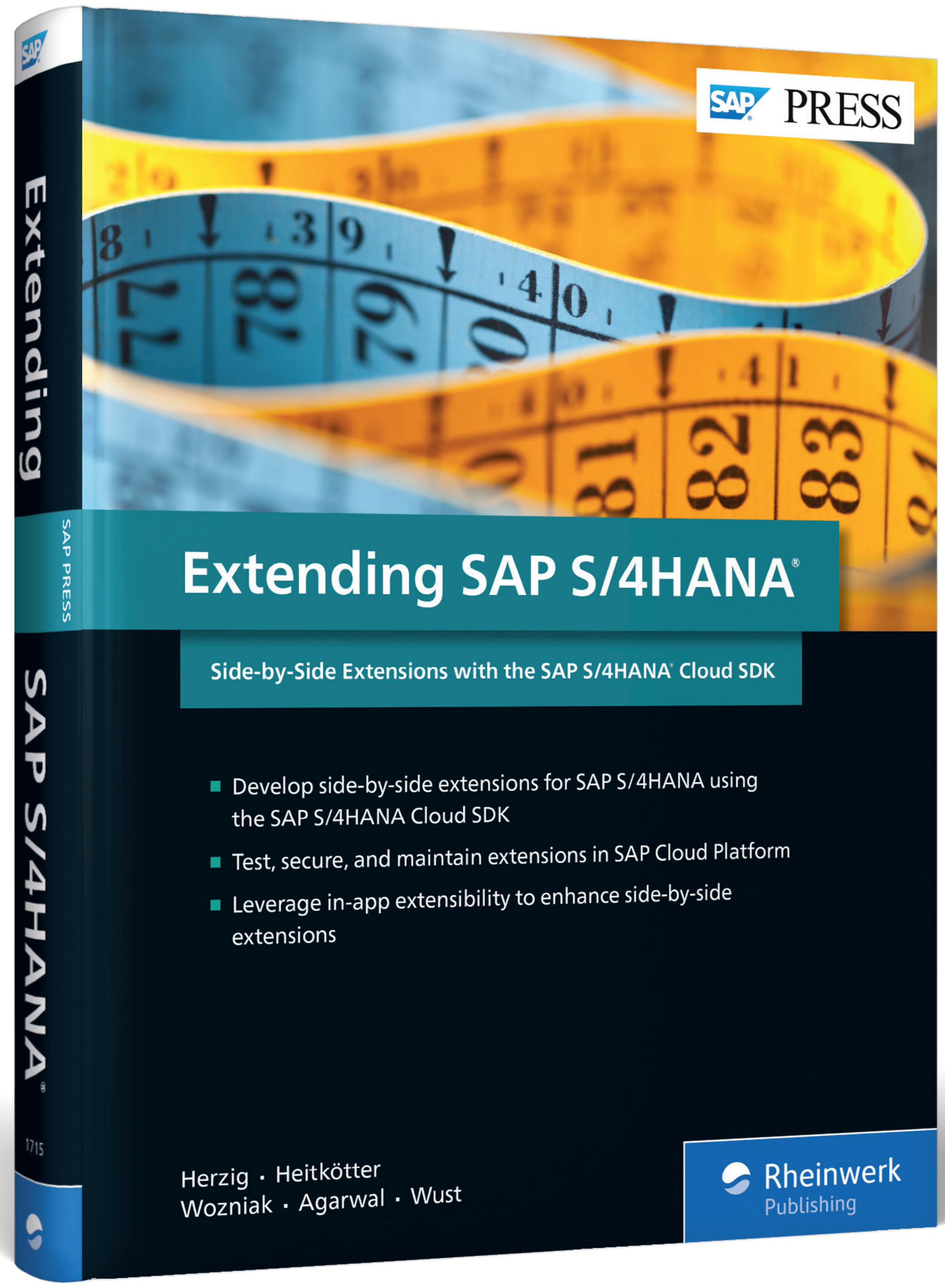
SAP PRESS First-hand knowledge.

Rheinwerk Publishing


SAP PRESS
Extending SAP S/4HANA
Side-by-Side Extensions with the SAP S/4HANA Cloud SDK
Develop side-by-side extensions for SAP S/4HANA using the SAP S/4HANA Cloud SDK
Test, secure, and maintain extensions in SAP Cloud Platform
Leverage in-app extensibility to enhance side-by-side extensions
Herzig · Heitkötter
Wozniak · Agarwal · Wust
Rheinwerk Publishing

Reading Sample

Explore two chapters from this book! In the first chapter you'll learn about SAP S/4HANA's new extensibility strategy and how it differs from traditional approaches. Then, read on to find out how to secure extension applications using microservices.

"SAP S/4HANA Extensibility"

"Application Security"

Contents

Index

The Authors

Herzig, Heitkötter, Wozniak, Agarwal, and Wust

Extending SAP S/4HANA: Side-by-Side Extensions with the SAP S/4HANA Cloud SDK

618 Pages, 2018, $79.95
ISBN 978-1-4932-1715-1

www.sap-press.com/4655

Chapter 2
SAP S/4HANA Extensibility

This chapter explains the various forms and modes through which SAP S/4HANA can be extended. This chapter, in particular, includes in-app extensibility (i.e., all extensions made with key user tools inside of the SAP S/4HANA core) as well as side-by-side extensibility (i.e., all extensions on SAP Cloud Platform). We'll provide an overview of extension capabilities and their corresponding integration points and outline patterns for extensibility and their characteristics. Although the book touches on in-app extensibility, our primary focus is on side-by-side extensibility.

After introducing the general design principles and rationales of the SAP S/4HANA architecture, this chapter explains SAP S/4HANA's extensibility options and their business implications in more detail.

The majority of customers using one of SAP's ERP solutions have significantly adjusted and extended the standard software provided by SAP to implement business processes that are specific to their organizations. Extensions range from simple reports or modifications of a few lines to SAP's ABAP source code to newly developed applications for an industry-specific adaptation of a standard process. Furthermore, many partners have provided significant industry-specific or line of business (LOB)-specific enhancements, usually in the form of large ABAP add-ons.

This chapter explains SAP S/4HANA's new extensibility strategy and how it differs from the traditional approaches. While the following chapters of this book dive into specific details of extensibility, the goal of this chapter is to provide an overview of extensibility options and capabilities.

The extensibility strategy is based on two pillars: in-app extensibility and side-by-side extensibility. As the name implies, in-app extensions are implemented using key user tools directly inside the SAP S/4HANA system, while side-by-side extensions leverage the SAP Cloud Platform as extension platform. For both approaches, we'll give examples as well as typical use cases that use both options in combination.

2.1 Separation of Concerns

In this section, we'll introduce how the new extensibility strategy differs from the well-known traditional approaches that were possible in the traditional architecture of the past. Faster innovation cycles demand a different approach that gives appropriate freedom both to SAP and to extension developers.

2.1.1 "The Good Old Days"

For decades, the SAP NetWeaver Application Server (AS) for SAP ERP was used to develop and run extensions, from simple enhancements like adding some validation logic on a business object to huge add-ons. Several circumstances led to the perception that developing extensions was simple. Let's explore these in the next sections.

Integration

SAP's ERP modules were deeply integrated because they ran on the same application servers and shared the same database instance. Custom applications and extensions could use this tight integration in the same way as SAP did, which made upgrades more difficult.

Access to Source Code

The ABAP source code and the data dictionary (that is, the metadata) of the database was fully available to customers and accessible for extension development. Many customer extensions copied major parts from existing SAP modules or reused code from the modules so that customers could easily replace parts in their own implementations, which again made lifecycle management more difficult.

Infrastructure

Customers often deployed extensions into the existing infrastructure so that the extension code ran on the same infrastructure as the SAP modules. By doing so, purchasing hardware, installing servers, and managing users as well as other administrative tasks were avoided.

Change Management

SAP's change and transport management was for many years an unmatched advantage of the SAP NetWeaver AS. SAP has shaped many industry best practices with its

way of handling multiple software versions and the infrastructure of transporting changes from development to quality and production systems.

ABAP

ABAP is SAP's programming language, which was designed exclusively for business application development and features all the capabilities generally expected from a fourth-generation language. SAP has added many frameworks, libraries, and tools, such as the internal table concept or business partner APIs so that application and extension development was efficient. Consequently, SAP also made ABAP available to customers and partners for extension development.

Reliability

The SAP NetWeaver AS introduced the client-server approach to enterprise software, which is stable and reliable. By simply adding more application servers, SAP NetWeaver AS can scale from the smallest installations to the demands of the largest enterprises in the world.

2.1.2 Principle of Least Knowledge

In SAP S/4HANA on-premise, all capabilities mentioned in the previous section are still available and valid, although SAP imposes a more strict separation of internal and external APIs (public model) as explained in Chapter 1. Customers are strongly advised to build on the public model only and to avoid using or removing any other software artifact so that future upgrades can be executed smoothly.

In contrast, SAP S/4HANA Cloud exposes the public model only. The basic approach taken by SAP can be expressed with the *principle of least knowledge*, also known as the *law of Demeter*, introduced by Ian Holland in 1987. This principle postulates that any consuming component may not know the internal details of another provider component, but only uses its exposed functionality. Based on this principle, SAP can regularly update SAP S/4HANA Cloud without any adaptation needed on the part of customers and partners, enabling a faster innovation cycle. Consequently, long and error-prone upgrade procedures are significantly reduced, which increases customers' ability to benefit from new innovations faster. Furthermore, even for SAP, the violation of this principle reduced the ability to optimize the internals of the modules. Thus, code and objects created by SAP, by customers, and by partners code must be clearly and logically separated. That is, customers can no longer modify objects delivered by SAP in SAP S/4HANA Cloud.

While all traditional modification techniques are still possible in the on-premise version of SAP S/4HANA, to support the migration to SAP S/4HANA, we strongly recommend that customers *not* use these capabilities anymore. Technically, the separation of concerns by using decoupled extensions is one of the most important aspects of SAP S/4HANA and one of its major differences with SAP ERP and SAP Business Suite.

However, we'll need to determine the technologies, tools, and methods to use in conjunction with SAP S/4HANA if the traditional approaches are either no longer supported or are at least discouraged. In general, SAP S/4HANA provides two main types of extensions, which implement the separation of concerns:

- In-app extensions
- Side-by-side extensions

In-app and side-by-side extensions can be used individually but also in a seamlessly integrated way. As discussed earlier, in SAP S/4HANA Cloud, these options are the only ones available today. In contrast, in SAP S/4HANA on-premise, customers may also use traditional extension approaches, for example, using ABAP development tools such as ABAP in Eclipse and the ABAP Workbench (Transaction SE80). However, the traditional approach is no longer recommended for new developments and, therefore, is not covered in this book at all.

2.2 In-App Extensions

All in-app extensions are technically implemented inside the core of SAP S/4HANA, that is, on the same servers as SAP S/4HANA. As a result, no remote communication between the extension and the extended app is required. Many processes in SAP S/4HANA support extensions with key user tools integrated into the SAP Fiori user interface for SAP S/4HANA. As such, in-app extensions enable business experts without deep technical knowledge, typically referred to as *key users*, to implement key types of customer extensions, such as creating custom business objects and adding custom fields to business objects.

