prs.co/v540905>.

– Navigation/relationships

You can define relationships among the resources in the OData service. Each resource (entity) can be logically connected to another entity by this concept. For instance, a customer can get the orders created by using the navigation property *Orders* (see Figure 2.7 for an example).

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xmlns:base="https://services.odata.org/V2/Northwind/Northwind.svc/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Orders</title>
  <id>https://services.odata.org/V2/Northwind/Northwind.svc/Customers('%20'ALFKI')/Orders</id>
  <updated>2022-03-11T05:05:12Z</updated>
  <link rel="self" title="Orders" href="Orders" />
  <entry>
    <id>https://services.odata.org/V2/Northwind/Northwind.svc/Orders(10643)</id>
    <title type="text"></title>
    <updated>2022-03-11T05:05:12Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Order" href="Orders(10643)"/>
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Customer" type="application/atom+xml;type=entry" title="Customer" href="Orders(10643)/Customer" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Employee" type="application/atom+xml;type=entry" title="Employee" href="Orders(10643)/Employee" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Order_Details" type="application/atom+xml;type=feed" title="Order_Details" href="Orders(10643)/Order_Details" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Shipper" type="application/atom+xml;type=entry" title="Shipper" href="Orders(10643)/Shipper" />
    <category term="NorthwindModel.Order" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:OrderID m:type="Edm.Int32">10643</d:OrderID>
        <d:CustomerID m:type="Edm.String">ALFKI</d:CustomerID>
        <d:EmployeeID m:type="Edm.Int32">6</d:EmployeeID>
        <d:OrderDate m:type="Edm.DateTime">1997-08-23T00:00:00</d:OrderDate>
        <d:RequiredDate m:type="Edm.DateTime">1997-09-22T00:00:00</d:RequiredDate>
        <d:ShippedDate m:type="Edm.DateTime">1997-09-02T00:00:00</d:ShippedDate>
        <d:ShipVia m:type="Edm.Int32">1</d:ShipVia>
        <d:Freight m:type="Edm.Decimal">29.4600</d:Freight>
        <d:ShipName m:type="Edm.String">Alfreds Futterkiste</d:ShipName>
        <d:ShipAddress m:type="Edm.String">Obere Str. 57</d:ShipAddress>
        <d:ShipCity m:type="Edm.String">Berlin</d:ShipCity>
        <d:ShipRegion m:type="Edm.String" m:null="true" />
        <d:ShipPostalCode m:type="Edm.String">12209</d:ShipPostalCode>
        <d:ShipCountry m:type="Edm.String">Germany</d:ShipCountry>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>https://services.odata.org/V2/Northwind/Northwind.svc/Orders(10692)</id>
    <title type="text"></title>
    <updated>2022-03-11T05:05:12Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Order" href="Orders(10692)"/>
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Customer" type="application/atom+xml;type=entry" title="Customer" href="Orders(10692)/Customer" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Employee" type="application/atom+xml;type=entry" title="Employee" href="Orders(10692)/Employee" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Order_Details" type="application/atom+xml;type=feed" title="Order_Details" href="Orders(10692)/Order_Details" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Shipper" type="application/atom+xml;type=entry" title="Shipper" href="Orders(10692)/Shipper" />
    <category term="NorthwindModel.Order" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:OrderID m:type="Edm.Int32">10692</d:OrderID>
        <d:CustomerID m:type="Edm.String">ALFKI</d:CustomerID>
        <d:EmployeeID m:type="Edm.Int32">4</d:EmployeeID>
        <d:OrderDate m:type="Edm.DateTime">1997-10-03T00:00:00</d:OrderDate>
        <d:RequiredDate m:type="Edm.DateTime">1997-10-31T00:00:00</d:RequiredDate>
        <d:ShippedDate m:type="Edm.DateTime">1997-10-13T00:00:00</d:ShippedDate>
        <d:ShipVia m:type="Edm.Int32">2</d:ShipVia>
        <d:Freight m:type="Edm.Decimal">61.0200</d:Freight>
        <d:ShipName m:type="Edm.String">Alfreds Futterkiste</d:ShipName>
        <d:ShipAddress m:type="Edm.String">Obere Str. 57</d:ShipAddress>
        <d:ShipCity m:type="Edm.String">Berlin</d:ShipCity>
      </m:properties>
    </content>
  </entry>
```

Figure 2.7 Navigation property

Example

To access the related entities via navigation, see the example at the following URL: <http://s-prs.co/v540906>.

■ Queries

With the help of queries, you can filter, sort, or restrict the response fields of your OData request. These query options are added at the end of an OData service, as shown in the following illustration.

For sorting the data by ascending and descending, you can pass the `orderby` system query option (`$orderby`).



**Example**

You can apply sorting in the following ways:

- Ascending (see <http://s-prs.co/v540907> for an example)
- Descending (see <http://s-prs.co/v540908> for an example)

If you want to get the top three entries, then you can use the system query option `$top`, which gives the top *n* entries.



**Example**

To access the top *n* entries, add `$top` at the end of the URL. For the top two customer entries, the URL is [https://services.odata.org/V2/Northwind/Northwind.svc/Customers?\\$top=2](https://services.odata.org/V2/Northwind/Northwind.svc/Customers?$top=2).

You can also skip the first *n* entries and get the remaining ones by using `$skip`.



**Example**

To skip the top *n* entries, add `$skip` at the end of the URL. For skipping two customer entries, the URL is [https://services.odata.org/V2/Northwind/Northwind.svc/Customers?\\$skip=2](https://services.odata.org/V2/Northwind/Northwind.svc/Customers?$skip=2).

