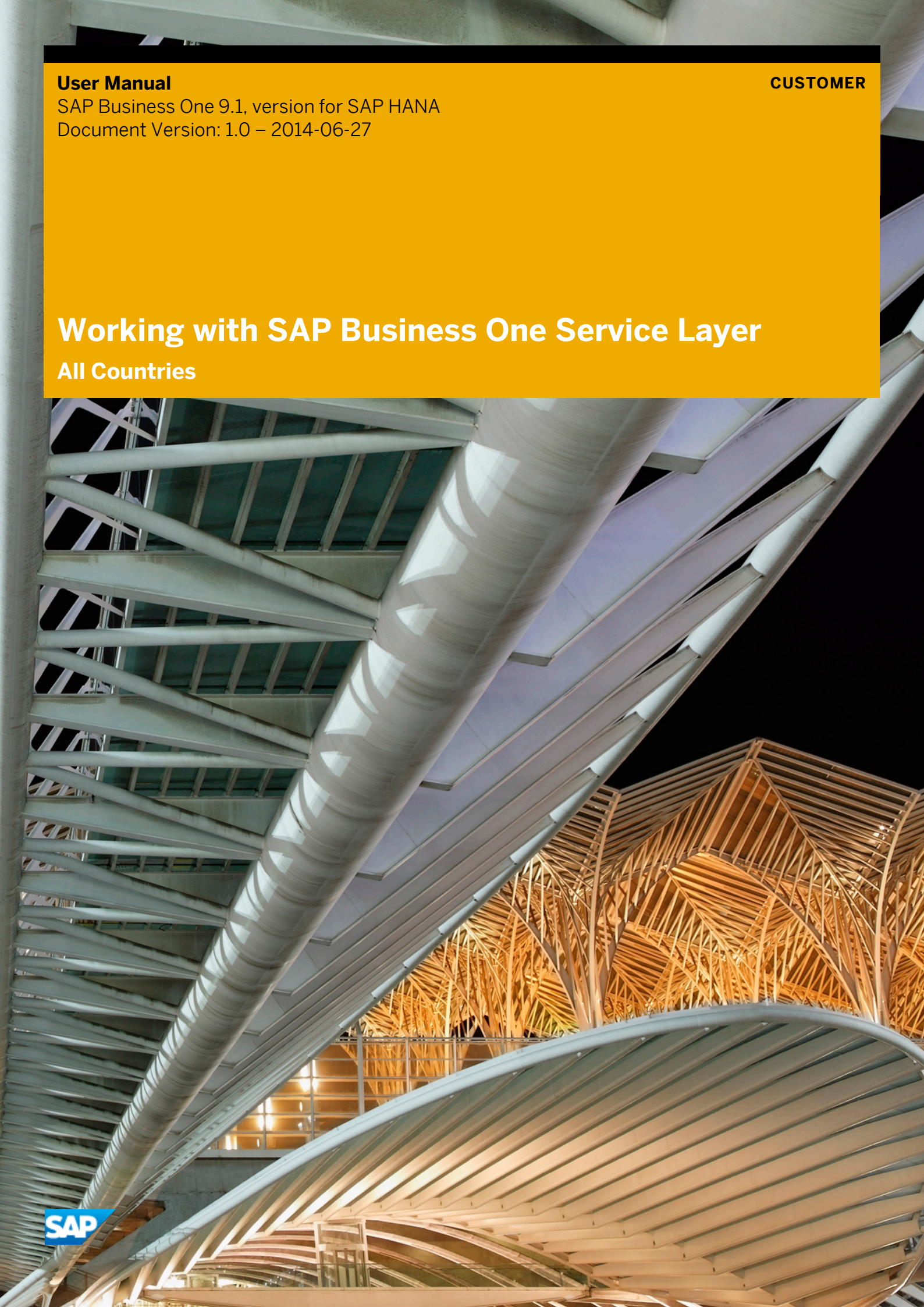
User Manual
SAP Business One 9.1, version for SAP HANA
Document Version: 1.0 – 2014-06-27

CUSTOMER

Working with SAP Business One Service Layer

All Countries



Typographic Conventions

Type Style	Description
Example	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Textual cross-references to other documents.
Example	Emphasized words or expressions.
`EXAMPLE`	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
EXAMPLE	Keys on the keyboard, for example, F2 or ENTER.

Document History

Version	Date	Change
1.0	2014-06-27	The first release of SAP Business One Service Layer

Table of Contents

1 Introduction

1.1 About This Document

This document covers the basic API usage of SAP Business One Service Layer and explains the technical details of building a stable, scalable Web service using SAP Business One Service Layer.

1.2 Target Audience

We recommend that you refer to this document if you are:

- Developing applications based on Service Layer API
- Planning your first load balancing deployment
- Improving your system's performance
- Assuring your system's stability under heavy work load

This document is intended for system administrators who are responsible for configuring, managing, and maintaining an SAP Business One Service Layer installation. Familiarity with your operating system and your network environment is beneficial, as is a general understanding of web application server management.

This document is also relevant for software developers who build add-ons for SAP Business One.

1.3 About SAP Business One Service Layer

SAP Business One Service Layer is a new generation of extension API for consuming SAP Business One data and services. It builds on core protocols such as HTTP and OData, and provides a uniform way to expose full-featured business objects on top of a highly scalable and high-availability Web server. Currently, Service Layer supports OData version3 and a few selected OData client libraries, for example, WCF for .Net developers; data.js for JavaScript developers.

2 Getting Started

2.1 System Requirements

SAP Business One Service Layer runs on SUSE Linux Enterprise Release 11 SP2, 64-bit edition. It is an application server built on the Apache HTTP Web server. The required database backend is SAP HANA Platform Edition 1.0 SPS 07 Rev74.

SAP Business One Service Layer can be deployed in one of two different modes:

- An integrated mode, installing in the same SAP HANA server so as to keep the system landscape as simple as possible
- A distributed mode, installing in separate machines to obtain more computing power for higher concurrent throughput

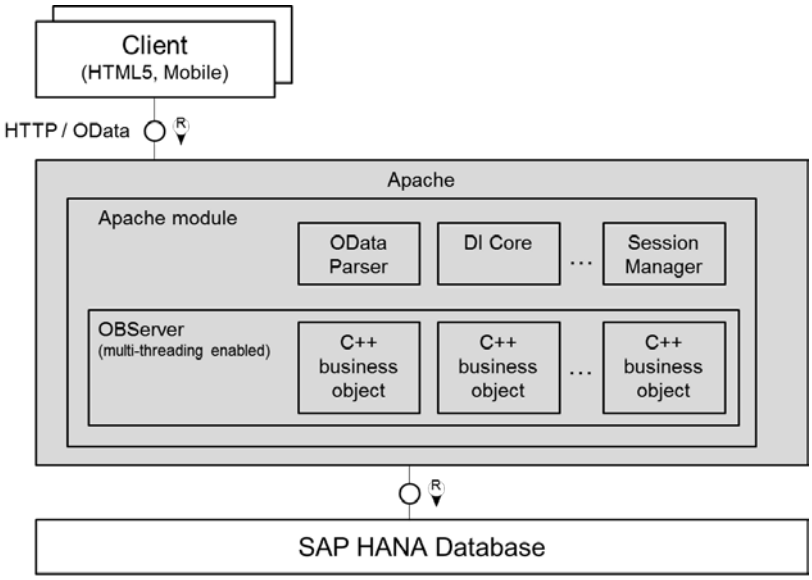
For hardware requirements, such as memory capacity or number of CPU cores, refer to SAP HANA hardware specifications.

2.2 Architecture Overview

SAP Business One Service Layer has a 3-tier architecture: the clients communicate with the Web server using HTTP/OData, and the Web server relies on the database for data persistence.