Note that the in-app extension concept is essential for SAP S/4HANA Cloud. However, this concept is also invaluable for traditional, on-premise deployments in order to align with the separation of concerns philosophy. With the inevitable move to the cloud, we recommend thinking of extensions from a cloud-first perspective even when applications are still running on-premise. In other words, in-app extensibility features should also be considered in SAP S/4HANA on-premise.

Definition: Key User

A key user in SAP S/4HANA is a member of a functional team that adopts the software to the needs of his or her business department—being either a direct employee or an external person, such as a consultant. As this person typically has some level of technical knowledge but no in-depth knowledge like a developer, all extension steps need to be covered by tools that hide complexity and technical detail. The old, strict separation of requestors stating requirements and developers implementing those requirements is no longer the preferred model for many SAP customers. Especially in software-as-a-service (SaaS) applications, for simple cases, customers expect to create custom forms, reports, fields, and even small pieces of business logic with appropriate tools that hide the technical complexity involved in connectivity, deployment, or lifecycle management.

This set of tools applies consistently across all applications in SAP S/4HANA; in other words, there is exactly one way to add custom logic to a business object or to extend a business object's data model with a field. This unification was achieved through SAP S/4HANA's overall streamlined architecture. The layers of a modern SAP Fiori application in SAP S/4HANA are shown in Figure 2.1.

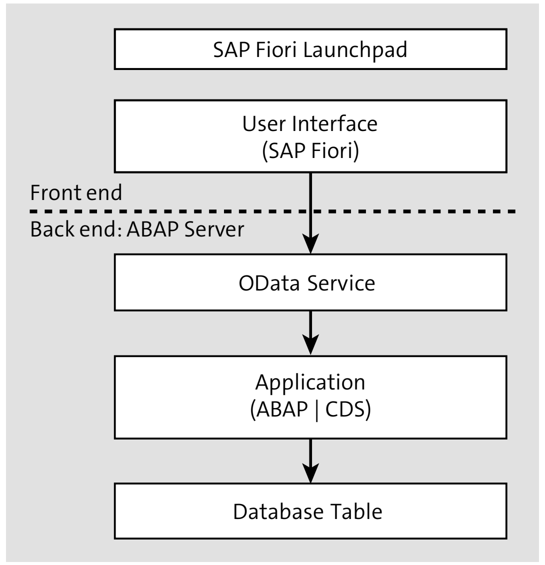


Figure 2.1 Anatomy of an SAP Fiori Application

An application consists of five layers. At the bottom, the database model contains database tables, which are used by an application layer. In the application layer, the core data service (CDS) model contains views for efficient consumption and analytical requests. In addition, this layer may contain ABAP source code, organized in business objects, for example, to implement validation logic. On the third layer from the bottom, the functionality of the business objects is exposed in the form of RESTful Open Data Protocol (OData) services, which can be considered the communication layer. At the top are the responsive SAP Fiori user interface and the SAP Fiori launchpad.

Open Data Protocol

OData was originally developed by Microsoft but today falls under the leadership of the OASIS organization. OData is an open protocol that standardizes the creation and use of RESTful HTTP communication APIs. OData client and server libraries are available for all major programming languages.

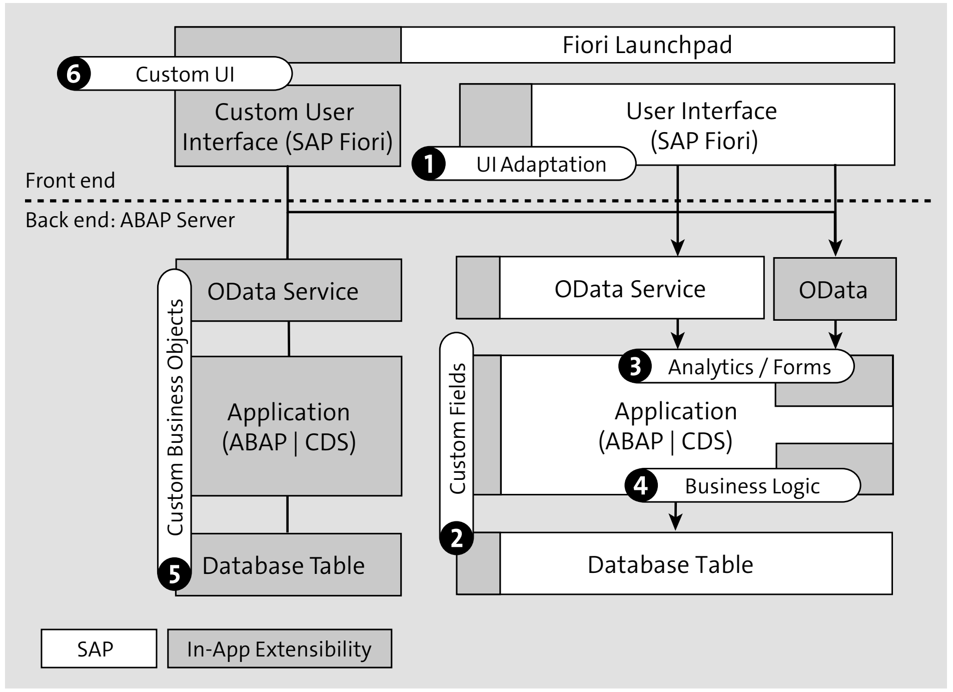


Figure 2.2 In-App Extensibility Points

On any layer, customers may apply in-app extensions. The diagram shown in Figure 2.2 provides an overview of the most important in-app extension points. We'll go through each extension point following the outlined numbering schema. Note that all the concepts we'll mention next will be described in more detail in Chapter 18 of this book.

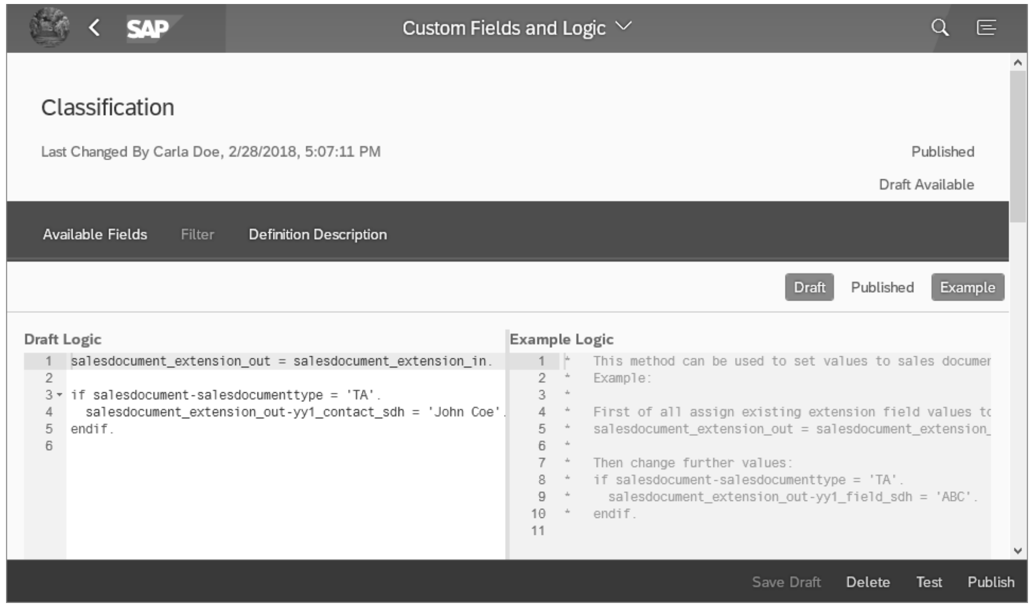
❶ **User Interface Adaptation**
Key users may change the layout of tables and forms directly in the running user interface. A special graphical user interface adaptation mode provides the mechanisms to hide fields in existing forms, tables, or filters; to rename labels; to add fields to the user interface from the field repository; and to move fields or entire blocks to other sections of the screen.

❷ **Custom Fields**
In the Custom Fields and Logic SAP Fiori application, you can add and edit custom fields to extend SAP database tables, CDS views, and OData APIs. Moreover, this extension application helps to define how custom fields are used in user interfaces, reports, or forms. Furthermore, you can use custom fields along with predefined business scenarios so that all involved business objects are extended and values are passed along automatically.

❸ **Custom CDS Views, Analytics, Forms**
The SAP S/4HANA data model, with its public views based on CDS, might not always provide the best technical option to fetch data. When related data needs to be pulled out of various views, pushing the join and aggregation operators into the SAP HANA database by creating a custom view might be more convenient or efficient.

The Custom CDS View application provides an overview of all CDS views in the public model as well as custom CDS views. This app enables customers to create new views based on an existing view model supporting features like associations, joins, transformations, field relabeling, selections, etc. With OData APIs on top of these views, customers may, in addition, expose the data for consumption outside the SAP S/4HANA core. Furthermore, the custom views might be used as data sources to define analytical queries. Those queries are used in embedded analytic user interfaces integrated into SAP S/4HANA applications. Another use case for custom views is to build forms with the Forms Designer application. Forms can consume data from custom OData services and use the data, for example, in custom email templates.

❹ **Business Logic**
Many SAP S/4HANA applications have enhancement spots (also called Business Add-Ins [BAdIs]). With the ABAP web editor (Figure 2.3), customers may create custom logic for the BAdIs. Typically, customers implement additional checks, set default values, or create mappings in combination with custom fields. The available features of the ABAP language in the web editor are limited compared to the capabilities in the SAP NetWeaver AS to ensure no negative impact on the robustness of the system as well as to comply with high security and data consistency standards. For example, you cannot perform database updates or generate source code dynamically from the web editor.