You can combine `$top` and `$skip` to simulate pagination with OData. SAPUI5 controls such as Table and List natively support this function to simulate pagination (threshold approach). This approach is mainly used in SAP Fiori elements applications to display the data efficiently.



**Example**

The following URL combines `$top` and `$skip` to simulate pagination: [https://services.odata.org/V2/Northwind/Northwind.svc/Customers?\\$skip=2&\\$top=1](https://services.odata.org/V2/Northwind/Northwind.svc/Customers?$skip=2&$top=1).

To find out the total number of entries in the collection of entries, you can use `$count` or query option `$inlinecount` that provides both entries and count. The main difference is that `$inlinecount` provides both the count and data in the same request, whereas `$count` only returns the total count of entries. SAPUI5 controls and SAP Fiori elements templates use `$count` to perform pagination (see Figure 2.8 for an example of `$inlinecount`).

**Example**

Visit the following URLs to see examples of the `$count` and `$inlinecount` options:

- `$count`: <http://s-prs.co/v540909>
- `$inlinecount`: <http://s-prs.co/v540910>

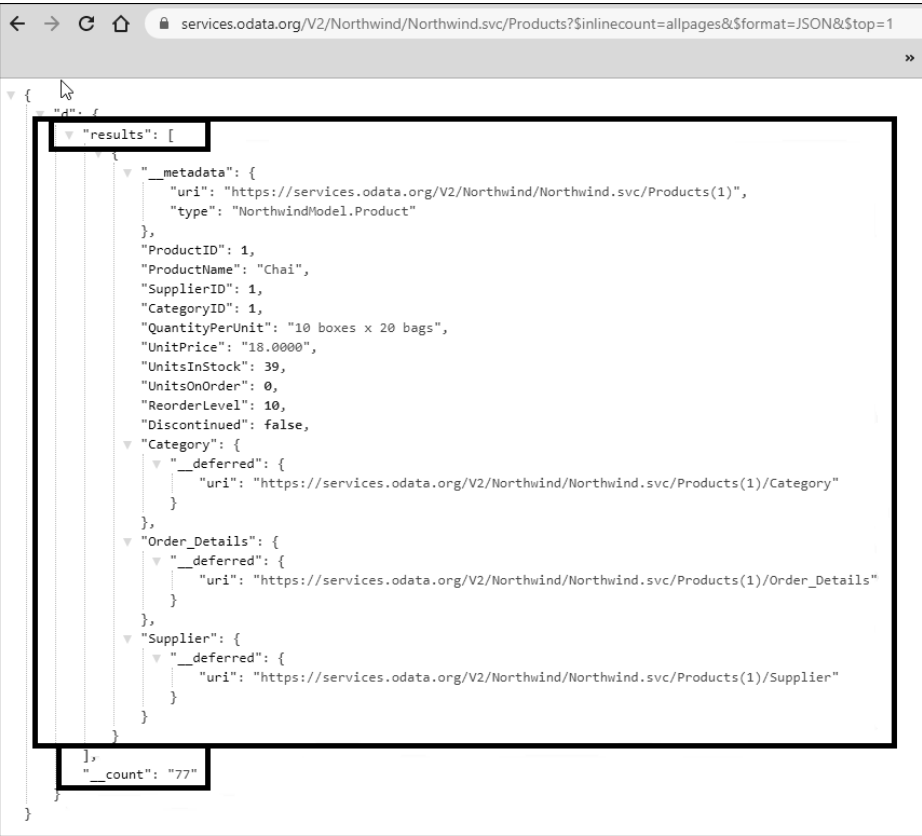


Figure 2.8 OData Inline Count

For filtering all the customers belonging to a country (e.g., Germany), you have to use the query option `$filter` along with the filter operator `eq`, as shown in Figure 2.9.

**Example**

Visit the following URL for an example of a query option: <http://s-prs.co/v540911>.

You can also use other filter operators to filter the entries, such as `Ne` (not equal), `gt` (greater than), or functions such as `startswith`, `endswith`, and `tolower`.



Figure 2.9 OData Filter Query Parameter



Example

Visit the following URLs for examples of filters:

- Less than filter: <http://s-prs.co/v540912>  
This URL returns the entries where freight is less than 10.
- Greater than filter: <http://s-prs.co/v540913>  
This URL returns entries with freight greater than 200.

You can use the OR operator to perform a logical OR operation on the data. The data contains Germany or Mexico as the country.

Example

The following example shows multiple filters with the logical OR operator: <http://s-prs.co/v540914>.

You can refer to the following documentation for other operators and functions that you can use in the filters: <http://s-prs.co/v540915>. You can also format the output as JSON instead of XML with the \$format query option, as shown in Figure 2.10.

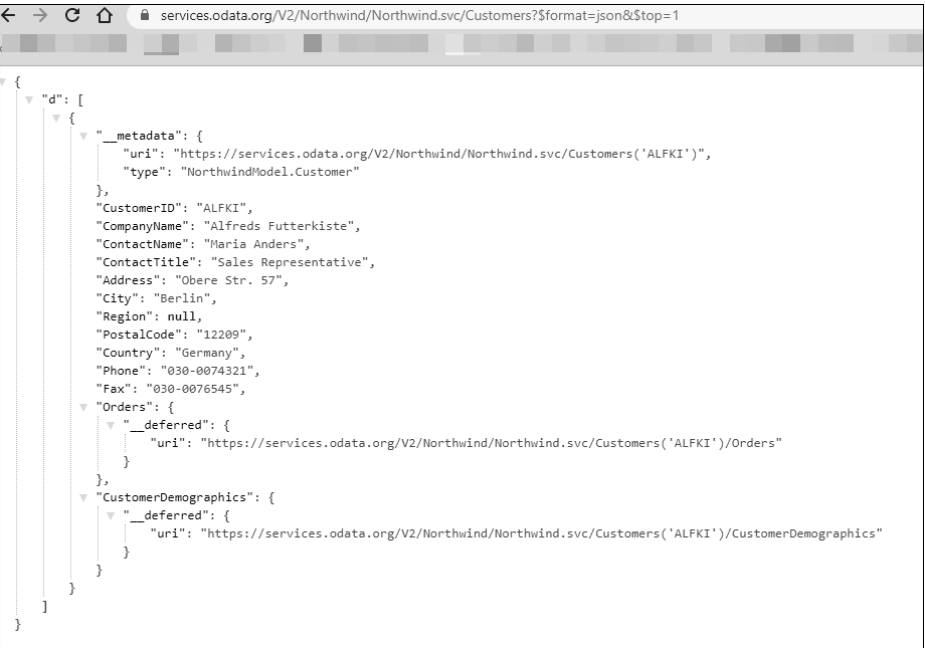


Figure 2.10 JSON Formatted Output

Example

Adding \$format at the end of the URL will format the output: [https://services.odata.org/V2/Northwind/Northwind.svc/Customers?\\$format=json&\\$top=1](https://services.odata.org/V2/Northwind/Northwind.svc/Customers?$format=json&$top=1).

You saw earlier how to get customer relationship data using the navigation property Orders. If you use that, the service only gives the orders entries. To get data for multiple relationships simultaneously, along with the parent entity, you can use the \$expand query option, as shown in Figure 2.11.

Example

To access both parent and related child entities data, you can use \$expand, as shown in this URL: [https://services.odata.org/V2/Northwind/Northwind.svc/Products?\\$expand=Supplier,Order\\_Details&\\$format=json&\\$top=1](https://services.odata.org/V2/Northwind/Northwind.svc/Products?$expand=Supplier,Order_Details&$format=json&$top=1).





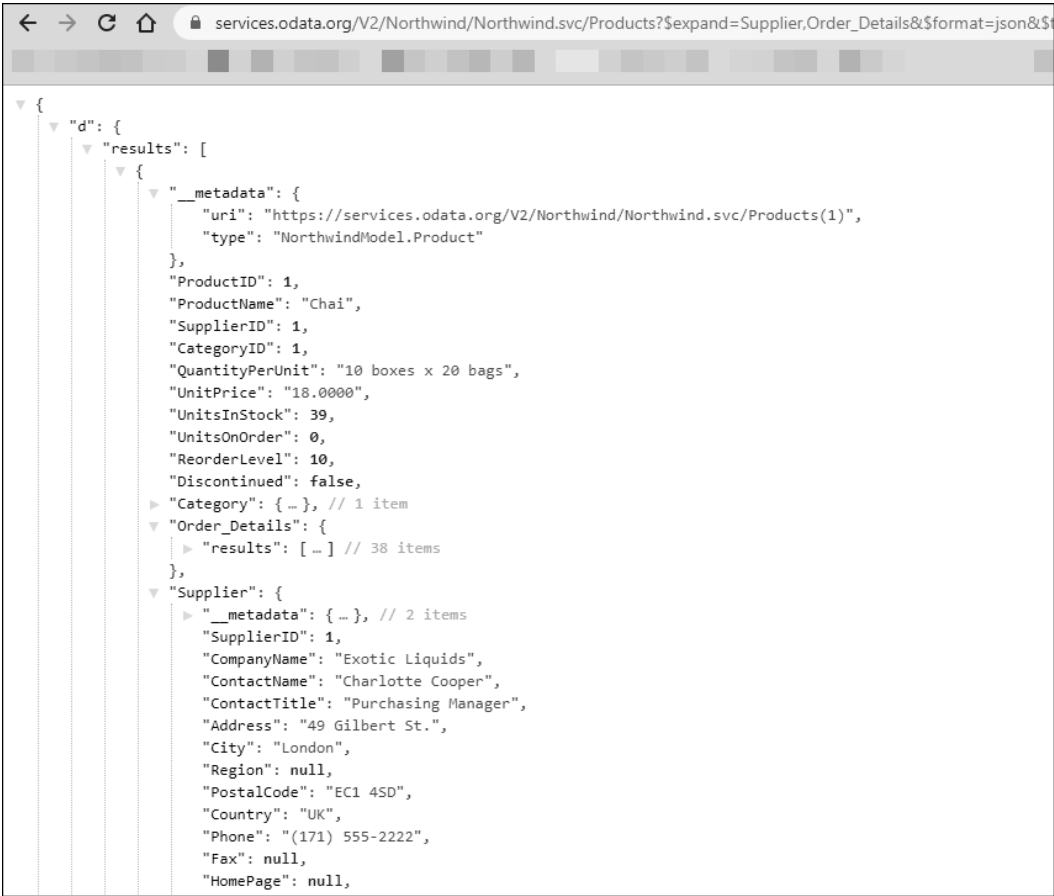


Figure 2.11 OData Expand

In Figure 2.11, you can see that we’re fetching the relationship data `Supplier`, `Order_Details` for the `Products` entity. The response also has the parent data (`Products`) along with the related entities.

You can use `$select` to specify the response entity’s properties. For example, you can pass the properties `ProductID` and `ProductName` to the query option `$select` to fetch only those properties, as shown in Figure 2.12.



**Example**

See an example of how to use query option `$select` at the following URL: <http://s-prs.co/v540916>.

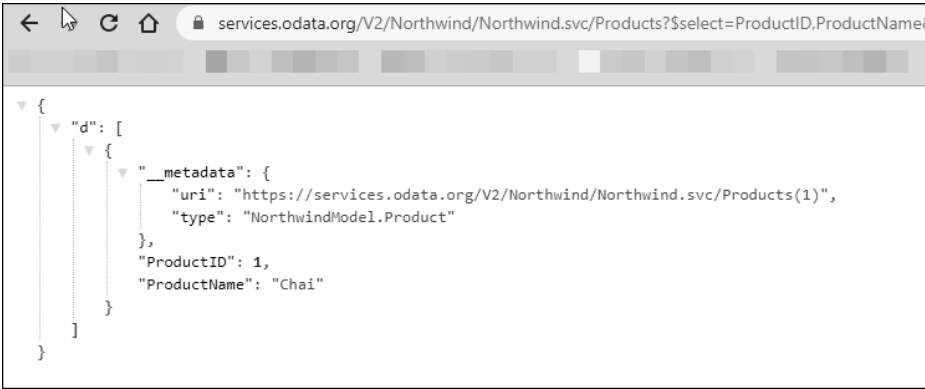


Figure 2.12 OData \$select

## 2.3 JavaScript Object Notation and Yet Another Markup Language

JSON and YAML are two popular formats for data exchange and configuration files. They are similar in functions and features, but the difference will be in the design, which affects the scope of use. This section will provide an overview of JSON and YAML, followed by an explanation of their differences.

### 2.3.1 JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays. It’s one of the common data formats used in modern web applications. Although JSON was derived from JavaScript, it’s a language-independent data format. JSON files use the extension `.json`.

JSON is built on two structures:

- A collection of name-value pairs, which are called object, record, or struct in other programming languages
- An ordered list of values, which are called array, vector, or sequence in most languages.

#### JSON Object/Structure

An object is an unordered set of name-value pairs. It starts with a left brace (`{`) and ends with a right brace (`}`). Name-value pairs, separated by a colon (`:`) will be added inside these braces, and these values can be of type string, number, array, or other data types.



Example

Listing 2.1 shows an example JSON object.

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    }
  ]
}
```

Listing 2.1 Example JSON Object

You can notice in the preceding example that the JSON object has a mix of data types: a string, "Super hero squad"; a number, 2016; a Boolean, true; and an array.