Within the Web server, several key components are involved in handling incoming OData-based HTTP requests:

- The OData Parser looks at the requested URL and HTTP methods (`GET/POST/PATCH/DELETE`), translates them into the business objects to be operated on, and calls each object's respective method for create/read/update/delete (CRUD) operations. In reverse, the OData Parser also receives the returned data from business objects, translates them into HTTP return code and JSON data representatives, and responds to the original client.
- The DI Core is the interface for accessing SAP Business One objects and services, the same one that is used by SAP Business One DI API. As a result, Service Layer API and DI API have identical definitions for objects and object properties, smoothing the learning curve for developers who have already acquired DI API development experience.
- The session manager implements session stickiness, working with the Service Layer load balancer, so that requests from the same client will be handled by the same Service Layer node.
- OBServer is the body of business logic dealing with the actual work, for example, tax calculation, posting, and so on. Service Layer is the first application that enables OBServer's multi-threading feature to support high performance and scalability.

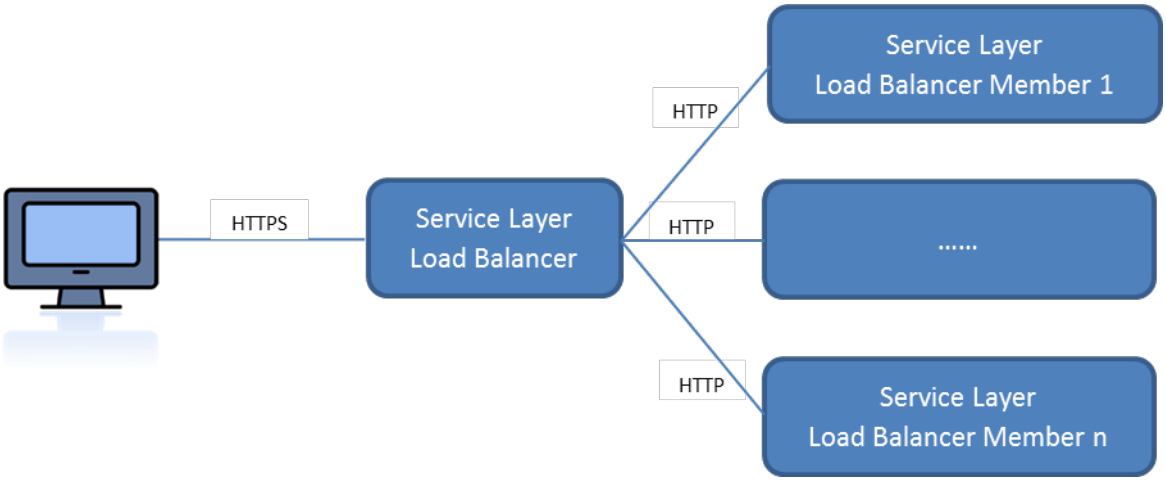


In order to achieve even higher availability and scalability, we recommend deploying multiple Service Layer instances with a load balancer in the front. The benefits include the following:

- Client requests can be dispatched to different Service Layer instances and executed in parallel.
- If Service Layer is installed in a distributed mode, and there is a hardware failure in one host machine, Service Layer is smart enough to re-dispatch client requests to another live instance without asking users to log on again.

2.3 Installing SAP Business One Service Layer

SAP Business One Service Layer is available as a server component of SAP Business One, version for SAP HANA. You must install Service Layer on the SAP HANA server machine (Linux). The landscape of Service Layer is illustrated below:



Recommendation

As the communication between the load balancer and the load balancer members is transmitted via HTTP instead of HTTPS, you should configure the firewall on each load balancer member machine in such a way that only visits from the load balancer are allowed to the load balancer members.

You may set up Service Layer in one of the following ways:

- [Recommended] The load balancer and load balancer members are all installed on different physical machines.
- The load balancer and one or more load balancer members are installed on the same machine while the other load balancer members are installed on different machines.
- The load balancer and all load balancer members are installed on the same machine.

Remote installation of Service Layer is **not** supported. For example, if you intend to install the load balancer on server A and two load balancer members on servers B and C, you must run the server components setup wizard on each server separately.

The installation order of the load balancer and load balancer members does not affect the functioning of Service Layer. However, we recommend that you create load balancer members first because you must specify the information of load balancer members when installing the load balancer. Below is a simplified procedural description for installing the Service Layer (load balancer and load balancer members).

Prerequisites

When copying installation files to each server, ensure the following points are met:

- The following files are available:
  - RPM packages:
    - `B1ServerToolsCommon`
    - `B1ServerToolsJava64`
    - `B1ServerToolsSupport`
    - `B1ServerToolsTomcat`
    - `B1ServiceLayerApacheWebServer`
    - `B1ServiceLayerComponent`
  - `install.bin`
- The original folder structure is kept. For example, the RPM packages must all reside in an `RPM` folder, separate from the binary file `install.bin`.

Procedure

1. Log on to the Linux server as a root user.
2. In a command line terminal, navigate to the directory …`/Packages.Linux/ServerComponents` where the `install.bin` script is located.
3. Start the installer from the command line by entering the following command:

   ```
   ./install.bin
   ```

   The installation process begins.

> **Note**
>
> If you receive the error message “Permission denied”, you must set execution permission on the installer script to make it executable. To do so, run the following command:
>
> ```
> chmod +x install.bin
> ```

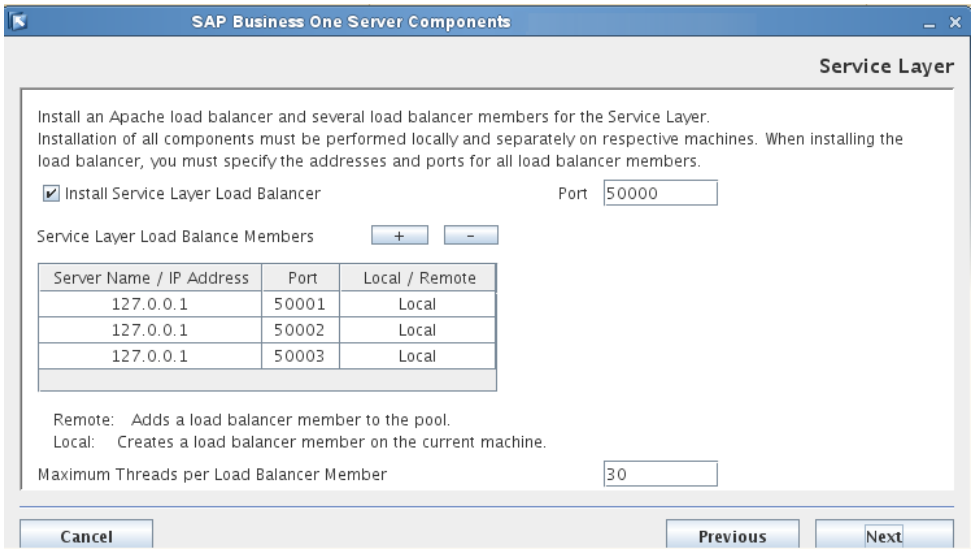
4. In the welcome window of the setup wizard, choose the *Next* button.
5. In the *Specify Installation Folder* window, specify a folder in which you want to install Service Layer and choose the *Next* button.
6. In the *Select Features* window, select *Service Layer*.
7. In the *Specify Security Certificate* window, specify a security certificate and choose the *Next* button. You can also choose to use a self-signed certificate.
8. In the *Database Server Specification* window, specify the information of your SAP HANA database server.
9. In the *Service Layer* window, specify the following information for Service Layer and then choose the *Next* button:
   - *Install Service Layer Load Balancer*: Select the checkbox to install the load balancer.