Draft Logic

```
salesdocument_extension_out = salesdocument_extension_in.

if salesdocument-salesdocumenttype = 'TA'.
  salesdocument_extension_out-yy1_contact_sdh = 'John Coe'.
endif.
```

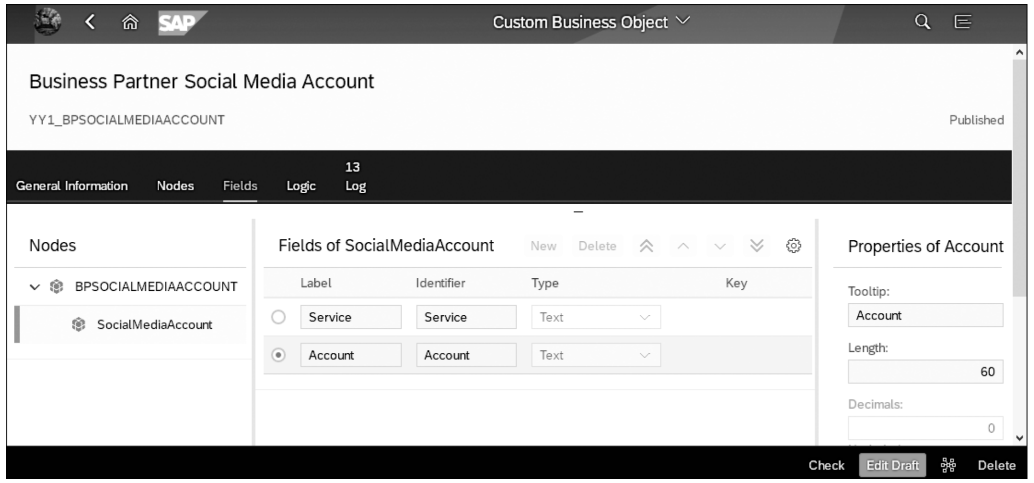
Example Logic

```
*   This method can be used to set values to sales documer
*   Example:
*
*   First of all assign existing extension field values tc
*   salesdocument_extension_out = salesdocument_extension_
*
*   Then change further values:
*   if salesdocument-salesdocumenttype = 'TA'.
*     salesdocument_extension_out-yy1_field_sdh = 'ABC'.
*   endif.
```

Figure 2.3 Custom Fields and Logic SAP Fiori Application

❺ **Custom Business Objects**
Besides custom views, you can create completely new business objects using the Custom Business Objects application. This application helps define the business object structure in the form of business object nodes, which automatically create the required database tables (Figure 2.4). A built-in web editor supports with writing the corresponding code. Again, for writing customer business objects, you can only use whitelisted APIs.

Changing existing SAP business objects is not supported. However, the creation of custom business objects can be supported by reusing custom CDS views, OData APIs, and custom user interfaces so that a fully-fledged, self-contained extension to SAP S/4HANA can be created.



	Label	Identifier	Type	Key
○	Service	Service	Text	
●	Account	Account	Text	

Tooltip: Account
Length: 60
Decimals: 0

Figure 2.4 Creation of a Node in a Custom Business Object

❻ **Custom User Interfaces**
You can use the SAP Web IDE to create your own SAP Fiori user interfaces either from scratch or based on templates. The SAP Fiori user interfaces consume SAP S/4HANA's RESTful OData interfaces. After creation and successful testing, you can deploy these user interfaces into SAP S/4HANA's user interface repository and integrate them into the customer's SAP Fiori launchpad as new tiles using the Custom Tiles and Custom Catalog Extensions application.

In SAP S/4HANA Cloud, the transport of adaptations from the test system to the production system is also performed by the key user with another SAP Fiori application. These tools can be used without interaction with the service provider and outside the service provider's maintenance window. The extensibility transport tools are available as part of the Manage Software Collection application.

Thus concludes our overview of in-app extensibility as one major form of SAP S/4HANA extensibility. More details on in-app extensibility and its intersection with side-by-side extensibility can be found in Chapter 20 of this book. Furthermore, we can recommend some additional literature on this topic in the box below.

Further Information on In-App Extensions

Online Help on Extensibility: *http://tiny.cc/sap-help-in-app*

SAP S/4HANA Extensibility Tutorial: *http://tiny.cc/in-app-tutorial*

SAP S/4HANA Extensibility: Use Case Overview: *http://tiny.cc/in-app-overview*

Now, we'll switch perspectives to building side-by-side extensions with SAP Cloud Platform and discuss the implications of the side-by-side approach.

2.3 Side-by-Side Extensions

Scenarios exist where in-app extensions are not sufficient, which we'll explain in Section 2.4. In such cases, customers and partners may use side-by-side extensions as an alternative approach. The name stems from the fact that, in contrast to in-app extensions, side-by-side extensions are only implemented using the SAP Cloud Platform.

2.3.1 Overview of the SAP Cloud Platform

The SAP Cloud Platform is a platform-as-a-service (PaaS) offering for creating applications or extensions in a secure cloud-based computing environment managed by SAP. It is the default choice for building side-by-side extensions to SAP S/4HANA Cloud or on-premise. The platform also serves extension cases for other SAP products, be it other on-premise applications or the modern LOB-specific SaaS solutions such as SAP SuccessFactors (Figure 2.5). With SAP Cloud Platform, SAP offers a product as well as a methodology to develop technically decoupled extensions and cloud applications to meet challenging requirements in terms of cloud qualities such as availability and scalability, which we'll explain in Chapter 3.

SAP Cloud Platform provides a broad range of managed services as well as open source technologies that can be used to build new solutions. Business services include a currency service, a translation hub, and business logging. Furthermore, SAP Cloud Platform contains a broad range of technical services such as SAP Cloud Platform Integration, SAP Leonardo Internet of Things (IoT), or SAP Leonardo Machine Learning to name just a few. Finally, SAP Cloud Platform also provides storage services such as SAP HANA and open-source technologies like Redis or PostgreSQL. The entire list is continuously extended with new services on all levels of the platform.

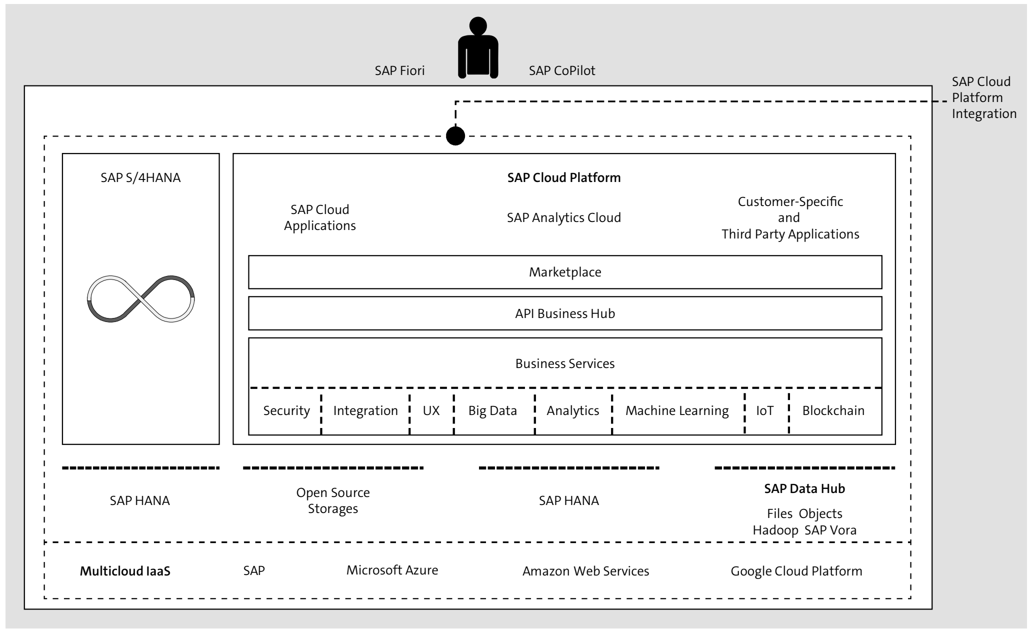


Figure 2.5 High-Level Overview of SAP Cloud Platform as the Extension Platform

In addition, SAP provides several programming environments and runtimes. First, the SAP Cloud Platform Neo environment is an SAP-proprietary environment for Java and HTML5 applications. Second, various runtimes are available as part of the Cloud Foundry environment on the SAP Cloud Platform, such as Java, Node.js, Python, and basically any other runtime supported by Cloud Foundry. Cloud Foundry is an open source industry standard to provide a PaaS standard that can run on multiple clouds. Cloud Foundry helps automate, scale, and manage cloud applications throughout the application lifecycle. SAP supports the Cloud Foundry Foundation as a Platinum member with code contributions and helps advance the evolution of open cloud computing technologies along with other companies such as Cisco, Dell, IBM, Pivotal, Suse, Google, and Microsoft. Complementing the existing capabilities of Cloud Foundry, SAP offers tools for developing applications such as the SAP Web IDE, the SAP API Business Hub, and an operations cockpit as well as several services for application management and monitoring. Figure 2.6 presents a high-level overview of the capabilities that are part of the SAP Cloud Platform. You may check out the overview of capabilities on the SAP Cloud Platform web page (*http://tiny.cc/scp-services*) for a more recent picture since the portfolio continuously expands.

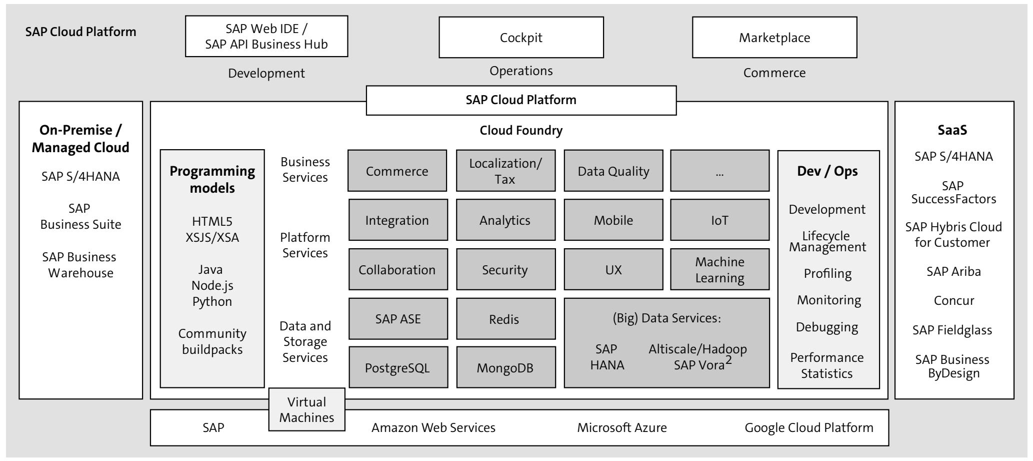


Figure 2.6 High-Level Overview of SAP Cloud Platform Capabilities

Thanks to SAP's multicloud strategy, you can use SAP Cloud Platform on multiple infrastructures, such as SAP's own data centers, Amazon Web Services, Microsoft Azure, and Google Cloud Platform. Customers can use SAP Cloud Platform in combination with other cloud services or co-located to existing cloud infrastructure and services.

2.3.2 Connectivity between SAP Cloud Platform and SAP S/4HANA

Connectivity between SAP Cloud Platform and SAP S/4HANA Cloud uses HTTPS communication, mostly via OData APIs. SAP S/4HANA Cloud exposes public APIs that can be directly called from applications on SAP Cloud Platform, and vice versa. Connectivity between SAP Cloud Platform and SAP S/4HANA on-premise is often more complex because you may not want to expose your internal networks for direct internet access. For this purpose, SAP offers the Cloud Connector to establish a secure tunnel for the communication. The tunnel can support ODBC, RFC, and HTTPS communication protocols so that you can, for example, invoke classic BAPIs, perform data replication from the database, or call OData APIs. The Cloud Connector is a small server installed inside the customer's landscape and acts as a reverse proxy to establish a connection with the connectivity service on SAP Cloud Platform. Once the connection is established, the Cloud Connector accepts calls from SAP Cloud Platform and routes them through to the correct endpoint of one of the on-premise systems, as shown in Figure 2.7.