JSON Array

An *array* is an ordered collection of objects. They begin with [ (left bracket), end with ] (right bracket), and have JSON objects between them.



Example

Listing 2.2 shows an example array.

```
[
  {
    "name": "Hyper Man",
    "age": 19,
    "secretIdentity": "Dan King",
    "powers": [
      "Super Strength",
      "Cold Breath",
      "Flight"
    ]
  },
  {
    "name": "Madame Tsunade",
    "age": 29,
    "secretIdentity": "Jane Tsunade",
  }
]
```

```
"powers": [
  "Million times punch",
  "Super Strength",
]
}
```

Listing 2.2 Example Array

2.3.2 YAML

YAML is a human-friendly data serialization language. It's a superset of JSON, so JSON files are valid in YAML. The extensions of YAML files are *.yaml* or *.yml*. It's primarily used in configuration files, internet messaging, data auditing, and so on.

The YAML file is made up of name-value pairs but follows a special indentation to indicate nesting. It doesn't have braces or square brackets like JSON does.



Example

A typical YAML file looks like Listing 2.3.

```
---
# Super Hero Detail
Name: Hyper Man
Age: 19
Secret Identity: Dan King
Powers:
- Super Strength
- Cold Breath
- Flight
Description1: >
  Although this text
  appears in different
  lines, it will be formatted
  to single line
```

Listing 2.3 Example YAML

From the preceding example, you can observe that name-value pairs are separated by a colon (:) and a long string, which, when specified with >, ensures that a multi-line string is formatted to a single line. A comment will have a hash (#) in the beginning. An array of values is separated by - with a single tab indentation.



Example

You can see an array in the following formats:

```
Type1:['value1', 'value2', 'value3']
Type2:
- value1
- value2
- value3
```

```
Type3:
- id: 1
  Name: franc
- id: 11
  Name: Tom
```

The YAML array example can also be represented in the JSON format as shown here:

```
{
  "Type1": [
    "value1",
    "value2",
    "value3"
  ],
  "Type2": [
    "value1",
    "value2",
    "value3"
  ],
  "Type3": [
    {
      "id": 1,
      "name": "franc"
    },
    {
      "id": 11,
      "name": "Tom"
    }
  ]
}
```

2.3.3 YAML versus JSON

Although both YAML and JSON are two popular human-readable data formats, both have different priorities. Technically, YAML is a superset of JSON, so it offers more functionalities than JSON; for example, in a JSON file, you can have duplicate key values, but that can be prevented easily in YAML files. So, you can say for this specific reason, YAML is preferred in configuration files. YAML tends to be simpler to read and understand because you won't find unnecessary braces. It also has tons of better features such as comments, recursive structures, and so on. YAML also supports nonhierarchical data and doesn't need to follow the typical parent and child relationship like the JSON model.

On the other hand, JSON isn't as complex as YAML as it has simple data types and structures. It can also be parsed and generated faster than YAML. Because the JSON format is used mostly in JavaScript, it has a better developer ecosystem than YAML.

2.4 Twelve-Factor App Principles

Web applications are ubiquitous these days. As browsers become more and more powerful, desktop applications are shrinking, and delivering applications via browser is becoming increasingly common. These applications are called web applications as they are accessed using the World Wide Web (the internet).

As web applications became popular in the past two decades, many best practices and principles have emerged to develop and maintain web applications, including the twelve-factor app set of principles for building performant, scalable, and resilient web applications.

As the name suggests, twelve-factor app principles provide 12 principles for creating modern, microservice-based cloud-native web applications. It was originally introduced by Adam Wiggins in the 2011. He also happens to be the cofounder of Heroku, which was once a very popular platform-as-a-service (PaaS) for running enterprise applications.

Now, let's explore these 12 principles:

- **Codebase**  
The codebase principle states that the application resources should have a version-managed, source-code repository such as Git or Subversion. Multiple apps sharing the same code is a violation of the twelve-factor app principles. The solution here is to factor shared code into libraries that can be included through the dependency manager.  
Per the twelve-factor app methodology, a deploy is an instance of an app in the developer's system, development system, quality system, and production system, though different versions can be active in each deployment (see Figure 2.13). The codebase can be accessible to Continuous Integration/Continuous Delivery (CI/CD) as part of the software development cycle. A system's codebase can be in sync with some level of automation using CI/CD.

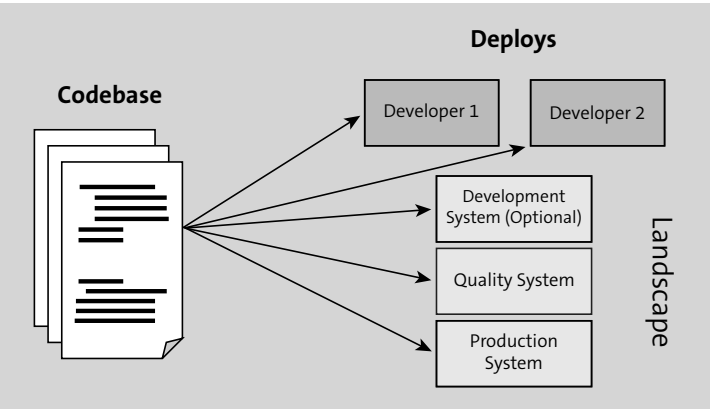


Figure 2.13 Twelve-Factor App Principle: Codebase

■ Dependencies

When you build an application, it's common to use other reusable repositories and tools within the application that you create. One easy way to handle dependencies is to include those dependent library codes into your application itself. However, that comes with the disadvantage of mixing the lifecycle of the application code and the dependent library code.

This principle specifies that the dependencies should be set in your app manifest or configuration file and managed externally instead of including them directly in the source code.

If you're using Node.js, you specify the dependencies inside a *package.json* file (see Figure 2.14). The node package manager (NPM) takes care of downloading the specified versions and installing them to make them available for the application to use. Similarly, in a Java application, you set the dependencies in a *build.gradle* file, and the Gradle dependency manager takes care of those dependencies.

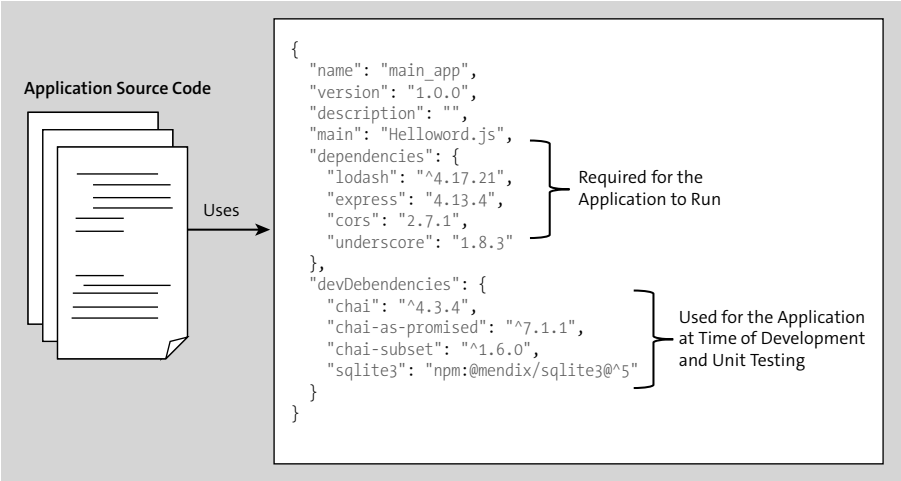


Figure 2.14 Twelve-Factor App Principle: Node.js Dependencies

■ Config

This principle states that configurations of an application need to be stored independently from the code itself, as environment variables, or in a configuration file.

Examples of configuration data are hostname, port number, and credentials. These configuration data are different for each of the deployment environments where the application is going to run. By separating such configuration data, we're making it easy to run the application in different environments by just applying the independently maintained configuration data to the runtime.

Figure 2.15 shows that when running in different landscapes, the same codebase is run, but different sets of configuration data are applied in different environments.

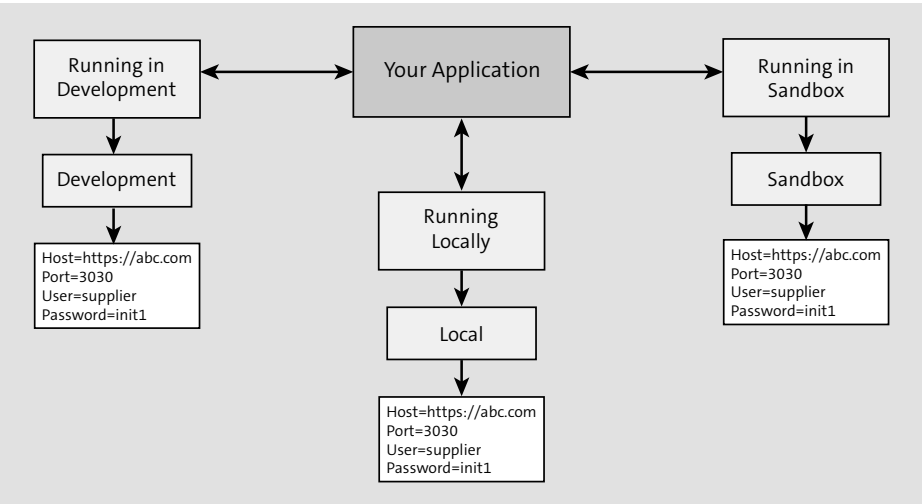


Figure 2.15 Same Application Code Running in Different Environments

■ Backing services

The backing service principle suggests that services such as databases, messaging systems, Simple Mail Transfer Protocol (SMTP) services, and so on, should be architected as external resources. The application consumes these backing services over the network.