     When installing the load balancer, you need to specify the following information:
     - Port for the load balancer
     - Server name or IP address of all load balancer members, as well as their ports.
   - *Port*: Specify a port for the load balancer. Note that if the load balancer and load balancer members are installed on the same machine, each must use a different port.
   - *Service Layer Load Balancer Members*: Specify the server address and port number for each load balancer member.

     If you have selected the *Install Service Layer Load Balancer* checkbox, you can specify load balancer members either on local (current) or remote (different) machines. If on the local machine, the installer creates a local load balancer member; if on a remote machine, the load balancer member is added to the pool (cluster) of load balancer members, but you need to install the specific load balancer member on its own server.

     If you have not selected the checkbox, you cannot edit the server address, which is automatically set to `127.0.0.1` (`localhost`). All specified load balancer members are created.

     > **Note**
     >
     > IPv6 addresses are not allowed.
   - *Maximum Threads per Load Balancer Member*: Define the maximum number of threads to be run for each load balancer member.



10. In the *Review Settings* window, review your settings and then choose the *Install* button to start the installation.
11. In the *Setup Progress* window, when the progress bar displays 100%, choose the *Next* button to finish the installation.
12. In the *Setup Process Completed* window, review the installation results and then choose the *Finish* button to exit the wizard.

Results

After installing Service Layer, you can check the status of each balancer member in the balancer manager. To do so, in a Web browser, navigate to **`https://<Balancer Server Address>:<Port>/balancer-manager`**.

To start working with Service Layer, ensure that the system database **SBOCOMMON** and your company database are installed or upgraded to the same version. In addition, the SAP HANA database user must have proper privileges on these databases.

You can access the API help document for Service Layer in a Web browser: URL: **`https://<Load Balancer Server>:<Load Balancer Port>/b1s/docs/v1/index.html`**. Note that only the following Web browsers are supported:

- Microsoft Internet Explorer 7 and higher
- Google Chrome
- Mozilla Firefox
- Apple Safari

For more information, see the *SAP Business One Administrator's Guide, version for SAP HANA* at http://service.sap.com/smb/sbocustomer/documentation.

3 Consuming SAP Business One Service Layer

This section explains how to consume SAP Business One Service Layer and provides examples. For a full list of exposed entities and actions, refer to metadata returned by your service or the API reference of SAP Business One Service Layer.

Before interacting with Service Layer, refer to the following table for the key elements and terms:

Key Elements and Terms	Description/Activity	URL/Sample Code

Service Root URL	Identifies the root of Service Layer API. Both HTTP and HTTPS connections are supported.	`http://<server>:<port>/b1s/<version>`
		Example: `http://localhost:50000/b1s/v1`
		`https://<server>:<port>/b1s/<version>`
		Example: `https://localhost:8443/b1s/v1`

Resource Path	Identifies the resource to be interacted with. It can be a collection of entities or a single entity.	`http://<server>:<port>/b1s/<version>/<resource_path>`
		Example:
		`http://localhost:50000/b1s/v1/Items`

Query Options	Specifies multiple query options and operation parameters.	`http://<server>:<port>/b1s/<version>/<resource_path>?<query_options>`
		Example:
		`http://localhost:50000/b1s/v1/Items?$top=2&$orderby=itemcode`

HTTP Verb	Indicates the action to be taken against the resource, in accordance with the RESTful architectural principles.	In the following example, the 2 requests are equivalent: • `POST https://localhost/b1s/v1/Login` • `POST /Login`
JSON Resource Representation	Represents and interacts with structured content, embedded in Service Layer requests and responses.	`{"key1": "value1", "arr1": [100, 200], "key2": "value2"}`

Recommendation

To test Service Layer without developing a program, you can install the "POSTMAN" browser extension in Google Chrome, or install equivalent add-ons on other browsers.

3.1 Login and Logout

Before you perform any operation in Service Layer, you first need to log into Service Layer.

Send this HTTP request for login:

```
POST http://localhost/b1s/v1/Login
{"CompanyDB": "US506", "UserName": "manager", "Password": "1234"}
```

If the login is successful, you get the following response:

```
HTTP/1.1 200 OK
Set-Cookie:B1SESSION=002e31e8-df89-11e3-8000-047d7ba5aff2

{"SessionId": "002e31e8-df89-11e3-8000-047d7ba5aff2"}
```

Send this HTTP request for logout:

```
POST https://localhost/b1s/v1/Logout
Cookie: B1SESSION=002e31e8-df89-11e3-8000-047d7ba5aff2
```

> **Note**
>
> Cookie is set in the HTTP header. The cookie item `B1SESSION` can be used by all requests. If you write a client application, do not forget to add the cookie item in the HTTP header, as in the above example of Logout. Otherwise, you may receive the "Invalid session" error:
>
> ```
> HTTP/1.1 401 Unauthorized
>
> {
>    "error" : {
>       "code" : -1001,
>       "message" : {
>          "lang" : "en-us",
>          "value" : "Invalid session."
>       }
>    }
> }
> ```

3.2 Metadata Document

Metadata describes the capability of the service. It mainly defines types, entities (for example, SAP Business One objects) and actions (for example, SAP Business One services).

Send the following HTTP request to retrieve metadata:

```
GET /$metadata
```

Using SAP Business One business partners and sales orders as examples, you can see the following sections in the metadata:

```
<!-- section 1.1 -->
<EnumType Name="BoCardTypes">
  <Member Name="cCustomer" Value="C"/>
  <Member Name="cSupplier" Value="S"/>
  <Member Name="cLid" Value="L"/>
</EnumType>

<!-- section 1.2 -->
<EntityType Name="BusinessPartner">
  <Key>
    <PropertyRef Name="CardCode"/>
  </Key>
  <Property Name="CardCode" Nullable="false" Type="Edm.String"/>
  <Property Name="CardName" Type="Edm.String"/>
  <Property Name="CardType" Type="SAPB1.BoCardTypes"/>
  ...
</EntityType>

<!-- section 1.3 -->
<ComplexType Name="DocumentParams">
  <Property Name="DocEntry" Nullable="false" Type="Edm.Int32"/>
</ComplexType>

<!-- section 1.4 -->
<EntityType Name="Document">
  <Key>
    <PropertyRef Name="DocEntry"/>
  </Key>
  <Property Name="DocEntry" Nullable="false" Type="Edm.Int32"/>
  <Property Name="DocNum" Type="Edm.Int32"/>
  <Property Name="DocType" Type="SAPB1.BoDocumentTypes"/>
  ...
  <Property Name="DocumentLines" Type="Collection(SAPB1.DocumentLine)"/>
```

```
  ...
</EntityType>

<!-- section 1.5 -->
<ComplexType Name="DocumentLine">
  <Property Name="LineNum" Nullable="false" Type="Edm.Int32"/>
  <Property Name="ItemCode" Type="Edm.String"/>
  <Property Name="ItemDescription" Type="Edm.String"/>
  <Property Name="Quantity" Type="Edm.Double"/>
  ...
</ComplexType>

<!-- section 2.1 -->
<Action IsBindable="true" Name="Close">
  <Parameter Name="Document" Type="SAPB1.Document"/>
</Action>

<!-- section 2.2 -->
<Action Name="OrdersService.Close">
  <Parameter Name="DocumentParams" Type="SAPB1.DocumentParams"/>
</Action>

<!-- section 3 -->
<EntityContainer Name="ServiceLayer">
  <EntitySet EntityType="SAPB1.BusinessPartner" Name="BusinessPartners"/>
  <EntitySet EntityType="SAPB1.Document" Name="Orders"/>
  ...
</EntityContainer>
```

The above metadata sections indicate how the entities and actions are exposed:

- In Section 3, you can see that entities *BusinessPartners* and *Orders* are exposed. You can perform standard create/read/update/delete (CRUD) operations on them.
- In Section 2.1, you can see that a bindable action named *Close* is defined and can be bound to type *SAPB1.Document*. As orders are of this entity type, therefore, orders has a `Close` action (`POST /Orders(id)/Close`).
- In Section 2.2, you can see a global action named *OrdersService.Close* is defined. (You can use `POST /OrdersService.Close`).