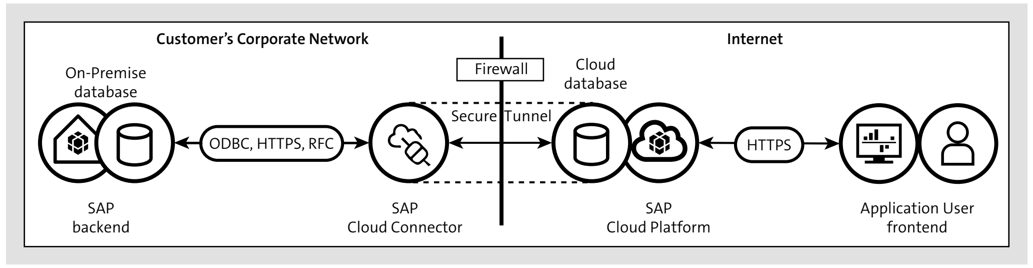


Figure 2.7 The Cloud Connector in a Hybrid Scenario

We'll go into more detail on how to set up the connection between SAP S/4HANA and the SAP Cloud Platform in Chapter 6. We'll also include detailed instructions on the differences between SAP S/4HANA Cloud and SAP S/4HANA on-premise as well as discuss some security concerns with delegating business users between the two stacks.

2.3.3 Side-by-Side Extension Scenarios

Side-by-side extensions are by design technically decoupled from the SAP S/4HANA standard. Consequently, an extension on the SAP Cloud Platform can be scaled and delivered independently from SAP S/4HANA. The extension may even provide its business function in case SAP S/4HANA or the connection to it has limited availability or other technical issues. Highly agile teams may use side-by-side extensions to update and improve parts of the system without close alignment with SAP S/4HANA teams and release plans. The following patterns cover the most important extension points next to the ones already introduced for in-app extensions. The patterns themselves consider typical, disjointed usage scenarios and, thus, do not reflect every possible combination that might be possible across patterns. Thus, you'll have to pick and combine various patterns depending on your use case and requirements.

Custom User Interfaces

The SAP Web IDE provides the tools for customers to create web-based user interfaces. The SAP Web IDE puts special focus on SAP Fiori and SAPUI5 with many templates for commonly used patterns such as the master-detail page layout. The user interfaces can then be deployed to an SAP Cloud Platform account for testing purposes or for productive use. The user interfaces may use existing whitelisted OData services from SAP S/4HANA (Figure 2.8). This functionality might be especially helpful for you to copy existing SAP standard user interfaces as a starting point.

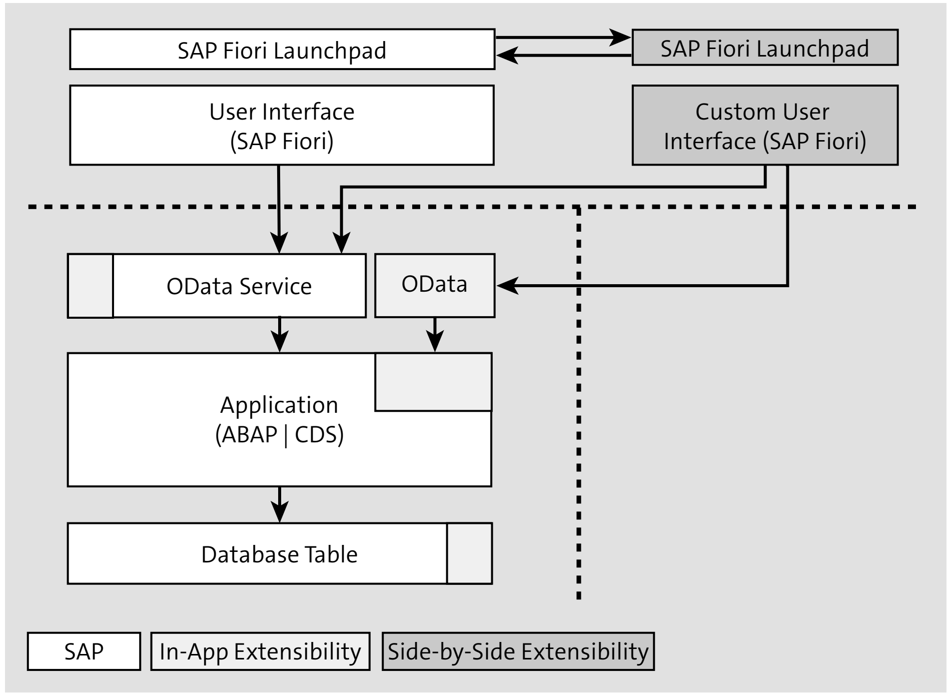


Figure 2.8 SAP Fiori User Interface as a Side-by-Side Extension

However, you can combine in-app extensions with side-by-side extensions by creating new OData services inside the core of SAP S/4HANA that can be consumed from SAP Cloud Platform. As explained earlier, such OData APIs may either serve additional data, for example, from a custom business object, or can combine data from several existing views so that roundtrips between the user interface and the backend are reduced (Section 2.2).

Custom Applications

As shown in Figure 2.9, you can also create completely new applications spanning across user interfaces, application logic, and the storage layer. For partner applications, this structure is expected to be the dominant extension pattern. On all layers, the SAP Cloud Platform provides numerous technologies and tools so that customers or partners may choose the best ones to fulfill their requirements. Although the

Cloud Foundry environment on the SAP Cloud Platform allows you to write applications in almost any programming language, SAP is primarily investing in Java, JavaScript, and ABAP. For example, SAP provides libraries to create OData services or to use the SAP S/4HANA public data model inside Java and JavaScript backend services. A good part of these libraries and options, such as the SAP S/4HANA Cloud SDK, are covered in this book.

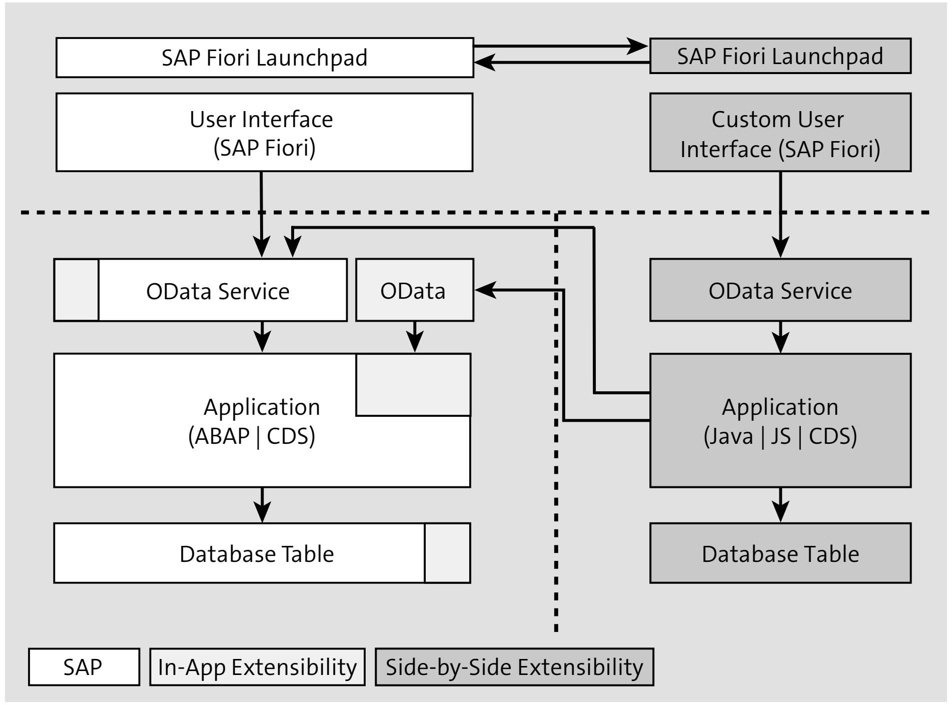


Figure 2.9 Side-by-Side Extension App on SAP Cloud Platform

Again, the extension application may integrate with SAP S/4HANA Cloud through existing public APIs or by additional APIs created with in-app extensions. In addition, SAP S/4HANA on-premise exposes APIs in the form of BAPIs using the RFC protocol or the HTTP-based SOAP protocol.

Side-by-Side Service and Events

Sometimes, an in-app extension, in the form of a code extension or a custom business object, may require capabilities not provided by SAP S/4HANA. For this purpose, customers may implement a service on the SAP Cloud Platform and invoke the service by a RESTful call from the in-app extension to the SAP Cloud Platform service, as shown in Figure 2.10.

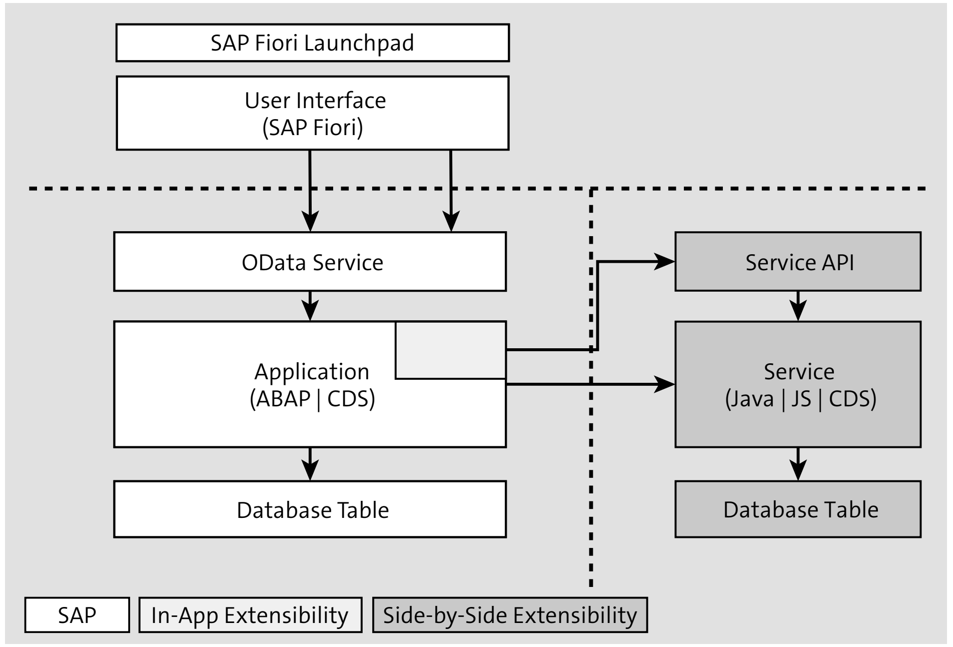


Figure 2.10 Side-by-Side Extension Service on SAP Cloud Platform

Besides RESTful service calls, you can also emit events in SAP S/4HANA that can be consumed on the SAP Cloud Platform. Events contain messages with data similar like in a service call. The main difference, however, is the asynchronous nature of an event. The process continues after emitting the event without waiting for a response. The eventing capabilities between SAP S/4HANA and the SAP Cloud Platform will be explained more detail in Chapter 16.