These services can be locally managed or provided by third parties such as Amazon Simple Storage Service (Amazon S3) or Google Maps. URLs, credentials, and so on should be maintained in a configuration file, and when required to replace an existing service with another one, you can easily change the details in the configuration file. Each of these backing services is referred to as a resource. In Figure 2.16, you can see an example of the SAP Cloud Application Programming Model application accessing three different resources.

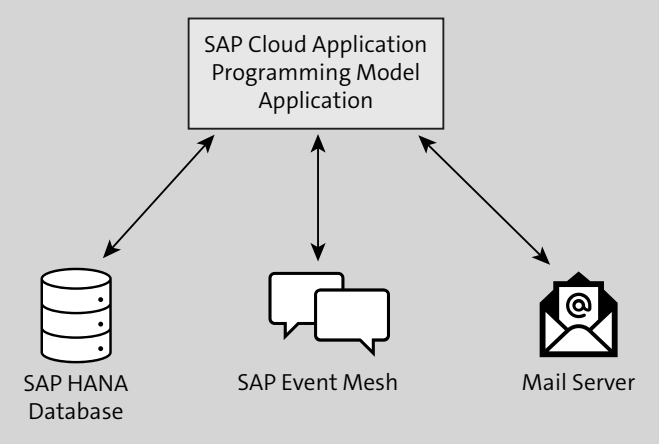


Figure 2.16 Twelve-Factor App Principle: SAP Backing Services



■ Build, release, and run

This principle states that there should be three independent steps in your deployment process (see Figure 2.17):

- The *build stage* converts the code repo into an executable bundle. At this stage, all the dependencies will also be fetched and compiled into the executable bundle.
- The *release stage* combines the configurations from configuration files and environment variables with the executable bundle. The resulting build will be an executable file ready to be run in the execution environment.
- The *run stage* runs the app in the execution environment, which can be development, quality, or production.

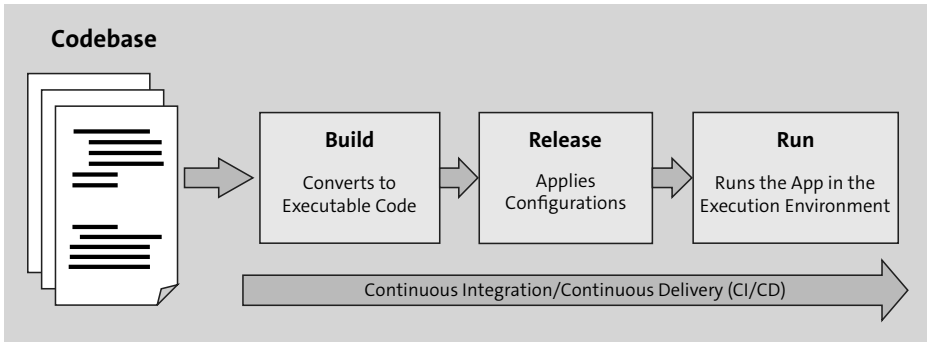


Figure 2.17 Twelve-Factor App Principle: Build, Release, Run

■ Stateless processes

This principle states that applications should have the provision to be served by multiple stateless, independent processes, as shown in Figure 2.18.

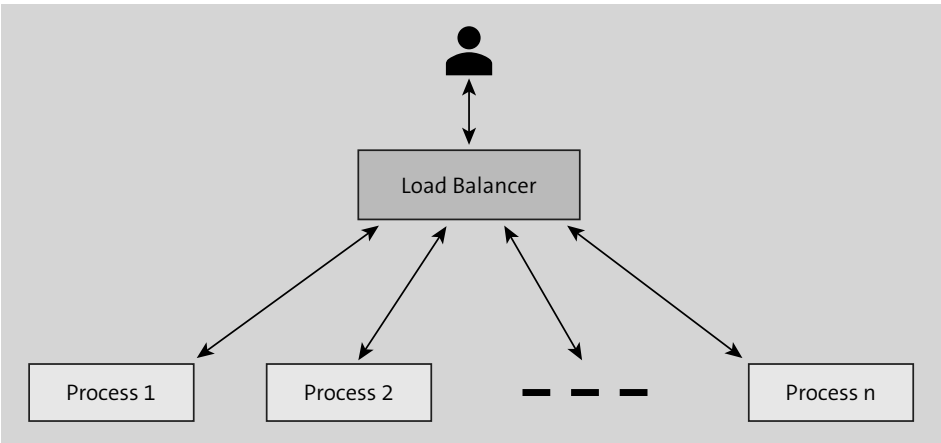


Figure 2.18 Stateless Processes Serving the Application

Consider that a user’s request is served by process 1. Subsequent requests should be able to be served by process 2 or any other processes as well. There will be no session data maintained in any of the processes, and each process independently serves the request without communicating with other processes. Any data that needs to persist must use a backing service such as a database.

Following this principle makes it easy to scale the infrastructure up and down, thus making it ideal for cloud deployments.

■ Port binding

The twelve-factor app is a self-contained standalone app that doesn’t require a web server to create a web-facing service. Instead of having a web server to handle the requests and sending to the individual services, where dependency with the web server is created, a twelve-factor app directly binds to a port and responds to incoming requests (see Figure 2.19).

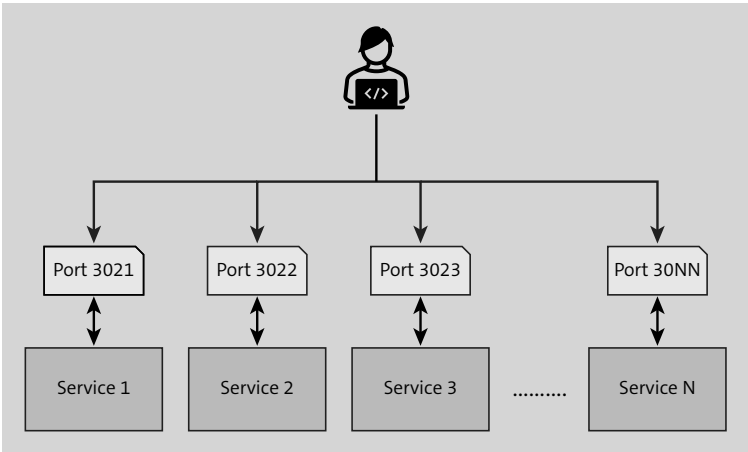


Figure 2.19 Port Binding

An individual service can act as a backing service to another app by providing the backing service URL in the configuration of the consumption app.

■ Concurrency

This principle states that the application needs to be broken down into multiple modules so that each of those modules can be scaled up and down independently. For instance, HTTP requests can be handled by a web process, and a worker process can take care of background jobs. Again, these individual processes can be scaled up or down to handle the increased workloads independently.

■ Disposability

A twelve-factor app should maximize robustness with fast startup and graceful shutdown. The processes should minimize the time to startup and ideally take a few seconds to start and receive the incoming requests; it helps while scaling up the processes. At the same time, the processes should shut down gracefully

and cease to listen on the service port without allowing any incoming requests; in such cases, if there is a queuing system, the requests can be queued and processed once the processes are up.

#### ■ DEV/PROD parity

The twelve-factor app methodology suggests that an app's development, staging, and production are kept as similar as possible. A twelve-factor app should be designed with the CI/CD approach by making the time gap small—where the developer writes some code and deploys it in hours or even minutes—and keeping DEV and PROD as similar as possible. This eliminates the risk of bugs in production when new changes are moved with different versions.

#### ■ Logs

This rule suggests treating logs as event streams. Logs are typically time-ordered event information, or logs can be error or success messages recorded by an app. A twelve-factor app never concerns itself with storing the log information in the app, as it can die and, as a result, lose the information. Instead, the app should treat log entries as event streams and use a separate service to save them. These can be consumed by interested parties to perform analytics or for monitoring.

#### ■ Admin processes

The developer often needs to perform administrative or maintenance activities for apps that need data migration, running processes, or one-time scripts. These should also be identical across different landscapes (DEV, QA, and PROD). These processes should also be shipped along with the application code to avoid synchronization issues.

## 2.5 Important Terminology

In this chapter, the following terminology was used:

#### ■ Application programming interface (API)

APIs enable companies to securely connect their internal applications or external partners to transfer data in both directions.

#### ■ JavaScript Object Notation (JSON)

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays.

#### ■ Open Data Protocol (OData)

OData is a REST-based protocol approved by ISO/IEC used for building and consuming RESTful APIs.

#### ■ Representational State Transfer (REST)

REST is a famous web API architecture that is mainly used for lightweight web services and mobile applications.

#### ■ Simple Object Access Protocol (SOAP)

SOAP is a stricter protocol with defined security rules that is standardized by the W3C and uses an XML payload to transfer the data between systems.

#### ■ Twelve-factor app

A twelve-factor app defines the principles for building performant, scalable, and resilient microservice-based cloud-native web applications.