3.3 Service Document

The service document is a list of exposed entities. Use the root service URL to retrieve the service document. Send the HTTP request:

```
GET /
```

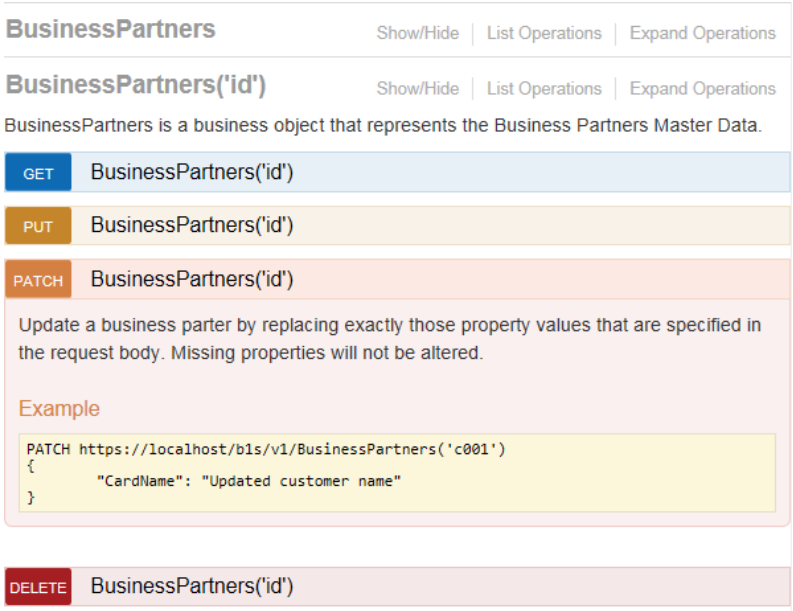
The response is:

```
HTTP/1.1 200 OK

{
    "value": [
        {
            "name": "ChartOfAccounts",
            "kind": "EntitySet",
            "url": "ChartOfAccounts"
        },
        {
            "name": "SalesStages",
            "kind": "EntitySet",
            "url": "SalesStages"
        },
        ...
       ]
}
```

3.4 Create/Read/Update/Delete (CRUD) Operations

OData protocol defines a standard way to create/read/update/delete (CRUD) an entity. The CRUD operations are all similar. You can refer to the API reference document for details (see the screenshot below).



3.4.1 Creating Entities

Use the HTTP verb **POST** and the content of an entity to create the entity.

The response on success is also the content of the entity.

Example

How to create a customer (business partner) named "c1"

Send this HTTP request:

```
POST /BusinessPartners
{
    "CardCode": "c1",
    "CardName": "customer c1",
    "CardType": "cCustomer"
}
```

All valid fields are defined in its type - *SAPB1.BusinessPartner* in metadata section 1.2.

Note that *CardType* is of type `Enumeration` (*BoCardTypes*, defined in metadata section 1.1). Both the enumeration name and value are accepted by Service Layer. So these two statements are equivalent:

```
{"CardType": "cCustomer",}
{"CardType": "C",}
```

On success, the server returns HTTP code `201` (Created) and the content of the entity is as follows:

```
HTTP/1.1 201 Created

{
    "CardCode": "c1",
    "CardName": "customer c1",
    "CardType": "cCustomer",
    "GroupCode": 100,
    ...
}
```

On error, the server returns HTTP code 4XX (for example, 400) and the error message as content is as follows (suppose customer "c1" exists):

```
HTTP/1.1 400 Bad Request

{
    "error": {
        "code": -10,
        "message": {
            "lang": "en-us",
            "value": "1320000140 - Business partner code 'c1' already assigned
to a business partner; enter a unique business partner code"
        }
    }
}
```

Example

How to create a sales order with two document lines

The `POST` content - entity `Orders` - is of type `Document` and defined in metadata section 1.4. `DocumentLines`, known as the sub-object of sales order, is a collection of the complex type `DocumentLine`, which is defined in metadata section 1.5. In JSON format, it is an array in square brackets [].

Send this HTTP request:

```
POST /Orders

{
    "CardCode": "c1",
    "DocDate": "2014-04-01",
    "DocDueDate": "2014-04-01",
    "DocumentLines": [
        {
            "ItemCode": "i1",
```

```
            "UnitPrice": 100,
            "Quantity": 10,
            "TaxCode": "T1",
        },
        {
            "ItemCode": "i2",
            "UnitPrice": 120,
            "Quantity": 8,
            "TaxCode": "T1",
        },
    ]
}
```

On success, the server returns `201` (Created) and the content of the entity is as follows:

```
HTTP/1.1 201 Created

{
    "DocEntry": 22,
    "DocNum": 11,
    "DocType": "dDocument_Items",
    ...
    "DocumentLines": [
        {
            "LineNum": 0,
            "ItemCode": "i1",
            ...
        },
        {
            "LineNum": 1,
            "ItemCode": "i2",
            ...
        }
    ],
    ...
}
```

3.4.2 Reading Entities

Use the HTTP verb `GET` and the key fields to read the entity.

Example

How to get the customer "c1" in the previous example

As defined in metadata section 1.2, *CardCode* is the key property (type is string). To retrieve the customer "c1", send the HTTP request:

```
GET /BusinessPartners('c1')
```

or

```
GET /BusinessPartners(CardCode='c1')
```

The service returns HTTP code `200` that indicates success with the content of the object in JSON format:

```
HTTP/1.1 200 OK

{
    "CardCode": "c1",
    "CardName": "customer c1",
    "CardType": "cCustomer",
    "GroupCode": 100,
    ...
}
```

Example

How to get the sales order in the previous example

As defined in metadata section 1.4, *DocEntry* is the key property (type is Int32). To retrieve the sales order, send the HTTP request:

```
GET /Orders(22)
```

or

```
GET Orders(DocEntry=22)
```

Note

Single quotes are required for string values such as 'c1', and no single quotes around integer values such as 22.

If the entity key contains multiple properties, send the HTTP request:

```
GET /SalesTaxAuthorities(Code='AK',Type=-3)
```

3.4.3 Updating Entities

Use the HTTP verb `PATCH` or `PUT` to update the entity. Generally, `PATCH` is recommended.

The difference between `PATCH` and `PUT` is that `PATCH` ignores (keeps the value) those properties that are not given in the request, while `PUT` sets them to the default value or to null.

Example

How to update the name of the customer "c1"

Send the HTTP request:

```
PATCH /BusinessPartners('c1')

{
    "CardName": "Updated customer name"
}
```

On success, HTTP code `204` is returned without content.

```
HTTP/1.1 204 No Content
```

Note

Read only properties (for example, *CardCode*) cannot be updated. They are ignored silently if assigned in the request.

3.4.4 Deleting Entities

Use the HTTP verb `DELETE` and the key fields to delete the entity.

Example

How to delete the customer "c1"

Send the HTTP request:

```
DELETE /BusinessPartners('c1')
```

On success, HTTP code `204` is returned without content.

```
HTTP/1.1 204 No Content
```

Note

You cannot delete the sales order in SAP Business One. If you try to delete the sales order No.22:

```
DELETE /Orders(22)
```

An error is reported to deny the operation:

```
HTTP/1.1 400 Bad Request
{
    "error": {
        "code": -5006,
        "message": {
            "lang": "en-us",
            "value": "The requested action is not supported for this object."
        }
    }
}
```

3.5 Actions

Besides the basic entity CRUD operations, Service Layer provides you with two kinds of actions:

- Bound action (bound to entity for operations other than CRUD)
- Global action (mainly used to expose SAP Business One services)

The request and response for each action are described in the metadata. For example, the login function that was introduced above is a global action. You can find its definition in metadata.