Data Mart Extension

Analytical applications implementing data mart-like scenarios, may need to bring data together in one database so that the response to an analytical query is fast (Figure 2.11). With SAP HANA being the database on both sides, you can replicate data from the SAP S/4HANA system into the database of the analytical application on the SAP Cloud Platform. The basis for the replication is the public model so that, after replication, the same views can be used in selects and joins. To use the

replication mechanisms with SAP S/4HANA Cloud, a fully managed SAP service exists. At the time of this writing, the service is in restricted mode, accessible only to pilot customers.

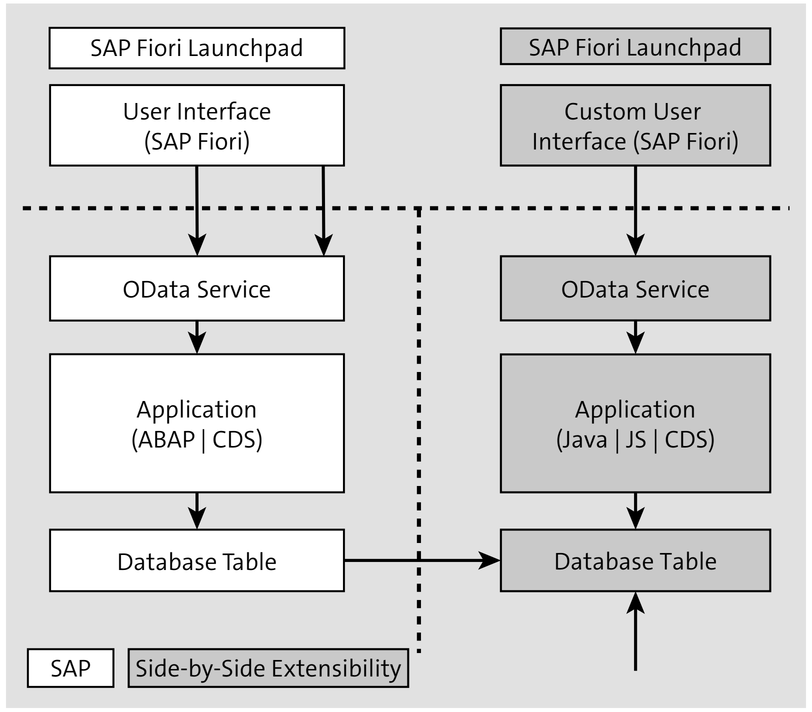


Figure 2.11 Data Mart on SAP Cloud Platform as a Side-by-Side Extension

For SAP S/4HANA on-premise and other SAP solutions, customers can perform the replication with help of the Cloud Connector and, optionally, also SAP Landscape Transformation Replication Server. Further, you can copy data from various other sources into the analytical application using RESTful services so that customers can join data across business applications and processes as part of the analytical application.

Further Information on Data Replication to SAP Cloud Platform

Replicate ABAP CDS Views from SAP S/4HANA Cloud to SAP Cloud Platform: *http://tiny.cc/data-replication*

Replicating Back-End Data to the Cloud: *https://sap.github.io/cloud-s4ext/week-4/unit-3/*

This concludes our review of the general patterns possible using the side-by-side extension approach. In the next section, we'll discuss typical extension use cases based on the presented patterns and principles.

2.4 Extension Use Cases

With the side-by-side and in-app extensibility features explained in the previous sections, you can build additions to SAP S/4HANA that significantly increase the reach and the scope of the system. SAP S/4HANA can be connected in processes that engage internet users, such as in an online store, on mobile devices, or via experimental user interface technologies. Furthermore, you may extend existing processes but also invent completely new processes while easily combining SAP S/4HANA data with data from other systems. As a result, we've observed several typical extension archetypes that our customers or partners are implementing, as follows:

- Proxy applications
- Convenience applications
- Substitute applications
- Preprocessing applications
- Postprocessing applications
- Analytical applications

Note that this list does not claim to be exhaustive and you can combine several of these archetypes within one process or even within one application. The presented archetypes are typically driven from a side-by-side extension perspective. However, in most real-life scenarios, they are complemented and supported by in-app extensions.

Let's discuss these archetypes individually next.

2.4.1 Proxy Applications

Proxy applications are highly connected applications that shield the SAP S/4HANA system from internet traffic. In other words, these applications "buffer" data and communicate with the SAP S/4HANA in batch mode for more efficient data exchange. These applications often work with identities only known in the proxy application but not in SAP S/4HANA, for example, the end consumers of a retail enterprise who may not require an SAP S/4HANA backend user. Typical proxy applications include online stores, registration web sites, or any other publicly available web sites. A proxy application

on a mobile phone could also be used by customers to learn about promotions and products. These applications typically have high requirements with respect to availability and scalability to cater for traffic bursts at peak times.

2.4.2 Convenience Applications

Convenience applications aim at simplifying the user experience and, thus, often do not add many new features but instead make using existing features significantly easier. You can, for example, build a convenience application for a mobile device to support field engineers. In a convenience application, fields can be defaulted to the most commonly used values or can be hidden altogether because reasonable values can be provided automatically.

2.4.3 Substitute Applications

Substitute applications replace a specific process or process step in SAP S/4HANA. The reasons to use a substitute application might be to provide a functionality missing in SAP S/4HANA or existing processes that shouldn't be changed when deploying SAP S/4HANA. Substitute applications typically execute service calls on SAP S/4HANA in the background while end users can work in a completely different user interface with capabilities that are not part of SAP S/4HANA's scope.

For example, let's consider time recording in the service industry. Typically, external tools are used for project management and scheduling, such as Atlassian Jira. Jira ticket numbers and other fields can be added to the SAP S/4HANA data model using an in-app field extension for simplified end-to-end reporting of SAP S/4HANA. Then, the substitute application on SAP Cloud Platform may join the information from SAP S/4HANA (i.e., costs from a cost center) with information from Atlassian Jira (i.e., project times, statuses, and schedules).

2.4.4 Preprocessing Applications

Preprocessing applications are often used to collect data before a process is started in SAP S/4HANA. In preprocessing applications, data is collected until a certain status or state is achieved. Then, a process in SAP S/4HANA is started using the collected data. An example might be an app to create and maintain new products as a side-by-side extension of SAP S/4HANA. Once details about the new product have been approved, the product is promoted to and stored in SAP S/4HANA. Subsequently, SAP S/4HANA processes are triggered such as pricing or production-related activities.

2.4.5 Postprocessing Applications

Postprocessing applications receive events from SAP S/4HANA when certain processes or process steps are completed. Based on these events, additional activities are performed in a side-by-side extension of SAP S/4HANA. An example could be an application that updates a customer record. If the customer relationship ends, data may have to be deleted from corresponding applications or data stores hosted on the SAP Cloud Platform.

2.4.6 Analytical Applications

Analytical applications connect data from multiple sources into one analytical database so that analytical queries can be answered quickly and interactively. Ad hoc reporting is also available. For this purpose, data from SAP S/4HANA and other systems might be replicated into a SAP HANA database on SAP Cloud Platform. Then, analytical applications can leverage all the capabilities provided by SAP HANA, such as columnar storage, in-memory execution, and rich view building as well as the analytical user interface controls of SAPUI5.

2.5 Nonfunctional Requirements of Extension Applications

So far, we've presented extension archetypes and described common use cases for extensions from a functional perspective. Functional requirements specify the behavior of the system, that is, what it is supposed to do. However, other, nonfunctional requirements also exist that relate to the underlying qualities of the system such as performance, costs, availability, or resilience.

By definition, in-app extensions inherit their nonfunctional properties from SAP S/4HANA. In contrast, for side-by-side extensions, customers and partners are mainly responsible for ensuring certain qualities such as scalability and availability. To achieve these qualities, customers are supported by appropriate SAP Cloud Platform services, software development kits (SDKs), and best practices, many of which we'll explain in this book. However, not all qualities are equally important for all extensions. Some business cases may also not justify the efforts to build highly scalable and available extensions. Therefore, you'll need to understand the design trade-offs, when to invest, and how to achieve a high degree of quality.

Generally, we recommend identifying extensions that are mission critical; any failure in this extension would have a severe impact on business operations. For example,

customers often consider employee self-service applications less critical than applications involved in core business processes. In certain situations, such as a leave request, an employee may accept poor user interface performance and simply retry at a later point in time. On the other hand, an application intended to be used by the customers of the business such as an online shop or a customer-facing mobile application can have severe impact on the business success when customers cannot place orders (revenue loss) and instead select competitive products. This distinction between mission-critical and non-mission critical capabilities can help you find the right trade-offs.

Mission critical applications are business processes that involve customers, partners, suppliers, etc., for example, an internal application the sales force needs for daily operations. Mission critical applications may also target different users or interact with systems that are not under the control of the organization. Often, non-mission critical applications accept a much lower level of certain nonfunctional requirements for the sake of lower development or operations costs. In mission critical applications, certain nonfunctional requirements such as availability cannot be sacrificed, and higher development and operational costs might be acceptable.

Typical properties of internal applications include the following:

- Known user (e.g. SAP S/4HANA user), single-sign on
- Predictable scalability (e.g. based on number of users)
- SAP Fiori UI with moderate usability requirements
- Downtimes and times of limited availability are considered acceptable

Typical properties of external applications include the following:

- Known user and internet user (not a user in one of the organization's systems)
- Unknown scalability needs or high peaks (e.g. due to seasonal events), load impact on internal systems unclear
- SAP Fiori and freestyle UIs (e.g. created by an agency)
- Downtimes unacceptable (24/7) with continuously high performance

Based on these considerations and to best serve business needs, we recommend building external applications as side-by-side extensions to cope with the additional nonfunctional challenges introduced by cloud software. For internal applications, a careful trade-off analysis should be conducted to identify whether the functionality might be provided by either a pure in-app extension or by a side-by-side extension.

The criteria above may guide you in finding the appropriate trade-offs for your application scenario.

We'll further explore the qualities of cloud-native application development, the predominant development style for side-by-side extensions, in the next chapter.

2.6 Summary

In this chapter you learned about the various options you have for extending SAP S/4HANA. In the next chapter we'll dive into the specifics of side-by-side extensibility and cloud-native application development. You'll learn how the principles of cloud-native development, such as DevOps, microservices, continuous delivery, and containerization, can help you craft highly scalable, resilient, and consistently available applications on the SAP Cloud Platform.

Chapter 5
Application Security

This chapter outlines how to secure our example application, which consists of several microservices in a multitenant fashion. This chapter involves an overview of the architecture and the interaction among the corresponding components such as the application router (AppRouter) and the Identity Provider (IdP), as well as other relevant microservices. In addition to how to configure authentication and authorization between services and users, we'll cover essential security fundamentals and web security topics such as cross-site request forgery (CSRF) and clickjacking.

In this chapter, we'll introduce you to all concepts that you need to understand to secure your application in the Cloud Foundry environment on the SAP Cloud Platform. Before we dive more deeply into implementation details, in this chapter, we'll provide a comprehensive overview of the architectural components (e.g., the AppRouter, Extended Services for User Account and Authentication [XSUAA]) and patterns (e.g., OAuth) you'll need to understand for the full conceptual picture. Based on this overview, we walk through step-by-step how to protect our Business Partner Address Manager application so that only authenticated and authorized users can use the application. Additionally, we'll introduce some common web application threats, such as cross-site request forgery and clickjacking, and discuss how SAP technologies can help you to reduce your application's vulnerability to these threats.