#### ■ Yet Another Markup Language (YAML)

YAML is a human-friendly data serialization language. It's a superset of JSON, so JSON files are valid in YAML.

## 2.6 Practice Questions

- What is an application programming interface (API)?
  - ☐ A. Interface used for software applications to interact with each other
  - ☐ B. Tool for creating web applications
  - ☐ C. A software development kit of mobile applications
- Consumers of an API need which of the following?
  - ☐ A. Architectural design of an API
  - ☐ B. Server details
  - ☐ C. Database dump from the server
  - ☐ D. Documentation of the API
- What are the features of a RESTful API? (There are three correct answers.)
  - ☐ A. Client-server model
  - ☐ B. Stricter protocols
  - ☐ C. Statelessness
  - ☐ D. Uniform interface
- Which of the following is valid for the OData protocol? (There are two correct answers.)
  - ☐ A. Follow the new protocols instead of REST principles.
  - ☐ B. Support the extensions of API without breaking the client's functionality.
  - ☐ C. Act as a relational database.
  - ☐ D. Follow the REST principles.

5. Where can you find the top-level feeds and Entity Data Model (EDM) of an OData service? (There are two correct answers.)
- ☐ A. Annotations
  - ☐ B. Service document
  - ☐ C. Metadata
  - ☐ D. SOAP document
6. What can you find in the metadata document of an OData service? (There are three correct answers.)
- ☐ A. Entity definitions
  - ☐ B. Relations
  - ☐ C. Entity properties
  - ☐ D. Entity data (entries)
7. Which of the following URIs are correct for fetching a single resource? (There are two correct answers.)
- ☐ A. `/Customers(CustomerID = 'ALFKI')`
  - ☐ B. `/Customers?$top=1`
  - ☐ C. `/Customers?$select=CustomerID`
  - ☐ D. `/Customers?$count`
8. Which URI option will we use to fetch the parent and the associated entities together? (There are two correct answers.)
- ☐ A. `/Customers('ALFKI')/Orders`
  - ☐ B. `/Customers('ALFKI')?$expand=Orders`
  - ☐ C. `/Customers?$expand=Orders`
  - ☐ D. `/Customers('ALFKI')/Orders?$select=CustomerId`
9. Which functionalities are used by OData to implement pagination in frontend applications?
- ☐ A. `$top` and `$select`
  - ☐ B. `$top`, `$count`, and `$expand`
  - ☐ C. `$top`, `$skip`, and `$inlinecount`
  - ☐ D. `$top`, `$skip`, and `$select`

10. What is the relationship between YAML and JSON?
- ☐ A. YAML is a superset of JSON.
  - ☐ B. YAML is a subset of JSON.
  - ☐ C. YAML and JSON are the same.
11. For what is YAML primarily used?
- ☐ A. Creating configuration files
  - ☐ B. Transferring data
  - ☐ C. Storing data in the database
12. Which of the following supports nonhierarchical data?
- ☐ A. JSON
  - ☐ B. YAML
  - ☐ C. Both JSON and YAML
13. Which data format supports comments?
- ☐ A. YAML
  - ☐ B. JSON
  - ☐ C. Neither of them
14. Which of the following is true for JSON?
- ☐ A. It supports indentation.
  - ☐ B. Arrays should begin with a bracket (`[]`).
  - ☐ C. YAML can be used inside JSON.
15. The codebase principle of the twelve-factor app suggests which of the following? (There are two correct answers.)
- ☐ A. Code should be shared using libraries and the dependency manager.
  - ☐ B. Version/source control isn't mandatory for codebases.
  - ☐ C. The CI/CD approach can be used to keep the systems codebase in sync.
16. Where can configurations such as credentials, host, or port be saved?
- ☐ A. Dependencies
  - ☐ B. Code
  - ☐ C. Environment variables

17. Which of the following statements are true? (There are two correct answers.)
- ☐ A. Applications should access backing services using the configuration file.
  - ☐ B. A backing service is a library that can be reused in different codebases.
  - ☐ C. SAP HANA database and SAP Event Mesh are called backing services.
18. What principle should be followed for deploying the codebase to a production environment?
- ☐ A. Backing services
  - ☐ B. Build, release, and run
  - ☐ C. Dependencies
19. The stateless principle from the twelve-factor app suggests which of the following? (There are two correct answers.)
- ☐ A. Stateless independent processes
  - ☐ B. Dependent processes
  - ☐ C. Use of backing services for persistency
20. Which one of the following is true in the case of a twelve-factor app?
- ☐ A. Use ports to listen to incoming requests instead of a web server directly.
  - ☐ B. The application should be created from multiple modules that can be scaled up and down independently.
  - ☐ C. Development, staging, and production environments should be kept similar with the help of the CI/CD approach.
  - ☐ D. All of the above are true.
21. Which of the following is true for a twelve-factor app?
- ☐ A. Logs should be stored in the application.
  - ☐ B. Admin processes should be similar in all the landscapes.
  - ☐ C. Neither of the above are true.

## 2.7 Practice Question Answers and Explanations

1. Answer: A  
APIs are used for software applications to interact with each other. An API isn't a tool for creating web applications or an SDK for developing mobile apps.

2. Answer: D  
While designing an API, API producers should hide the complexity of what an API does in the background and only provide the information the consumer requests. API producers will give the consumers the documentation of an API explicitly or implicitly (via metadata) to understand how the API works.
3. Answer: A, C, and D  
RESTful APIs follow the REST guidelines such as client-server model, cacheability, uniform interface, statelessness, and the layered system, but they don't follow any strict protocols like SOAP does.
4. Answer: B and D  
OData services follow the REST principles, and they should support the extension of APIs without breaking the client's functionality.
5. Answer: B and C  
Service documents have all the top-level feeds of an OData service, whereas the EDM can be determined from the metadata document.
6. Answer: A, B, and C  
You can find entity definitions, properties, key properties, and related entities (relationships) in the metadata. You need to pass the entity set name in the resource path to access the entity data.
7. Answer: A and B  
`/Customers(CustomerID = 'ALFKI')` returns only one record matching the key property ALFKI and `/Customers?$top=1` returns up to one customer entry as the `$top=1` query option is used.
8. Answer: B and C  
`$expand` is used to fetch the parent entity and the mentioned associated entities in `$expand`.
9. Answer: C  
The `$top`, `$skip`, and `$inlinecount` or `$count` query options of an OData service are used to implement pagination in frontend applications such as SAPUI5.
10. Answer: A  
YAML is the superset of JSON. A valid YAML file can contain JSON. You can convert JSON to YAML without any loss of information, and every JSON file is also a valid YAML file.
11. Answer: A  
As YAML is more human-readable than JSON, it's preferred in creating configuration files.
12. Answer: B  
YAML supports nonhierarchical data and doesn't need to follow the typical parent and child relationship like the JSON model.



13. Answer: **A**

YAML supports comments by default.

14. Answer: **B**

Arrays should begin with a bracket ([ ]). YAML supports indentation to segregate data inside it, and JSON can be used inside YAML, but not otherwise. JSON arrays begin and end with brackets.

15. Answer: **A and C**

Reusable code can be shared using libraries with the help of a dependency manager. Codebase should have version control such as Git to track changes and push to different landscapes. CI/CD can keep the systems in sync with the latest codebase and patches, which is the recommended approach per the twelve-factor app principles.

16. Answer: **C**

We shouldn't save the data such as host, port, or credentials in the code; it should be saved in the environment variables.

17. Answer: **A and C**

The application should access backing services such as SAP HANA database or SAP Event Mesh via the environment variable or configuration files. Libraries are dependencies, not backing services.

18. Answer: **B**

The build, release, and run principle should be followed when deploying the codebase to a production environment. This principle states that there should be three independent steps in your deployment process: the build stage that converts the code repo into an executable bundle; the release stage that combines the configurations from configuration files and environment variables with the executable bundle; and the run stage that runs the app in the execution environment.

19. Answer: **A and C**

The twelve-factor app principle suggests that stateless and independent processes should serve an application, and if there is any need for data persistency, a backing service should be used.

20. Answer: **D**

The port binding principle suggests that instead of using a webserver to handle the requests, the application should listen to the ports to handle the incoming requests directly.

The concurrency principle suggests that an application should be broken down into multiple modules to be scaled up and down independently based on load.

The DEV/PROD parity principle requires all the environments to be similar and use CI/CD tools to reduce the time and have the application identical in all the environments.

21. Answer: **B**

Logs should not be stored in the application, and external services should be used to save the logs. Administrative or maintenance activities for apps such as migrations or on-time scripts should be similar in all the landscapes.

## 2.8 Test Takeaway

OData is a REST-based protocol for building RESTful APIs. SAP recommends using OData in its applications and frameworks, such as SAP Cloud Application Programming Model (Node.js) or RAP (ABAP), which have huge support for OData. JSON and YAML are the most popular data formats used in configuration files and while transferring the data across the web. Twelve-factor app principles are the widely endorsed principles that are recommended to be followed while designing an application with modern web capabilities.

## 2.9 Summary

In this chapter, you learned about important web standards such as API, REST, and OData. Then we covered YAML and JSON, two popular formats for data exchange and configuration files. We discussed how to create resilient cloud-native applications by understanding the principles of the twelve-factor app as well. In addition, we discussed the OData protocol in depth, which is a popular format used extensively in SAP applications.