You can use the HTTP verb `POST` for OData actions.

Example

How to use the bound action

In the metadata section 2.1, you can see a bindable action named "Close" with the first parameter bound to the `Document` type:

```
<!-- section 2.1 -->
<Action IsBindable="true" Name="Close">
  <Parameter Name="Document" Type="SAPB1.Document"/>
</Action>
```

As orders are of type `Document`, that means orders have a "Close" action. You can send the following HTTP request to close the document No. 22:

```
POST /Orders(22)/Close
```

Example

How to use the global action

In SAP Business One DIAPI, you can use the *SAPbobsCOM.Activity* object to operate the activities in SAP Business One. However, in Service Layer, you cannot find the *Activity* entity. Then how to use it?

By searching in metadata, you can find the action definitions, as follows:

```
<Action Name="ActivitiesService.GetActivity">
  <Parameter Name="ActivityParams" Type="SAPB1.ActivityParams"/>
  <ReturnType Type="SAPB1.Activity"/>
</Action>

<Action Name="ActivitiesService.AddActivity">
  <Parameter Name="Activity" Type="SAPB1.Activity"/>
  <ReturnType Type="SAPB1.ActivityParams"/>
</Action>
```

It shows that you can use the *ActivitiesService* to get and add activity objects. The related types are also defined in metadata, as follows:

```
<ComplexType Name="ActivityParams">
  <Property Name="ActivityCode" Nullable="false" Type="Edm.Int32"/>
</ComplexType>
```

```
<ComplexType Name="Activity">
  <Property Name="ActivityCode" Nullable="false" Type="Edm.Int32"/>
  <Property Name="CardCode" Type="Edm.String"/>
  <Property Name="Notes" Type="Edm.String"/>
  ...
</ComplexType>
```

To add an activity, send the HTTP request:

```
POST /ActivitiesService.AddActivity
{
    "Activity":{
        "ActivityCode": 1,
        "CardCode": "c1"
    }
}
```

On success, it returns the content of type `SAPB1.ActivityParams` as defined.

To get an activity, send the HTTP request:

```
POST /ActivitiesService.GetActivity
{
    "ActivityParams": {
        "ActivityCode": 1
    }
}
```

On success, it returns the content of type `SAPB1.Activity` as defined.

Example

Closing an order - non-bound version

There is a hidden action named `OrdersService.Close` that you can also use to close an order. To view the action, set `DebugLevel=2` in `b1s.conf`.

In metadata, you can see the definition in section 2.2:

```
<!-- section 2.2 -->
<Action Name="OrdersService.Close">
  <Parameter Name="DocumentParams" Type="SAPB1.DocumentParams"/>
</Action>
```

And the first parameter `DocumentParams` is defined in section 1.3.

```
<!-- section 1.3 -->
<ComplexType Name="DocumentParams">
  <Property Name="DocEntry" Nullable="false" Type="Edm.Int32"/>
</ComplexType>
```

To close the order No.22, you can send the HTTP request:

```
POST /OrdersService.Close
{
    "DocumentParams": {"DocEntry": 22}
}
```

3.6 Query Options

Query options within the request URL can control how a particular request is processed by Service Layer. The following table shows the query options supported by Service Layer.

Option	Description	Example

`$filter`	Queries collections of entities.
	The currently supported functions for `$filter` are:
	• `startswith`
	• `endswith`
	• `contains`
	• `substringof`
	Currently supported logical and relational operators include: `and`, `or`, `le` (less than or equal to), `lt` (less than), `ge` (greater than or equal to), `gt` (greater than), `eq` (equal to), and `ne` (not equal to). Parenthesis is also supported.
		`/Orders?$filter=DocTotal gt 3000`
		
		`/Orders?$filter=DocEntry lt 8 and (DocEntry lt 8 or DocEntry gt 116) and CardCode eq 'c1'`
		
		`/Orders?$filter=DocEntry lt 8 and ((DocEntry lt 8 or DocEntry gt 116) and startswith(CardCode,'c1'))`

`$select`	Returns the properties that are explicitly requested.	`/Orders?$select=DocEntry, DocTotal`
`$orderby`	Specifies the order in which entities are returned.	`/Orders?$orderby=DocTotal asc, DocEntry desc`
`$top`	Returns the first n (non-negative integer) records.	`/Orders?$top=3`
`$skip`	Specifies the result excluding the first n entities.	`/Orders?$top=3&$skip=2` Where `$top` and `$skip` are used together, the `$skip` is applied before the `$top`, regardless of the order of appearance in the request.
`$count`	Returns the count of an entity collection.	`/Orders/$count`

The combination of query options enables Service Layer to support any complex query scenarios, while keeping the API interface as simple as possible.

Example

How to get all entities

You can use the following ways to get all entity records:

```
GET /Items
```

or

```
GET /Items?$select=*
```

Example

How to get fields of an entity

You can use the following ways to get item fields:

```
GET /Items('i1')?$select=ItemCode,ItemName,ItemPrices
```

or

```
GET /Items(ItemCode='i1')?$select=ItemCode,ItemName,ItemPrices
```

Example

How to query properties of the enumeration type

Enumeration value and enumeration name are both supported in a query option. You can use the following ways to get all customers:

```
GET /BusinessPartners?$filter=CardType eq 'C'
```

or

```
GET /BusinessPartners?$filter=CardType eq 'cCustomer'
```

Note that '`C`' is an enumeration value while '`cCustomer`' is an enumeration name.

Example

How to query properties of the datetime type

Multiple date formats are supported. For example:

```
GET /Orders?$filter=DocDate eq '2014-04-23'
GET /Orders?$filter=DocDate eq '20140423'
GET /Orders?$filter=DocDate eq datetime'2014-04-23'
GET /Orders?$filter=DocDate eq datetime'20140423'
GET /Orders?$filter=DocDate eq '2014-04-23T12:21:21'
GET /Orders?$filter=DocDate eq '20140423000000'
```

Note that SAP Business One ignores the HOUR/MINUTE/SECOND parts. The `datetime` keyword prefix can also be added before the datetime value.

Example

How to query properties of the time type

Multiple time formats are supported. For example:

```
GET /Orders?$filter=DocTime eq '18:38:00'
GET /Orders?$filter=DocTime eq '18:38'
GET /Orders?$filter=DocTime eq '183800'
GET /Orders?$filter=DocTime eq '1838'
GET /Orders?$filter=DocTime eq '2014-06-18T18:38:00Z'
GET /Orders?$filter=DocTime eq '2014-06-18T18:38'
```

Note that SAP Business One ignores the YEAR/MONTH/DAY parts; only the HOUR/MINUTE parts are effective.

Example

How to paginate the selected orders

The pagination mechanism is implemented through top and skip. It allows the data to be fetched chunk by chunk. For example, after you send the HTTP request:

```
GET /Orders
```

The service returns:

```
HTTP/1.1 200 OK

{
"value": [
       {"DocEntry": 7,"DocNum": 2,...},
       {"DocEntry": 8,"DocNum": 3,...},
       ...
       {"DocEntry": 26,"DocNum": 21,...}
],
"odata.nextLink": "/b1s/v1/Orders?$skip=20"
}
```

Annotation `odata.nextLink` is contained in the body for the link of the next chunk.

Note

For OData V3, the next link annotation is `odata.nextLink`; For OData V4, the next link annotation is `@odata.nextLink`.

The default page size is 20. You can customize the page size by changing the following options:

- Set the configuration option `PageSize` in `conf/b1s.conf`.
- Use the OData recommended annotation `odata.maxpagesize` in the `Prefer` header of the request:

  ```
  GET /Orders
  Prefer:odata.maxpagesize=50
  ... (other headers)
  ```

  The response contains HTTP header `Preference-Applied` to indicate whether and how the request is accepted:

  ```
  HTTP/1.1 200 OK
  Preference-Applied: odata.maxpagesize=50
  ...
  ```

If `PageSize` or `odata.maxpagesize` is set to 0, the pagination mechanism is turned off.