5.1 Security on SAP Cloud Platform

Figure 5.1 presents the general architectural setup of security configuration in the Cloud Foundry environment on the SAP Cloud Platform. So far in this book, we learned about the single Java-based microservice that comprised our application, which consists of a few backend APIs and the UI (a micromonolith if you will). However, instead of accessing this service directly, we'll now use the AppRouter, which serves three main purposes.

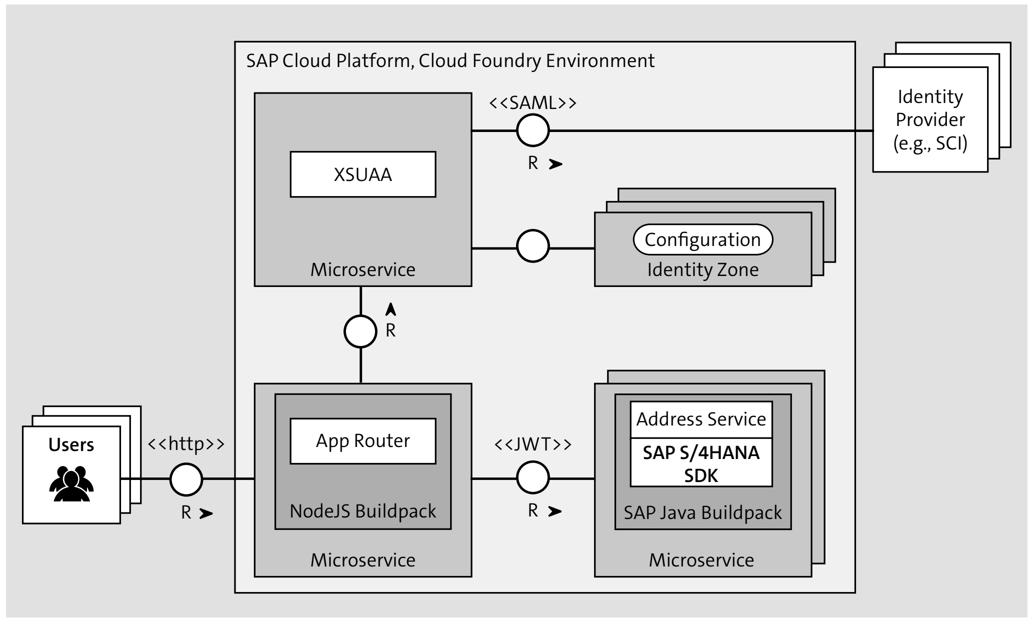


Figure 5.1 High-Level Authentication Setup with AppRouter and Extended Services for User Account and Authentication (XSUAA)

First, the AppRouter is the central entry point for our application into the world of real microservices. We can dissect our application into multiple microservices while hiding the resulting complexity from our end users. As such, the AppRouter dispatches requests to our backend microservices, thus acting as a reverse proxy, in particular, since the backend microservices should not be directly accessible by the client.

Second, the AppRouter can serve static content such as web pages, SAPUI5, or any other client-side code. We'll postpone discussion about static content until Chapter 14.

Third, the AppRouter is an important component for managing the authentication flows for our entire application and is, in addition, capable of strengthening our application against common web application threats. We'll focus on these features throughout this chapter.

For the purposes of authentication (who the user is) and authorization (what the user is allowed to do), the AppRouter takes all incoming, unauthenticated request and initiates an OAuth2 flow (authorization code grant) with the *Extended Services for User Account and Authentication (XSUAA)* service of the SAP Cloud Platform in the Cloud

Foundry environment. The XSUAA is a specific service provided by SAP to deal with authentication and authorization for business applications instead of using the standard user account and authentication service available in the Cloud Foundry environment. By default, the XSUAA uses the SAP ID service as user provider managing all users of the SAP Cloud ecosystem such as public users (P-users) or support user (S-users). Customers may replace the default configuration with any other Security Assertion Markup Language (SAML) 2.0-compliant Identity Provider (IdP) like the SAP Cloud Platform Identity Authentication service.

The full runtime flow of all involved components for authentication is shown in Figure 5.2. If you're not comfortable with OAuth yet, we recommend studying the RFC at *https://tools.ietf.org/html/rfc6749*.

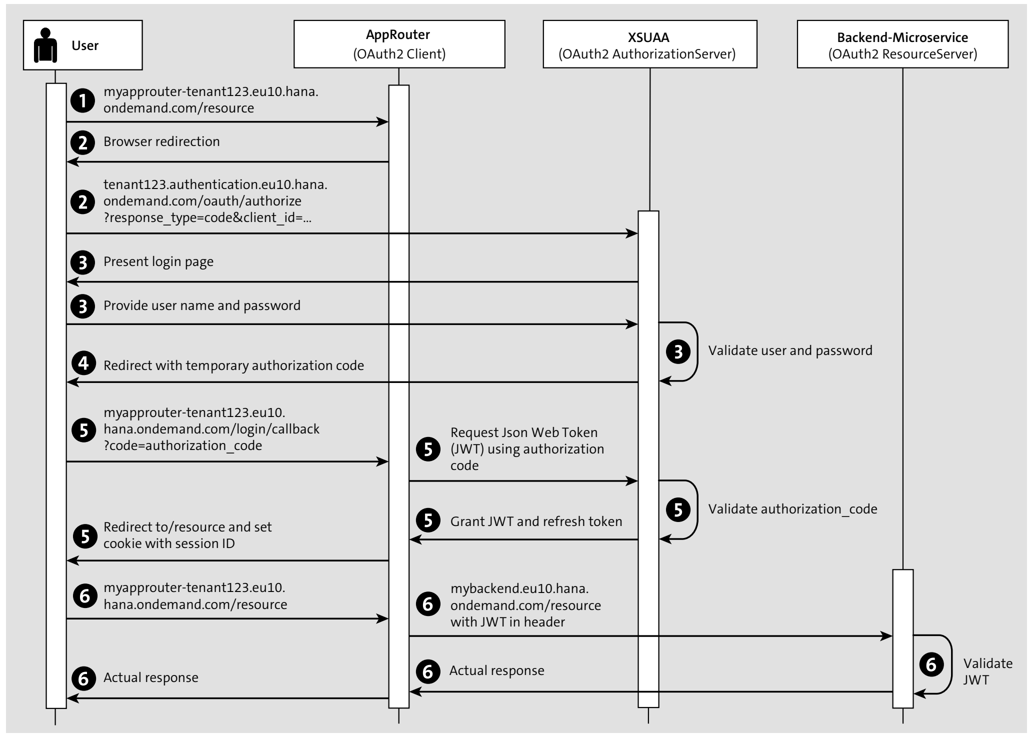

User
AppRouter (OAuth2 Client)
XSUAA (OAuth2 AuthorizationServer)
Backend-Microservice (OAuth2 ResourceServer)
1 myapprouter-tenant123.eu10.hana.ondemand.com/resource
2 Browser redirection
2 tenant123.authentication.eu10.hana.ondemand.com/oauth/authorize ?response_type=code&client_id=...
3 Present login page
3 Provide user name and password
3 Validate user and password
4 Redirect with temporary authorization code
5 myapprouter-tenant123.eu10.hana.ondemand.com/login/callback ?code=authorization_code
5 Request Json Web Token (JWT) using authorization code
5 Validate authorization_code
5 Grant JWT and refresh token
5 Redirect to/resource and set cookie with session ID
6 myapprouter-tenant123.eu10.hana.ondemand.com/resource
6 mybackend.eu10.hana.ondemand.com/resource with JWT in header
6 Validate JWT
6 Actual response
6 Actual response

Figure 5.2 Runtime Flow for Authentication in SAP Cloud Platform

In the first step of the runtime flow ❶, a nonauthenticated user may request a certain backend resource from one of our microservices using the AppRouter as the single entry point. Since the user is not authenticated, in the second step ❷, the AppRouter, acting as an OAuth client, will respond with a URL redirect to the XSUAA service, which serves as the OAuth authorization server. In Step ❸, the XSUAA now responds with a login page asking the user for a valid user name and password. If the login information is correct, the XSUAA responds with a redirect back to the AppRouter including a temporary authorization code in Step ❹. In the fifth step ❺, the AppRouter exchanges the authorization code against an access token, represented as a JSON Web Token (JWT). The JWT contains all the information about the user: the formal user name, the tenant the user belongs to, any granted authorizations as OAuth scopes, etc. The JWT is also digitally signed so that every involved party can check its validity and integrity. The AppRouter stores the received JWT in the user's session for future use. For more information on the JWT standard, refer to the official RFC at *https://tools.ietf.org/html/rfc7519*. In the sixth and final step ❻, the AppRouter will forward the original request with the granted JWT to our backend resource which, after also validating the validity of the JWT, will return the correct response payload to the user. Note that this entire flow needs to be repeated only when the session of the user inside the AppRouter expires or the user has issued an explicit logout.

As mentioned earlier, the JWT contains a signature that must be verifiable by every microservice constituting our application. Therefore, every service must maintain a service binding to the XSUAA service, which provides this information at runtime for verification purposes and to fulfill the OAuth flow (Figure 5.3).

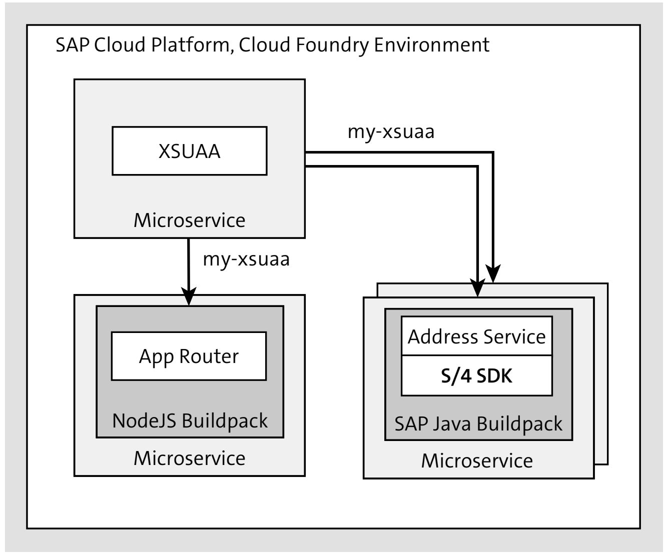


Figure 5.3 Runtime Binding between Microservices and XSUAA

Technically, every microservice needs aa binding to the specific XSUAA instance that writes this information into the `VCAP_SERVICES` environment variable, which the microservices can use to check the JWT's validity. With these basics in mind, let's create the setup shown in Figure 5.1 in the following sections.

5.2 Configuring Authentication

In the following section, we'll show you how to secure our Business Partner Address Manager application so that only authenticated users can access it.

5.2.1 Setting Up the Application Router

As a first step, we'll need to get and set up the AppRouter as explained in the previous section. The AppRouter is a Node.js component distributed via the publically available SAP NPM registry.