In the next chapter, we'll discuss the the SAP Cloud Application Programming Model.

# Contents

Foreword .....	13
Introduction .....	15
<b>1 SAP Business Technology Platform</b> .....	<b>25</b>
<b>1.1 Objectives of This Portion of the Test</b> .....	<b>26</b>
<b>1.2 Cloud Computing</b> .....	<b>26</b>
1.2.1 Types of Cloud Computing .....	27
1.2.2 Cloud Computing Models .....	28
<b>1.3 SAP Business Technology Platform</b> .....	<b>30</b>
1.3.1 The Uniqueness of SAP Business Technology Platform .....	30
1.3.2 SAP Business Technology Platform Capabilities .....	31
1.3.3 SAP Business Technology Platform Environments .....	35
1.3.4 SAP Business Technology Platform Cockpit .....	37
<b>1.4 SAP Business Technology Platform Account Model</b> .....	<b>38</b>
1.4.1 Feature Sets .....	38
1.4.2 Account Model .....	38
<b>1.5 SAP Business Technology Platform Commercial Models</b> .....	<b>43</b>
1.5.1 Consumption-Based Commercial Model .....	43
1.5.2 Subscription-Based Commercial Model .....	44
1.5.3 Free-Tier .....	45
1.5.4 Trial Account .....	45
1.5.5 Entitlements, Service Plans, and Quotas .....	46
<b>1.6 Cloud Foundry</b> .....	<b>47</b>
1.6.1 Architecture .....	48
1.6.2 Cloud Foundry Layers .....	49
1.6.3 Enabling Cloud Foundry in SAP Business Technology Platform .....	50
1.6.4 Cloud Foundry Command-Line Interface .....	51
1.6.5 Scaling a Cloud Foundry Application .....	51
1.6.6 Past and Future of the Cloud Foundry Platform .....	53
<b>1.7 SAP Discovery Center</b> .....	<b>54</b>
<b>1.8 Important Terminology</b> .....	<b>55</b>
<b>1.9 Practice Questions</b> .....	<b>57</b>
<b>1.10 Practice Question Answers and Explanations</b> .....	<b>59</b>

- 1.11 Test Takeaway** ..... 61
  - 1.12 Summary** ..... 61
- 2 Web Development Standards** ..... 63
  - 2.1 Objectives of This Portion of the Test** ..... 64
  - 2.2 Application Programming Interface, Representational State Transfer, and Open Data** ..... 64
    - 2.2.1 API ..... 64
    - 2.2.2 REST ..... 65
    - 2.2.3 OData ..... 67
  - 2.3 JavaScript Object Notation and Yet Another Markup Language** ..... 77
    - 2.3.1 JSON ..... 77
    - 2.3.2 YAML ..... 79
    - 2.3.3 YAML versus JSON ..... 80
  - 2.4 Twelve-Factor App Principles** ..... 81
  - 2.5 Important Terminology** ..... 86
  - 2.6 Practice Questions** ..... 87
  - 2.7 Practice Question Answers and Explanations** ..... 90
  - 2.8 Test Takeaway** ..... 93
  - 2.9 Summary** ..... 93
- 3 SAP Cloud Application Programming Model** ..... 95
  - 3.1 Objectives of This Portion of the Test** ..... 96
  - 3.2 SAP Business Application Studio** ..... 97
    - 3.2.1 Technical Details ..... 97
    - 3.2.2 Getting Started with SAP Business Application Studio ..... 98
  - 3.3 Introduction to the SAP Cloud Application Programming Model** ..... 103
    - 3.3.1 Capabilities of the SAP Cloud Application Programming Model ..... 104
    - 3.3.2 Real-World Scenario ..... 106
    - 3.3.3 Following the Progress in the Real-World Scenario ..... 108
  - 3.4 Creating an SAP Cloud Application Programming Model Project** ..... 108
  - 3.5 Domain Modeling** ..... 111
    - 3.5.1 Entities ..... 112

- 3.5.2 Types ..... 113
  - 3.5.3 Aspects ..... 113
  - 3.5.4 Code List ..... 115
  - 3.5.5 Views and Projections ..... 116
  - 3.5.6 Associations ..... 116
  - 3.5.7 Unmanaged and Managed Associations ..... 117
  - 3.5.8 Compositions ..... 117
  - 3.5.9 Actions and Functions ..... 118
  - 3.5.10 Real-World Scenario: Domain Model ..... 119
  - 3.5.11 Core Data Services Language to Core Schema Notation ..... 120
  - 3.5.12 Database interactions ..... 121
- 3.6 Creating OData Services** ..... 122
  - 3.6.1 Service Modeling ..... 122
  - 3.6.2 Real-World Scenario: Create a Service ..... 124
- 3.7 Running Locally** ..... 125
  - 3.7.1 Running the SAP Cloud Application Programming Model Project ..... 125
  - 3.7.2 Adding Test Data ..... 126
  - 3.7.3 Connecting to SAP HANA Cloud ..... 128
  - 3.7.4 CDS Bind Command ..... 132
- 3.8 Custom Handlers** ..... 133
  - 3.8.1 Writing Custom Handlers ..... 133
  - 3.8.2 Application Programming Interface for Handler Registration ..... 134
  - 3.8.3 Explicit Way of Registering Event Handlers ..... 134
  - 3.8.4 Phases of Events ..... 135
  - 3.8.5 Real-World Scenario: Creating Custom Handlers ..... 137
- 3.9 Emitting and Subscribing to Events** ..... 138
- 3.10 Building and Deploying** ..... 140
  - 3.10.1 Cloud Foundry Native Deployment ..... 140
  - 3.10.2 Multi-Target Applications ..... 143
- 3.11 Annotations** ..... 146
- 3.12 Important Terminology** ..... 148
- 3.13 Practice Questions** ..... 149
- 3.14 Practice Question Answers and Explanations** ..... 152
- 3.15 Test Takeaway** ..... 154
- 3.16 Summary** ..... 155

<b>4</b>	<b>Connectivity</b>	157
4.1	Objectives of This Portion of the Test	158
4.2	Consuming External OData Services	158
4.3	Destination Service	165
4.3.1	Creating a Destination	165
4.3.2	Manually Creating and Binding SAP BTP Service Instances	167
4.3.3	Creating and Binding the Service via the Terminal	168
4.3.4	Creating and Binding the Service via mta.yaml	169
4.3.5	Running the Service Locally	171
4.3.6	Running on SAP Business Technology Platform	173
4.4	Cloud Connector	173
4.4.1	Installing the Cloud Connector	175
4.4.2	Configuring the Cloud Connector	175
4.5	Connectivity Service	179
4.6	Advanced Concepts	181
4.6.1	Sending the Application Programming Interface Key	181
4.6.2	Advanced Custom Handler	183
4.6.3	Application Details	184
4.7	Important Terminology	185
4.8	Practice Questions	186
4.9	Practice Question Answers and Explanations	187
4.10	Test Takeaway	188
4.11	Summary	189
<b>5</b>	<b>SAP Fiori Elements</b>	191
5.1	Objectives of This Portion of the Test	192
5.2	SAPUI5	192
5.3	SAP Fiori	194
5.4	SAP Fiori Elements	201
5.4.1	Generating the SAP Fiori Elements Application	202
5.4.2	Enabling the Draft Functionality	208
5.4.3	Adding Annotations	210
5.5	Important Terminology	224
5.6	Practice Questions	224

5.7	Practice Question Answers and Explanations	227
5.8	Test Takeaway	228
5.9	Summary	228
<b>6</b>	<b>Authorization and Trust Management</b>	229
6.1	Objectives of This Portion of the Test	230
6.2	AppRouter	231
6.2.1	Challenges with the Microservices Approach	231
6.2.2	Options to Add the AppRouter Module	232
6.2.3	Configuring AppRouter	233
6.2.4	Configuring the HTML5 Application Repository Service	236
6.2.5	Real-World Scenario: Adding an AppRouter Module	237
6.3	Authentication and Trust Management	241
6.3.1	Identity Provider at the Subaccount Level	241
6.3.2	Authentication Strategies	242
6.3.3	SAP Authorization and Trust Management Service	245
6.3.4	JSON Web Token	246
6.3.5	Types of Users	248
6.3.6	Real-World Scenario Adding Authentication	249
6.4	Authorization	251
6.4.1	Role Collections, Roles, Scopes, Attributes	251
6.4.2	Application Security Descriptor: xs-security.json	252
6.4.3	Assigning and Enforcing Authorizations	253
6.4.4	Real-world Scenario: Adding Authorizations	260
6.5	Important Terminology	263
6.6	Practice Questions	264
6.7	Practice Question Answers and Explanations	266
6.8	Test Takeaway	268
6.9	Summary	268
<b>7</b>	<b>Continuous Integration and Delivery</b>	271
7.1	Objectives of This Portion of the Test	273
7.2	Continuous Integration, Delivery, and Deployment	273
7.2.1	Continuous Integration	273



7.2.2	Continuous Delivery .....	274
7.2.3	Continuous Deployment .....	274
<b>7.3</b>	<b>Configuring the CI/CD Pipeline .....</b>	<b>275</b>
7.3.1	Unit Testing .....	276
7.3.2	Git Repository .....	279
7.3.3	Configure CI/CD with SAP Continuous Integration and Delivery Service .....	281
<b>7.4</b>	<b>Important Terminology .....</b>	<b>293</b>
<b>7.5</b>	<b>Practice Questions .....</b>	<b>294</b>
<b>7.6</b>	<b>Practice Question Answers and Explanations .....</b>	<b>297</b>
<b>7.7</b>	<b>Test Takeaway .....</b>	<b>298</b>
<b>7.8</b>	<b>Summary .....</b>	<b>299</b>
The Authors .....		301
Index .....		303