The by-request option `odata.maxpagesize` is prior to the configuration option `PageSize`.

3.7 Batch Operations

Service Layer supports executing multiple operations sent in a single HTTP request through the use of batching. A batch request must be represented as a Multipart MIME (Multipurpose Internet Mail Extensions) v1.0 message.

3.7.1 Batch Request Method and URI

Always use HTTP `POST` method to send a batch request. A batch request is submitted as a single HTTP `POST` request to the batch endpoint of a service, located at the URI `$batch` relative to the service root.

```
POST https://localhost:8443/b1s/v1/$batch
```

3.7.2 Batch Request Headers

The batch request must contain a `Content-Type` header that specifies a content type of multipart/mixed and a boundary specification as:

```
Content-Type: multipart/mixed;boundary=<Batch Boundary>
```

The boundary specification is used in the *Batch Request Body* section.

3.7.3 Batch Request Body

The body of a batch request is composed of a series of individual requests and change sets, each represented as a distinct MIME part, and separated by the boundary defined in the `Content-Type` header.

```
--<Batch Boundary>
<subrequest-1>
--<Batch Boundary>
<subrequest-2>
--<Batch Boundary>
Content-Type: multipart/mixed;boundary=<Changeset Boundary>

--<Changeset boundary>
<subchangeset-request-1>
--<Changeset boundary>
<subchangeset-request-2>
--<Changeset boundary>--
--<Batch Boundary>--
```

The service processes the requests within a batch request sequentially.

Each sub request must include a `Content-Type` header with value `application/http` and a `Content-Transfer-Encoding` header with value `binary`.

```
Content-Type:application/http
Content-Transfer-Encoding:binary

<sub request body>
```

The sub request body includes the real request content.

Example

```
POST /b1s/v1/Items

<Json format Items Content>
```

or

```
GET /b1s/v1/Item('i001')
```

Note that two empty lines are necessary after the `GET` request line. The first empty line is part of the `GET` request header, and the second one is the empty body of the `GET` request, followed by a `CRLF`.

3.7.4 Change Sets

A change set is an atomic unit of works. It means that any failed sub request in a change set will cause the whole change set to be rolled back. Change sets must not contain any `GET` requests or other change sets.

Sub change set requests basically have the same format as sub requests outside change sets, except for one additional feature: `Referencing Content ID`.

Referencing Content ID: New entities created by a `POST` request within a change set can be referenced by subsequent requests within the same change set by referring to the value of the `Content-ID` header prefixed with a `$` character. When used in this way, `$<Content-ID>` acts as an alias for the URI that identifies the new entity.

Example

How to use change set with Content-ID

1. Create an order.
2. Use `$<Content-ID>` to modify the order you just created.

```
--<Batch Boundary>
Content-Type: multipart/mixed;boundary=<Changeset Boundary>

--<Changeset boundary>
Content-Type:application/http
Content-Transfer-Encoding:binary
Content-ID:1
```

```
POST /b1s/v1/Items

<Json format Items Content>
--<Changeset boundary>
Content-Type:application/http
Content-Transfer-Encoding:binary
Content-ID:2

PATCH /b1s/v1/$1

<Json format Item update content>
--<Changeset boundary>--
--<Batch Boundary>--
```

Note that `Content-ID` only exists in change set sub requests:

- For OData Version 3, it is not necessary to use `Content-ID` unless you need to use it for reference.
- For OData Version 4, this header is a mandatory field, whether you use it or not.

3.7.5 Batch Request Sample Codes

The sample codes in this section show a complete batch request that contains the following operations:

- A query request
- A change set that contains the following requests:
  - Insert entity (with `Content-ID` = 1)
  - Update request (with `Content-ID` = 2)

Sample Codes

```
POST https://localhost:443/b1s/v1/$batch
OData-Version: 4.0
Content-Type: multipart/mixed;boundary=batch_36522ad7-fc75-4b56-8c71-56071383e77b

--batch_36522ad7-fc75-4b56-8c71-56071383e77b
Content-Type: application/http
Content-Transfer-Encoding:binary

GET /b1s/v1/Items('i001')

--batch_36522ad7-fc75-4b56-8c71-56071383e77b
```

```
Content-Type: multipart/mixed;boundary=changeset_77162fcd-b8da-41ac-a9f8-9357efbbd

--changeset_77162fcd-b8da-41ac-a9f8-9357efbbd
Content-Type: application/http
Content-Transfer-Encoding: binary
Content-ID: 1

POST /b1s/v1/Items('i002')
Content-Type: application/json

<Json format item(i002) body>
--changeset_77162fcd-b8da-41ac-a9f8-9357efbbd
Content-Type: application/http
Content-Transfer-Encoding: binary
Content-ID: 2

PATCH /b1s/v1/$1
Content-Type: application/json

<Json format item(i002) update body>
--changeset_77162fcd-b8da-41ac-a9f8-9357efbbd--
--batch_36522ad7-fc75-4b56-8c71-56071383e77b--
```

3.7.6 Batch Response

This section contains the batch responses after you execute the batch requests.

[Batch Request Format Invalid]

Service returns HTTP error code: `400 Bad Request` with error info in body if the request format is not valid.

Example

```
"error" : {
  "code" : -1000,
  "innererror" : {
        "context" : null,
        "trace" : null
  },
  "message" : {
        "lang" : "en-us",
        "value" : "Incomplete Batch Request Body!"
  }
```

```
}
```

[Batch Request Format Valid]

Service returns HTTP code: `202 Accept` (OData Version 3) or `200 OK` (OData Version 4) if the request format is valid, The response body that is returned to the client depends on the request execute result.

- If the batch request execution is successful, each sub request will have a corresponding sub response in the response body.

Example

```
--batchresponse_d878cedc-a0ad-4025-823e-5ee1aaffa288
Content-Type:application/http
Content-Transfer-Encoding:binary

HTTP/1.1 200 OK
Content-Type:application/json;odata=minimalmetadata;charset=utf-8
Content-Length:14729

<Json format Item(i001) Body>
--batchresponse_d878cedc-a0ad-4025-823e-5ee1aaffa288
Content-Type:multipart/mixed;boundary=changesetresponse_8bfb3c36-dbf7-46a0-bdfe-
670bbac86eb2

--changesetresponse_8bfb3c36-dbf7-46a0-bdfe-670bbac86eb2
Content-Type:application/http
Content-Transfer-Encoding:binary
Content-ID:1

HTTP/1.1 201 Created
Content-Type:application/json;odata=minimalmetadata;charset=utf-8
Content-Length:14641
Location:http://10.58.81.158:9090/b1s/v1/Items('i002')

<Json format Item(i002) Body>
--changesetresponse_8bfb3c36-dbf7-46a0-bdfe-670bbac86eb2
Content-Type:application/http
Content-Transfer-Encoding:binary
Content-ID:2

HTTP/1.1 204 No Content


--changesetresponse_8bfb3c36-dbf7-46a0-bdfe-670bbac86eb2--
```

```
--batchresponse_d878cedc-a0ad-4025-823e-5ee1aaffa288--
```

- If the batch request execution is not successful, the batch will stop executing once a sub request fails.

  Note that when there is a failure in the change set, only one response returns for this change set, no matter how many sub requests exist in this change set. For example, in the example below, the CREATE item operation fails because an item with the same item code already exists in the database.