To download and install the AppRouter, follow these steps:

1. Go to your favorite *<destLocation>*, which represents your preferred development directory.

```
cd <destLocation>
mkdir approuter
cd approuter
```

2. Place the package.json shown in Listing 5.1 in the newly created *approuter* directory. Please note that we're using version 4.0.1, which is the most recent version at the time of this writing. You may update the AppRouter's version in this file to the latest version.

```
{
  "name": "approuter",
  "dependencies": {
    "@sap/approuter": "4.0.1"
  },
  "scripts": {
    "start": "node node_modules/@sap/approuter/approuter.js"
  }
}
```

Listing 5.1 The package.json for the Application Router

3. Install AppRouter dependencies using the following commands:

```
npm config set @sap:registry https://npm.sap.com
npm install
```

4. If successful, you should find a *node_modules* folder in the same location as the package.json.

Install via Service Marketplace

Alternatively, you may download the AppRouter from the SAP Service Marketplace at *http://tiny.cc/xsjavascript* and pick the latest XS JavaScript package, which includes the AppRouter among other dependencies. However, at the time of this writing, only an older version was available. We don't recommend this option.

Now, we need to provide some design-time configuration to the AppRouter. The main configuration file is called xs-app.json (Listing 5.2), which we create and place under *<destLocation>/approuter*.

```
{
  "welcomeFile": "index.html",
  "routes": [{
  "source": "^/api/(.*)",
    "target": "/api/$1",
    "destination": "app-destination"
  },{
  "source": "^/address-manager/(.*)",
  "target": "/address-manager/$1",
  "destination": "app-destination"
  }]
}
```

Listing 5.2 The AppRouter's Design Time Descriptor xs-app.json

So far, this file contains two routes to our backend microservice: one for our backend APIs and a second for static frontend content. During runtime, a route is identified through a matching pattern provided under the `source` path matched against the relative URL requested by the client. Then, the relative URL for the backend service is calculated based on the expression under `target` and combined with the base URL read from the `destination` variable, which is provided as an environment variable. In this

way, we can flexibly change destinations later at runtime when our deployment model changes but leave all design-time parameters constant.

The Application Router's Design-Time Descriptor: xs-app.json

The xs-app.json file can be used to configure many more aspects at design-time of the AppRouter like authentication types, cache control, compression threshold for outgoing traffic (by default, 1 KB), client-initiated logout, custom settings for CSRF protection, and much more. We'll use some of these settings throughout this book. However, we recommend studying the up-to-date specifications and more detailed settings at *http://tiny.cc/approuterconfig*.

To correctly deploy our AppRouter, we'll create an additional manifest.yml file inside *<destLocation>* as the next step with the example content shown in Listing 5.3.

```
---
applications:
- name: approuter
  routes:
    - route: approuter- ↩
        <subdomain>.cfapps.eu10.hana.ondemand.com
  path: approuter
  memory: 128M
  env:
    TENANT_HOST_PATTERN: 'approuter- ↩
          (.*).cfapps.eu10.hana.ondemand.com'
    destinations: '[{"name":"app-destination", "url": ↩
          "https://address-manager-unsparse-subutopian ↩
          .cfapps.eu10.hana.ondemand.com", ↩
          "forwardAuthToken": true}]'
  services:
    - my-xsuaa
```

Listing 5.3 Providing a Deployment Descriptor for the AppRouter

Note

Replace the `route` parameter with your subaccount's subdomain as well as the URL parameter of the `destinations` variable with the correct base URL of our previously deployed Business Partner Address Manager microservice. For example, if you created a trial account on SAP Cloud Platform previously with a P-user, the route of the AppRouter will look like `approuter-p1942765239trial`, and the URL to the microservice might be `address-manager-unsparse-subutopian` based on the random route generated in Chapter 4.

As we discussed in Chapter 4, we'll use the standard Cloud Foundry manifest.yml file to instruct the AppRouter to describe basic metadata for deployment as we did already for the Java-based backend microservice. Additionally, we'll utilize two important user-provided environment variables, which are understood by the AppRouter.

First, the `destinations` variable will declare the base URL of our `app-destination` destination that we referenced earlier in the xs-app.json. You must provide an absolute URI including the protocol (*https://*), without any relative information as that information will be used to identify routes as specified in *xs-app.json*. We'll also specify that any JWT from the AppRouter shall be forwarded to this destination.

Second, the `TENANT_HOST_PATTERN` variable will define how tenants should be identified in our application's URL. Upon request, the AppRouter will match the incoming base URL against the provided regular expression and will consider the first matching group of the expression as the tenant. Then the AppRouter will use the matched string to identify the corresponding XSUAA tenant. If you need a different URL scheme for your tenants, you'll need to change this pattern accordingly. As we assume to build multitenant applications throughout this book, this parameter is required but can be omitted if implementing a single-tenant application only.

The AppRouter's Runtime Configuration

Besides destinations and tenant host patterns, the AppRouter understands many different parameters that can be changed during runtime, without redeploying the entire AppRouter, which is consistent with the twelve-factor application principles outlined in Chapter 3. Recognized parameters include cross-origin policies, session timeouts, compression thresholds, JWT refresh times, etc. You can find a full list of these parameters at *http://tiny.cc/approuterparams*. Besides declaring these variables in manifest.yml, please note that you can always use `cf set-env <appName> <variableName> <variableValue>` to set them.

Finally, in manifest.yml, we introduced a binding to a service instance, called `my-xsuaa` in our example. Our XSUAA service instance will be bound to the AppRouter so

that JWTs issued by the XSUAA can be verified and, therefore, trusted by the AppRouter. Technically, this binding will issue a `VCAP_SERVICES` entry for the AppRouter, which holds all required information such as the client secret or the public key, to verify all security-related artifacts, as shown in Listing 5.4.

```
{
 "VCAP_SERVICES": {
  "xsuaa": [{
    "binding_name": null,
    "credentials": {
     "clientid": "<clientid>",
     "clientsecret": "<clientsecret>",
     "identityzone": "<subdomain>",
     "identityzoneid": "<subaccountId>",
     ...
     "verificationkey": "-----BEGIN PUBLIC KEY----- ...",
     "xsappname": "<xsappname>"
     ...
```

Listing 5.4 XSUAA Binding Information

However, before we can deploy the AppRouter, we'll need to instantiate our `my-xsuaa` service instance as described in the next section.

5.2.2 Extended Services for User Account Authentication (XSUAA)

With the AppRouter, we've already set up the largest part required for authentication. Besides acting as an OAuth client, the AppRouter is required for many other aspects in SAP Cloud Platform and, thus, is an important prerequisite. Eventually, authentication and authorization will be performed by the XSUAA. Therefore, we'll need a service instance to the XSUAA service, which is then bound to our AppRouter.

To create a service instance, we'll require the xs-security.json descriptor file, which we'll put into our initially chosen *<destLocation>* for the AppRouter:

```
{
  "xsappname": "address-manager-<subdomain>",
  "tenant-mode": "shared"
}
```

Like with the AppRouter, this file provides important design-time information such as roles, authorizations, etc., to the service instance we are about to create. For authentication purposes, the file looks simple as we are basically telling the XSUAA to create an instance with a unique identifier. As a best practice, we'll again use the application's name and subaccount's domain, but you can use any other arbitrary identifier not yet taken. Furthermore, we'll use the XSUAA in multitenant mode (`tenant-mode: shared`). We'll enhance this file in Section 5.3, when we introduce authorization concepts.

The Application Security Descriptor: xs-security.json

The xs-security.json descriptor is an important design-time artifact to provide authorization scopes, role templates, and other attributes of the application itself as well as foreign applications. For full specifications, refer to *http://tiny.cc/securitydescriptor*.

We'll then create a service instance called `my-xsuaa` of the XSUAA service by issuing the following command and using the xs-security.json file as a configuration input:

```
cf create-service xsuaa application my-xsuaa -c xs-security.json
```

By creating this instance, we are now explicitly declaring our XSUAA tenant as a platform-as-a-service (PaaS) tenant, that is, as a tenant whose security configuration can be shared with software-as-a-service (SaaS) tenants of consumers subscribing to our application later (see also Chapter 3 and Chapter 7). Thus, the consumers of our multitenant application can see, configure, and assign scopes and roles to their tenant-specific users (Section 5.3.2).

5.2.3 A First Test

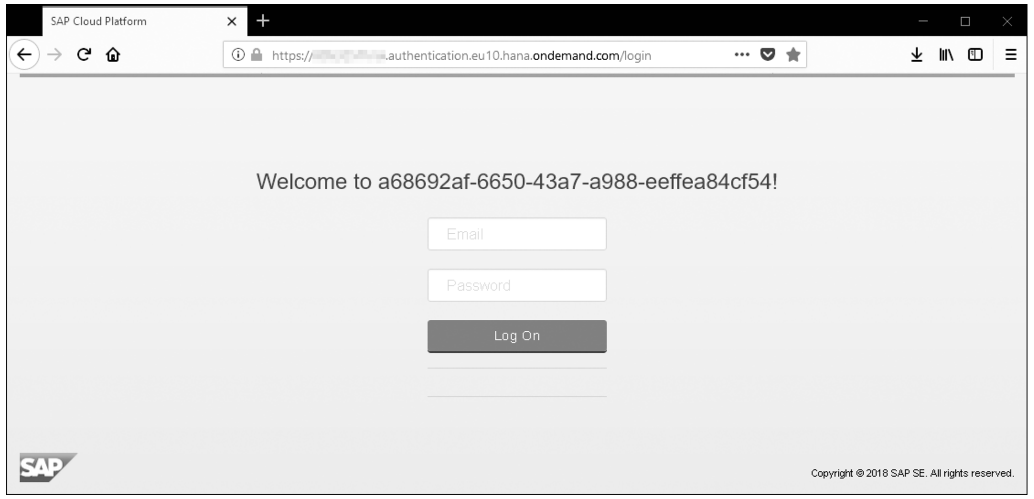
If you successfully went through the previous two sections, you should have in your file or version control system a structure that looks like the following:

```
.
├── approuter
│   ├── node_modules
│   ├── package.json
│   └── xs-app.json
├── manifest.yml
└── xs-security.json
```

Note that you may have different content in the *approuter* folder depending on your individual setup. Ensure that xs-app.json, package.json, manifest.yml, and the *node_modules* directory with correct contents are present at least. If your structure is sufficient, you're ready for the first test. Deploy the AppRouter using the following commands:

```
cd <destLocation>
cf api https://api.cf.eu10.hana.ondemand.com
cf login
cf push
```

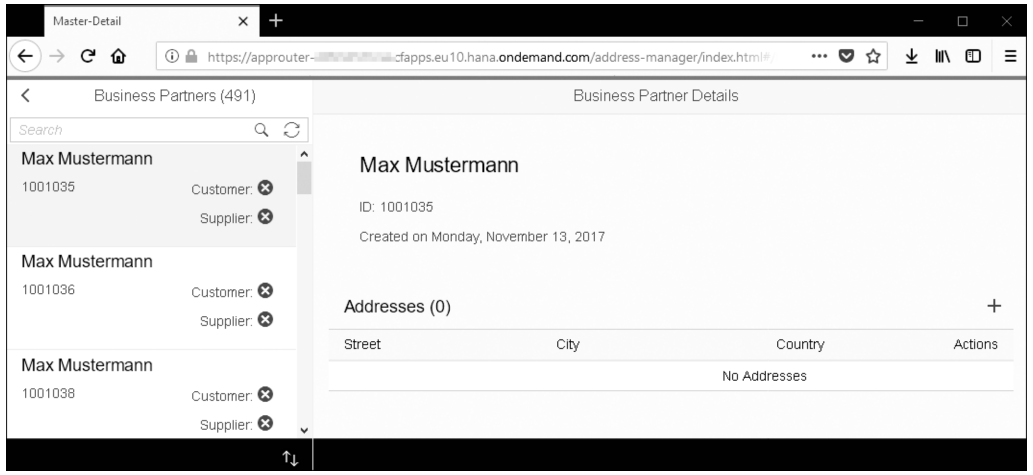
Now, you should be able to locate the AppRouter from within your browser using the host name of your deployment as well as the subaccount's subdomain following the pattern as specified earlier. For example, using our example user P1942765239, the final URL would be *https://approuter-p1942765239trial.cfapps.eu10.hana.ondemand.com/address-manager/*. This address should present a page similar to Figure 5.4 where you'll use your email address and password to log in.