# Index

.cdsrc.json ..... 110  
.cdsrc-private.json ..... 171  
.yaml ..... 79  
.yml ..... 79

## A

ABAP ..... 35  
ABAP RESTful Application Programming  
    Model ..... 67  
Admin processes ..... 86  
Alibaba Cloud ..... 41  
Amazon Simple Storage Service  
    (Amazon S3) ..... 83  
Amazon Web Services (AWS) ..... 28, 35, 41, 48  
Analytics ..... 32  
Annotations ..... 146, 201, 224  
Apache Tomcat ..... 174  
App ..... 110  
Application  
    *run locally* ..... 125  
Application Autoscaler ..... 53  
Application connector ..... 36  
Application development ..... 33  
Application integration ..... 33  
Application programming interface  
    (API) ..... 64, 86, 243  
    *purchase order* ..... 106  
    *query builder* ..... 104  
    *RESTful* ..... 66  
Application security descriptor ..... 252  
AppRouter ..... 110, 231, 263  
AppRouter configuration ..... 238  
AppRouter module ..... 237  
Association ..... 116  
Asynchronous communication ..... 139  
AtomPub ..... 67  
Atos Cloud Foundry ..... 48  
Attribute ..... 252  
Authentication ..... 241  
    *method* ..... 233  
    *vs. authorization* ..... 263  
Authorization ..... 241  
    *assign* ..... 258  
    *declaritive enforcement* ..... 253  
    *programmatic enforcement* ..... 257  
Authorization role ..... 263  
Authorization scope ..... 264  
AWS Elastic Beanstalk ..... 30  
Axios ..... 277

## B

Backing services ..... 83  
Basic authentication ..... 105, 243  
Binding ..... 166  
Blob store ..... 48  
BOSH ..... 48  
Bound ..... 118  
Build stage ..... 84  
Build, release, and run ..... 84  
build.gradle ..... 82  
Business user ..... 42, 264

## C

Cacheability ..... 66  
Cascading Style Sheet (CSS) ..... 192  
CDS Bind Command ..... 132  
CDS Definition Language (CDL) ..... 104  
CDS Query Language (CQL) ..... 104, 121  
cf deploy ..... 51  
cf login ..... 51  
cf push ..... 51  
Chai ..... 276, 293  
Client (CLI) tool ..... 51  
Client-server architecture ..... 66  
Cloud computing ..... 26  
    *models* ..... 28  
Cloud connector ..... 185  
Cloud Controller ..... 48  
Cloud credits ..... 43  
Cloud Foundry ..... 35, 47, 55  
    *architecture* ..... 48  
    *command-line interface* ..... 51  
    *enable in SAP Business Technology*  
        *Platform* ..... 50  
    *layers* ..... 49  
    *native deployment* ..... 140  
    *scalability* ..... 51  
Cloud MTA Build Tool (MBT) ..... 145  
Cloud Platform Enterprise Agreement  
    (CPEA) ..... 43, 55  
Cloud.gov ..... 48  
Cloud-native ..... 55  
Code list ..... 115  
Code on demand ..... 66  
Codebase ..... 81  
Command-line interface (CLI) ..... 98, 249  
Comma-separated values (CSV) ..... 127  
Concurrency ..... 85

Config ..... 82

Configuration variables ..... 143

Consumption-based commercial  
  model ..... 43, 55

Continuous delivery ..... 274

Continuous deployment ..... 274

Continuous integration ..... 272, 273

Continuous integration and continuous  
  delivery/deployment (CI/CD) ..... 81, 86, 272,  
  294

*pipeline* ..... 275, 294

Core Schema Notation (CSN) ..... 120, 160

Coredata services (CDS) ..... 101, 243

Create, read, update, and delete (CRUD) .... 118

Create-Service-Push ..... 141

Cross-origin request sharing  
  (CORS) ..... 231, 264

Custom handlers ..... 133, 148

D

Data Definition Language (DDL) ..... 120

Data integrations ..... 193

Data loss ..... 208

Database ..... 32

Database management ..... 32

db ..... 110

Decoupling ..... 139

Demilitarized zone (DMZ) ..... 174

Denial of service (DoS) ..... 173

Dependencies ..... 82

Destination ..... 165

Dev spaces ..... 101

DEV/PROD parity ..... 86

Diego ..... 48, 55

Disposability ..... 85

Document Classification ..... 33

Domain modeling ..... 111, 148

Draft ..... 208, 224

*edit* ..... 208

Dummy authentication ..... 243

E

Eclipse Theia ..... 97

EDMX ..... 159

EDMX file ..... 185

Emitting ..... 138

Enterprise microservices ..... 231

Entitlement ..... 47

Entity ..... 68

*set* ..... 68

*type* ..... 68

Entity Data Model (EDM) ..... 68

escalationsfe.zip folder ..... 206

ETag ..... 124

Event handler ..... 134

Event phase ..... 135

Exam objective ..... 16

Exam structure ..... 16

External systems ..... 160

F

Facet annotations ..... 213

Feature set ..... 38

Field group ..... 213

Free tier model ..... 45

Full-Stack Cloud Application ..... 101

G

Garden ..... 48, 55

Gartner 2021 Magic Quadrant for  
  Multiexperience Development  
    Platforms ..... 33

Gartner Magic Quadrant for Enterprise  
  Integration Platform as a Service ..... 34

Generic Application Content  
  Deployer (GACD) ..... 206

Git ..... 81, 272, 294

Git repository ..... 103, 273, 279

GitHub ..... 29, 109, 279, 294

Gmail ..... 29

Google App Engine ..... 30

Google Cloud ..... 28, 35, 41, 48

Google Maps ..... 83

Gorouter ..... 48

Gradle dependency manager ..... 82

H

Header ..... 247

Horizontal scaling ..... 52

HTML5 ..... 192

HTML5 ..... 35, 194

*application repository service* ..... 236

HTTP ..... 49

Hybrid cloud ..... 28, 55

I

IBM Bluemix ..... 30

IBM Cloud ..... 28

IBM Cloud Foundry ..... 47

Identity Authentication ..... 241

Identity federation ..... 241

Identity provider (IdP) ..... 232

Infrastructure-as-a-service (IaaS) ..... 28, 56

Installable version ..... 175

Integrated development environment  
  (IDE) ..... 34, 97

Intelligent technologies ..... 32

International Organization for  
  Standardization/International Electro-  
  technical Commission (ISO/IEC) ..... 67