  Example

```
--batchresponse_3aa0885d-245c-4164-b9a4-9c27f7a2c4d1
Content-Type:application/http
Content-Transfer-Encoding:binary

HTTP/1.1 200 OK
Content-Type:application/json;odata=minimalmetadata;charset=utf-8
Content-Length:14729

<Json format Item(i001) Body>
--batchresponse_3aa0885d-245c-4164-b9a4-9c27f7a2c4d1
Content-Type:application/http
Content-Transfer-Encoding:binary

HTTP/1.1 400 Bad Request
Content-Type:application/json;odata=minimalmetadata;charset=utf-8
Content-Length:233

{
   "error" : {
      "code" : -10,
      "innererror" : {
         "context" : null,
         "trace" : null
      },
      "message" : {
         "lang" : "en-us",
         "value" : "Item code 'i002' already exists"
      }
   }
}
--batchresponse_3aa0885d-245c-4164-b9a4-9c27f7a2c4d1--
```

3.8 User-Defined Fields (UDFs)

User-defined fields (UDFs) are treated as dynamic properties of an OData entity. An entity that has a dynamic property is of "open type", that is, in the `EntityType` XML node in metadata, it has the attribute `OpenType=true`.

Note

UDFs do not appear in the entity definition in metadata.

All UDFs in SAP Business One are prefixed with "U_".

Example

How to add Business Partners with a UDF "U_BPSpecRemarks"

Send the HTTP request:

```
POST /BusinessPartners
{
    "CardCode": "bpudf_004",
    ...
    "U_BPSpecRemarks": "First Business Partners with UDF remarks added by
Chrome."
}
```

The service returns:

```
HTTP/1.1 200 OK
{
    "CardCode": "bpudf_004",
    ...
    "U_BPSpecRemarks": "First Business Partners with UDF remarks added by
Chrome.",
    ...
}
```

Example

How to query entities using UDFs

Send the HTTP request:

```
GET /BusinessPartners?$filter=startswith(U_BPSpecRemarks, 'First')
```

The service returns:

```
HTTP/1.1 200 OK
{
    "value": [
        {
```

```
            "CardCode": "bpudf_001",
            ...
            "U_BPSpecRemarks": "First Business Partners with UDF remarks.",
        },
        {
            "CardCode": "bpudf_003",
            ...
            "U_BPSpecRemarks": "First Business Partners with UDF remarks added
by Chrome.",
        },
        ...
    ]
}
```

4 Configuring SAP Business One Service Layer

The installation wizard sets the common configuration options when you install the Service Layer load balancer or balancer members. The configuration options are in the configuration file `conf/b1s.conf`.

4.1 Configuration Options for Service Layer

You can specify the configuration options to control the behavior of the service in the file `conf/b1s.conf`.

Server Connection Options

Option	Type	Description and Default Values
Server	String	The SAP HANA instance. Default value is `127.0.0.1:30015`.
DbUserName *DbPassword*	String	The user name and password for the SAP HANA database instance. Default value is: `SYSTEM` and `manager` respectively. Generally *DbUserName* and *DbPassword* are encrypted in `b1s.conf` unless *DbUserName* is `SYSTEM` or `sa` - then Service Layer does not perform decryption on the two fields.
DbServerType	Integer	Default value is `9` (for SAP HANA database).
UseTrusted	Boolean	Default value is `False`.
LicenseServer	String	Default value is `localhost:40000`.

Other Options

Option	Type	Description and Default Values
ExperimentalMetadata	Boolean	Default value is `False`. By default, only a subset of entities and actions is exposed. If this option is `True`, the metadata returns the full set of entities and actions. However, we do not recommend that you set this option to `True`, for it has not been tested yet. We cannot guarantee that the feature currently works exactly as intended and will not be changed in the future.
WCFCompatible	Boolean	Default value is `False`. If the value is set to `True`, the Microsoft WCF component can consume Service Layer. The application works around some limitations of WCF and the application behavior is as follows:

Option	Type	Description and Default Values
		• `EnumType` is replaced by `Edm.String`, since `EnumType` is not supported by WCF in metadata. • The property name cannot be the same as the type name. For example, `BatchNumber.BatchNumber` is automatically renamed to `BatchNumber.BatchNumberProperty`. • Use type `Edm.DateTime` instead of `Edm.Time`, as Microsoft .net does not have a `Time` type and uses `TimeSpan` instead, which is not compatible with SAP Business One.
MetadataWithoutSession	Boolean	Default value is `False`. To get metadata, you must first log in and then request metadata in the same session, which means the HTTP request must contain the `B1SESSION` cookie item returned by the login request. If this option is set to `True`, you can get metadata anywhere after you log in to the service. Internally, Service Layer takes one of the existing sessions for the metadata request.
PageSize	Integer	Defines the page size when paging is applied for a query. Default value is `20`.

4.2 Configuration by Request

Service Layer supports the configuration options listed above to take effect immediately and limited to the request level. You can set the Service Layer-customized HTTP header to overwrite the settings in `b1s.conf` only for the current request.

To configure your settings for the current request, you should use the following format:

```
B1S-<configuration-item-name>: <value>
```

For example:

- `B1S-WCFCompatible: True`
- `B1S-PageSize: 100`

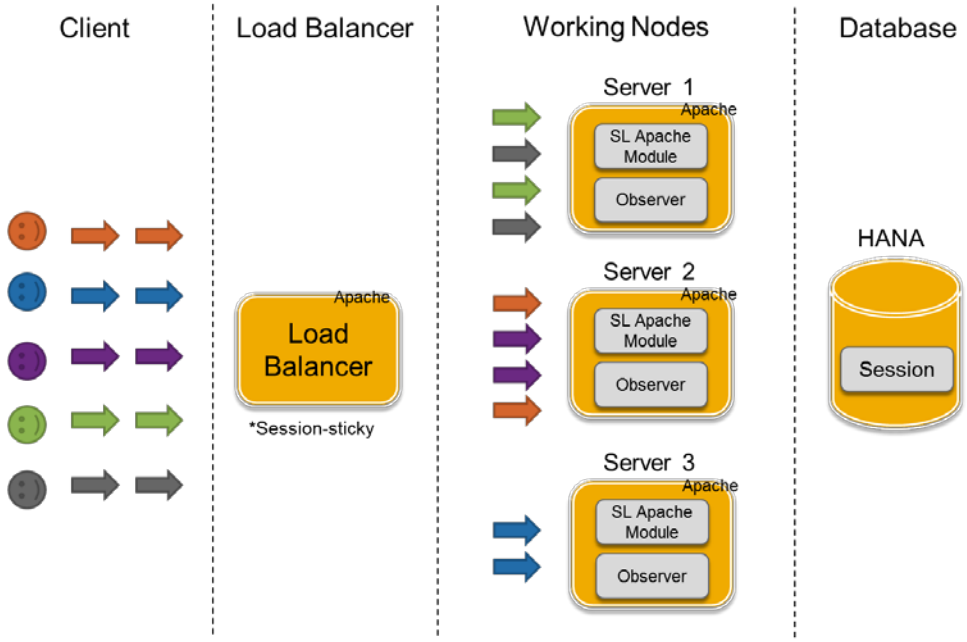
> **Note**
>
> This feature is available in SAP Business One 9.1 patch level 01 and later.

5 High Availability and Performance

5.1 High Availability and Load Balancing

In the context of Web-based mobile-accessible applications, providing highly available services becomes increasingly important. Service Layer is well-designed and thoroughly tested to ensure that it will be continuously operational for a significant length of time in a production system.

By default, Service Layer installs Apache Multi-Processing Modules (MPMs) and is configured as a load-balancing cluster. A central load balancer distributes HTTP loads amongst its nodes according to the number of requests. In addition, Service Layer implements sticky sessions to avoid an unnecessary login, which is considered a heavy job in SAP Business One, because same session requests will always be forwarded to the same working node. Those working nodes can be deployed in a clustered system, where hardware and software redundancy helps to scale performance and provide high availability.