Invoice Object Recommendation ..... 33

J

Java ..... 35, 82

JavaScript ..... 77, 192

JavaScript Object Notation (JSON) ... 64, 67, 77,  
  80, 86, 233

*array* ..... 78

*object* ..... 77

*structure* ..... 77

Jenkins ..... 275

Jest ..... 276, 294

JSON Web Token (JWT) ..... 105, 166, 241,  
  246, 264

*authentication* ..... 244

K

Kubernetes ..... 35

Kubernetes Pod ..... 101

Kyma ..... 56

Kyma project ..... 36

L

Layered system ..... 66

Lightweight Directory Access Protocol  
  (LDAP) ..... 177

Locking ..... 209

Logs ..... 86

M

makefile.mta ..... 145

manifest.json file ..... 239

Member management ..... 42

Metadata ..... 201

Micro apps ..... 143

Microservices ..... 231

Microsoft Azure ..... 28, 30, 35, 41, 48

Microsoft Visual Studio Code (VS Code) ..... 97

Mission ..... 54

Mobile development kit (MDK) ..... 101

Mocha ..... 276, 294

Modules ..... 143

Monaco Editor ..... 97

MTA Tools ..... 101

mta.yaml ..... 143

MultiApp ..... 51, 146

Multi-target application (MTA) ..... 51, 143, 169

Multitenancy scenario ..... 232

N

Neural Autonomic Transport System  
  (NATS) ..... 49

Node package manager (NPM) ..... 82

Node.js ..... 82, 239

O

OAuth server ..... 48

Object-relational mapping (ORM) ..... 104

OData V2 ..... 67

OData V4 ..... 67

Open Connectors ..... 34

Open Data Protocol (OData) ..... 64, 67, 86

Open VSX Registry ..... 98

openSAP ..... 19

Organization ..... 49

Outlook ..... 29

P

package.json ..... 82, 110

package.json folder ..... 206

Parameter ..... 145

Pay-As-You-Go for SAP BTP ..... 44, 56

Payload ..... 247

Personal Access Tokens (PAT) ..... 280

Piper ..... 275

Platform user ..... 42, 264

Platform-as-a-service (PaaS) ..... 29, 56, 81

Port binding ..... 85

Portable version ..... 175

Private cloud ..... 27, 56

Properties ..... 144

Pseudo role ..... 252

Public cloud ..... 27, 56

Publish-subscribe ..... 138

Q

Query ..... 71

R

Red Hat OpenShift ..... 30

Release stage ..... 84

Remote function call (RFC) ..... 177

Representational State Transfer (REST) ..... 64, 65, 86

Resiliency ..... 139

Resource ..... 143

*path* ..... 69

*server* ..... 241

Role ..... 252

*collection* ..... 252, 264

*template* ..... 252

role-collections ..... 253

role-templates ..... 253

Route ..... 233

Run stage ..... 84

S

Salesforce ..... 29

Salesforce Platform ..... 30

SAP AI Core ..... 33

SAP Analytics Cloud ..... 32

SAP API Business Hub ..... 158, 159

*sandbox* ..... 179

SAP API Management ..... 35

SAP AppGyver ..... 34

SAP Application Logging service

    for SAP BTP ..... 46

SAP Authorization and Trust

    Management service ..... 166, 241, 245

*authentication* ..... 245

SAP BTP cockpit ..... 37, 39, 42

SAP BTP connectivity service ..... 185

SAP BTP destination service ..... 166, 185

SAP BTP, ABAP environment ..... 36

SAP BTP, Cloud Foundry

    environment ..... 35, 36

SAP BTP, Kyma runtime ..... 36

SAP BTP, Neo environment ..... 35

SAP Business API Hub ..... 163

SAP Business Application Studio ..... 34, 42, 96, 97, 148, 201, 280

SAP Business Technology Platform

    (SAP BTP) ..... 26, 30, 56, 67, 273

*account model* ..... 38

*capabilities* ..... 31

*commercial model* ..... 43

*directory* ..... 39

*enterprise account* ..... 39

*features* ..... 30

*global account* ..... 38

*multi-cloud* ..... 35

*pillars* ..... 32

*subaccount* ..... 40

*subaccount region* ..... 41

SAP Business Technology Platform (SAP BTP)

    (Cont.)

*trial account* ..... 39

*users* ..... 42

SAP Cloud Application Programming

    Model ..... 83, 96, 103, 148, 158, 273

SAP Cloud Transport Management ... 275, 281

SAP Community ..... 21

SAP Continuous Integration and

    Delivery ..... 273, 281

SAP Conversational AI ..... 34

SAP Discovery Center ..... 54

*missions* ..... 20

SAP ERP ..... 158

SAP Event Mesh ..... 35, 105

SAP Extension Suite ..... 33, 106, 281

*exam* ..... 15

SAP Fiori ..... 72, 97, 101, 194, 224

    1.0 ..... 196

    2.0 ..... 196

    3.0 ..... 196

*design principles* ..... 195

*floorplan* ..... 224

*user interface* ..... 236

SAP Fiori elements ..... 201, 224

*generate application* ..... 202

SAP HANA Academy ..... 18

SAP HANA Cloud ..... 128, 141

SAP HANA extended application

    services ..... 35, 101, 145

SAP HANA Native Application ..... 101

SAP Help ..... 22

SAP Integration Suite ..... 34

SAP Launchpad ..... 34

SAP Learning Hub ..... 18

SAP Mobile Application ..... 101

SAP Mobile Services ..... 34

SAP NetWeaver Application Server

    for Java ..... 177

SAP S/4HANA ..... 28, 36, 67, 96, 105, 106, 119, 158, 163, 177, 194

SAP S/4HANA Cloud ..... 29, 36, 106

SAP SuccessFactors ..... 29, 158, 194

SAP tutorials for developers ..... 19

SAP UI Vocabulary ..... 211

SAP Web Analytics ..... 32

SAP Web IDE ..... 97

SAP Work Zone ..... 34

SAPUI5 ..... 67, 72, 192, 194, 224

*features* ..... 193

Scope ..... 251

scopes ..... 253

Security Assertion Markup Language

    (SAML) ..... 242, 245

Security descriptor ..... 264

Serve static file ..... 232

Service ..... 54

*broker* ..... 48

*definition* ..... 122

*document* ..... 68

*metadata document* ..... 68

*modeling* ..... 122

*plan* ..... 45, 46

Service root URI ..... 67

service\_annotaion.cds ..... 210

Signature ..... 247

Simple Mail Transfer Protocol (SMTP) ..... 83

Simple Object Access Protocol (SOAP) ... 64, 87

Single sign-on (SSO) ..... 232

Software-as-a-service (SaaS) ..... 29, 56, 97

Space ..... 49

SQLite ..... 125, 149

srv ..... 110

Stateless processes ..... 84

Statelessness ..... 66

Steampunk ..... 36

Subscription-based commercial

    model ..... 44, 56

Subversion ..... 81

SUSE Cloud Application Platform ..... 48

Swisscom Application Cloud ..... 48

T

tenant-mode ..... 253

Test preparation resources ..... 18

Test-taking strategies ..... 22

Thing perspective ..... 214

Transport layer security (TLS) ..... 174

Trial account ..... 45

Twelve-factor app principles ..... 81, 87

Type ..... 113

U

ui5-deploy.yaml ..... 206

Uniform interface ..... 66

URL rewriting ..... 235

User Account and Authorization (UAA) ..... 166

User experience (UX) ..... 193

User interface (UI) ..... 35, 275

*static* ..... 238

User management ..... 42

UUID key ..... 114

V

Vertical scaling ..... 52

View ..... 116

VMware Tanzu ..... 48

VMware Tanzu Application Service ..... 30

VS Code ..... 201

W

Web development standards ..... 64

webapp folder ..... 206

welcomeFile ..... 233

Windows PowerShell ..... 168

Workday ..... 29

World Wide Web Consortium (W3C) ..... 65

X

xsappname ..... 253

XSUAA instance ..... 238, 240, 249, 250, 252

Y

Yahoo! mail ..... 29

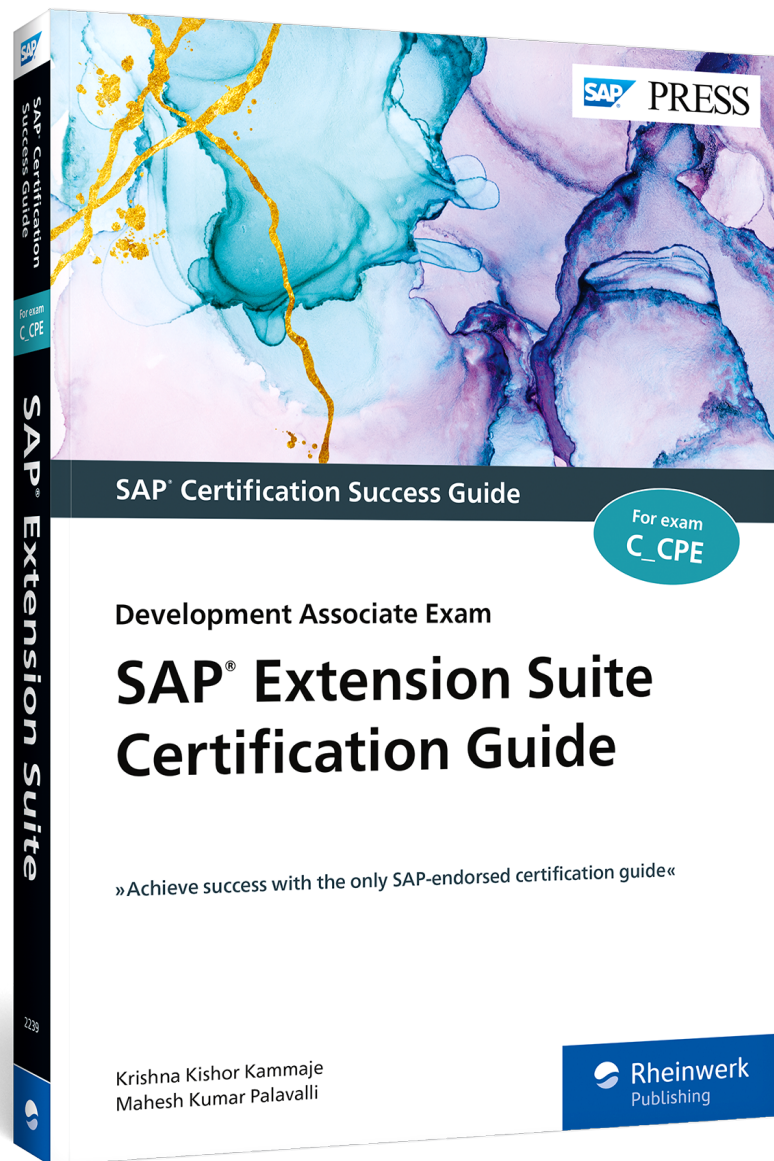
Yeoman generator ..... 103

Yet Another Markup Language

    (YAML) ..... 64, 79, 80, 87

Z

Zoho ..... 29



Krishna Kishor Kammaje, Mahesh Kumar Palavalli

## SAP® Extension Suite Certification Guide: Development Associate Exam

307 pages, 2023, \$79.95  
ISBN 978-1-4932-2239-1

 [www.sap-press.com/5490](https://www.sap-press.com/5490)



**Krishna Kishor Kammaje** is a passionate developer and application architect working at ConvergentIS. He is a recognized SAP Community contributor and was named as an SAP Mentor. He is also an author of the book *SAP Fiori Certification Guide*. His latest interests are in SAP Business Technology Platform, cloud computing, machine learning, and teaching.



**Mahesh Kumar Palavalli** is a senior developer at SAP Labs in Bangalore. He has more than 9 years of experience working with customers from government and private sectors in the areas of SAP Fiori, SAP Business Technology Platform, and ABAP. He was also recognized as a top contributor and a member of the month by SAP Community.

*We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.*