In an exceptional case, if the load balancer detects that one of its nodes has failed, it forwards subsequent requests to another valid node. The receiving node validates the session through the shared session info stored in the database. If valid, the receiving node automatically logs the user in, without interrupting the user actions or asking for user credentials. End-users will not notice the internal node failure, other than in a slight delay of the system response.

5.2 Load Test Benchmarking

Performance testing was conducted for the purpose of determining concurrent transaction capacity with the condition of reasonable response time for 90% of all transactions. Testing was done on a standard installed environment with a set of configuration options, including CPU core numbers used by SAP HANA, CPU core numbers used by Service Layer, number of concurrent requests, maximum working thread number in each Service Layer instance, and so on. Concurrent testing began with user login and comprised creating an order with 20 lines, copying the order to a delivery, and finally copying the delivery to an invoice. One thing to note is that, although the three documents were posted separately, all were counted as one transaction when calculating the transactions per second (tps). The correctness of posted transactions was checked after the test in order not to affect the throughput test result. Moreover, the SAP HANA log was saved to a fast-IO disk, such as a solid state disk (SSD), to maximize the performance of SAP HANA.

No	SAP HANA CPU Cores	Service Layer CPU Cores	Number of Concurrent Requests	Concurrent Throughput (tps)	Average Response Time in Seconds (Order/Delivery/Invoice)
1	32	10	60	2.6	3.9/18.4/8.0
2	32	10	40	2.7	3.2/10.1/6.3
3	32	10	20	0.9	4.2/9.4/6.9
4	32	4	40	1.5	4.2/10.6/7.1
5	32	2	40	0.9	7.8/18.9/13.9
6	80	15	60	6.7	1.3/3.6/3.1

According to the above test data, the highest concurrent throughput reaches 6.7 with a good response time on each kind of document. The final test result also shows that concurrent throughput is mainly affected by the following factors:

- Increasing the number of concurrent requests will gradually increase tps, but will reach the limit at 40 concurrent requests, where the benefit begins to diminish and response time become worse.
- Increasing Service Layer CPU cores will improve tps, for example, if you increase CPU cores from 4 to 10, the tps roughly doubles.
- Adding more CPU cores to Service Layer will eventually overwhelm the SAP HANA server; therefore, a larger multi-processor SAP HANA server is recommended. For example, in the last row, after increasing SAP HANA CPU cores from 32 to 80, the concurrent throughput increases significantly from 2.7 to 6.7.

6 FAQ

1. What are the differences between Service Layer and other SAP Business One extension APIs, such as DI API and DI Server?

   Service Layer is built on the DI core technology, which is also the foundation of DI API and DI Server. Therefore, all three extension APIs share similar business object definitions. However, their differences are significant:

   - DI API derives from Microsoft COM technology and fits best in the Windows native environment;
   - DI Server targets SOAP-based data integration scenarios and prefers Web-services architecture;
   - Service Layer is an OData-compliant data service with a smoother learning curve, which enables easy Web Mashup, or effortless add-on development in various languages (Java, JavaScript, .NET) using 3rd-party libraries. For a list of all OData client libraries, refer to http://www.odata.org/libraries/. Service Layer is also a full-featured web application server with capabilities of high availability and scalable performance.

2. Service Layer is OData-compliant and RESTful, what are the other implications of moving to such Web-services architecture?

   Service Layer provides lightweight and faster results and simple transactions (for example, CRUD operations). Querying objects is just a matter of changing URI in a uniform fashion. Batch operation is to support advanced transaction scenarios where multiple requests need to be applied in an atomic way.

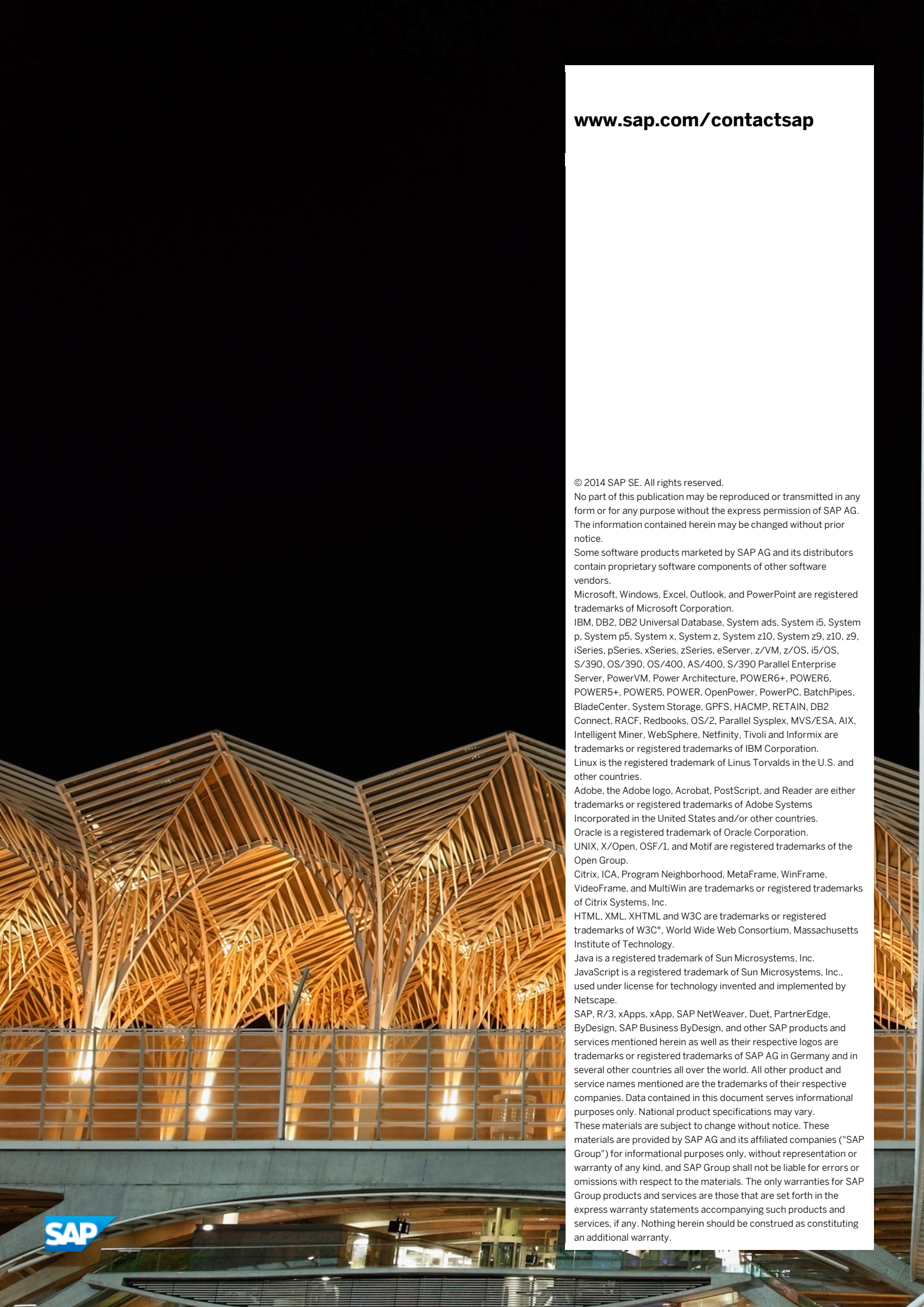
   Service Layer may not be a good choice to implement complex or distributed transactions where server-side state management is a must-have requirement.

3. Why does Service Layer support two types of update?

   The two types of update differ in the HTTP verb that is sent in the request:

   - A PUT request indicates a replacement update. All property values specified in the request body are replaced. Missing properties are set to their default values.
   - A PATCH request indicates a differential update. Only exactly those property values in the request body are replaced. Missing properties are not altered.

   In most cases, Patch request is the recommended approach to update object data.



www.sap.com/contactsap