SAP Cloud Platform
https://[illegible].authentication.eu10.hana.ondemand.com/login
Email
Password
Log On
SAP
Copyright © 2018 SAP SE. All rights reserved.

Welcome to a68692af-6650-43a7-a988-eeffea84cf54!

Figure 5.4 Login Screen of Our Application Based on Application Router and XSUAA

After logging in, you should also see our Business Partner Address Manager application, as shown in Figure 5.5, now being served via the AppRouter as you can see from the URL.



Master-Detail

https://approuter-[redacted]cfapps.eu10.hana.ondemand.com/address-manager/index.html#

Business Partners (491)

Search

Max Mustermann
1001035 Customer: ⊗
Supplier: ⊗

Max Mustermann
1001036 Customer: ⊗
Supplier: ⊗

Max Mustermann
1001038 Customer: ⊗
Supplier: ⊗

Business Partner Details

Max Mustermann

ID: 1001035

Created on Monday, November 13, 2017

Addresses (0)

Street	City	Country	Actions
No Addresses			

Figure 5.5 Business Partner Address Manager Application Delivered over Application Router after Authentication

Did You Know? Running the Application Router Locally

Sometimes it might be helpful to run the AppRouter locally for testing or debugging purposes, in particular, when you want to experiment with different settings and do not want to deploy the AppRouter to SAP Cloud Platform all the time. To run the AppRouter locally, place two required files called default-env.json (which contains the required environment variables) and default-services.json (which contains the XSUAA service binding information) next to xs-app.json. Check out the GitHub repository for this book for a concrete example. You may run the AppRouter locally using `npm start`.

5.2.4 Protecting Our Backend Microservice

Although authentication works with the AppRouter, our Java-based backend microservice is still fully visible in the web and not protected. We'll therefore need to protect our backend microservice as well so that it only accepts requests with valid and trusted JWTs. Two simple steps are all that's required.

First, we'll need to modify our Java backend's web.xml and introduce the lines of code shown in Listing 5.5.

```
<login-config>
    <auth-method>XSUAA</auth-method>
</login-config>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Read business partners ↩
        </web-resource-name>
    <url-pattern>/api/business-partners</url-pattern>
  </web-resource-collection>
    <auth-constraint>
        <role-name>ViewAddresses</role-name>
    </auth-constraint>
</security-constraint>
<security-role>
    <role-name>ViewAddresses</role-name>
</security-role>
```

Listing 5.5 Declarative Security Configuration for Backend Microservices

This code essentially leverages the SAP Java buildpack to validate incoming requests and only accept requests with a valid JWT for users who possess the `ViewAddresses` role (which we'll introduce in the next section).

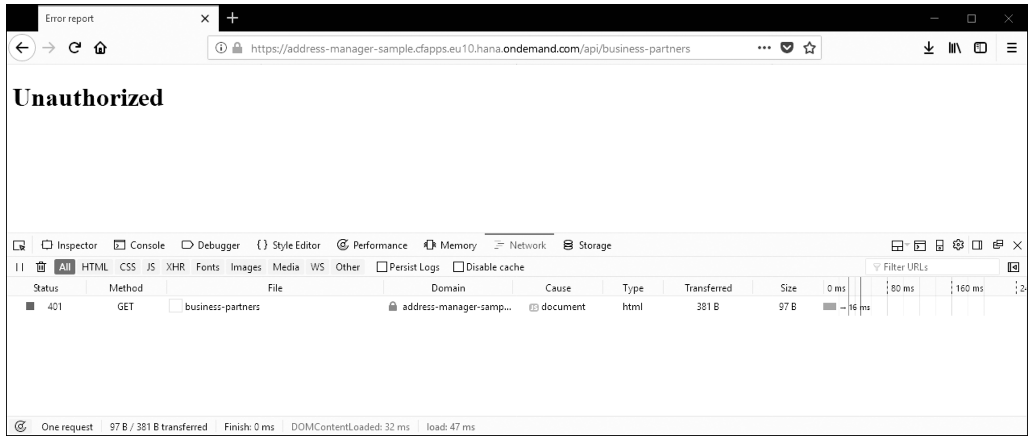
Second, we'll also need to bind our Java backend service to our previously created XSUAA instance `my-xsuaa`. Therefore, we'll slightly modify the manifest.yml definition of our Java backend and introduce the `my-xsuaa` service instance to the services section:

```
services:
- my-xsuaa
```

Now, we're ready to rebuild and redeploy our backend microservice with the commands:

```
mvn clean install
cf push
```

As a result of this change, accessing the backend service directly will reject any requests and show a 401 HTTP status (Unauthorized) because we are not providing any valid JWT (Figure 5.6).



Error report

https://address-manager-sample.cfapps.eu10.hana.ondemand.com/api/business-partners

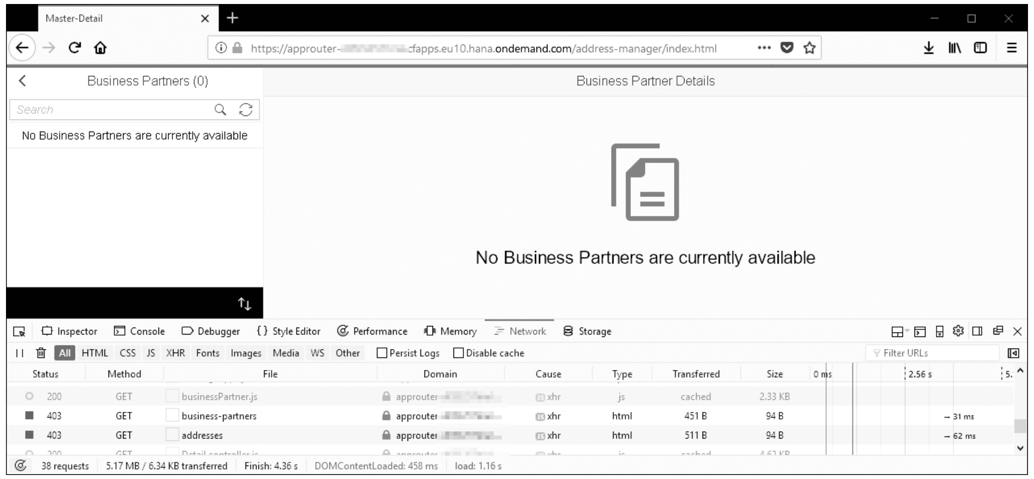
Unauthorized

Status	Method	File	Domain	Cause	Type	Transferred	Size
401	GET	business-partners	address-manager-samp...	document	html	381 B	97 B

One request | 97 B / 381 B transferred | Finish: 0 ms | DOMContentLoaded: 32 ms | load: 47 ms

Figure 5.6 Java Microservice Rejecting Unauthenticated Traffic

When using the AppRouter as the entry point, the backend service will now respond with a 403 HTTP status (Forbidden) as authentication has worked, but now, the user lacks the `ViewAddresses` authorization required to protect the endpoints accordingly (Figure 5.7). Giving the user access to the roles will be the topic of the next section.



Master-Detail

https://approuter-[redacted]cfapps.eu10.hana.ondemand.com/address-manager/index.html

Business Partners (0)

No Business Partners are currently available

Business Partner Details

No Business Partners are currently available

Status	Method	File	Domain	Cause	Type	Transferred	Size
200	GET	businessPartner.js	approuter[redacted]	xhr	js	cached	2.33 KB
403	GET	business-partners	approuter[redacted]	xhr	html	451 B	94 B
403	GET	addresses	approuter[redacted]	xhr	html	511 B	94 B

38 requests | 5.17 MB / 6.34 KB transferred | Finish: 4.36 s | DOMContentLoaded: 458 ms | load: 1.16 s

Figure 5.7 Backend Requests Authenticated over the AppRouter but Forbidden Due to Missing Authorizations

Did You Know? Providing Logout Functionality

While experimenting with different authentication and authorization options, you may require logging out every now and then to explicitly invalidate established session information on the server- and client-sides. This forced logout can be an important capability for your productive application later on.

Fortunately, the AppRouter also provides a mechanism for client-initiated logout that provides a central logout facility in our distributed system. By adding the snippet in Listing 5.6 to xs-app.json, we can tell the AppRouter under which relative URL we want to expose the logout (for example, */logout*):

```
"routes": [{<route_definitions>}],
"logout" : {
  "logoutEndpoint": "/logout",
  "logoutPage": "/logout.html"
}
```

Listing 5.6 Adding Logout Functionality to AppRouter

Troubleshooting JSON Web Tokens

In some cases, you might need to investigate the content of the JWT to identify any potential issues. The JWT is passed using an HTTP header with key `Authentication`, and the value always starts with `Bearer:` followed by a Base64-encoded string. You can, for example, create an arbitrary servlet for debugging purposes and log the header to the console. Afterwards, you may use a page like *https://jwt.io* to decode the token. However, *never* do this with any productive tokens and use local solutions instead, such as the publically available *jwt-cli* tool.

5.3 Configuring Authorization Using OAuth2

Now that we've secured our application against unauthenticated traffic, let's now turn to how to check concrete business authorizations as part of the application semantics.

5.3.1 Overview

In the previous section, we provided the setup required to authenticate our users end-to-end using the AppRouter and our previously built Business Partner Address

Manager application. However, we haven't yet considered any business authorizations, the more fine-grained controls over our exposed business capabilities. For example, let's say we only want to allow certain users *read* rights on business partner addresses while other users might be allowed to also *write* new addresses. For this distinction, we'll require authorization definitions at design-time and corresponding runtime checks, which we'll introduce in this section. Figure 5.8 provides a high-level overview on how authorization concepts are modelled and handled in the Cloud Foundry environment on SAP Cloud Platform.

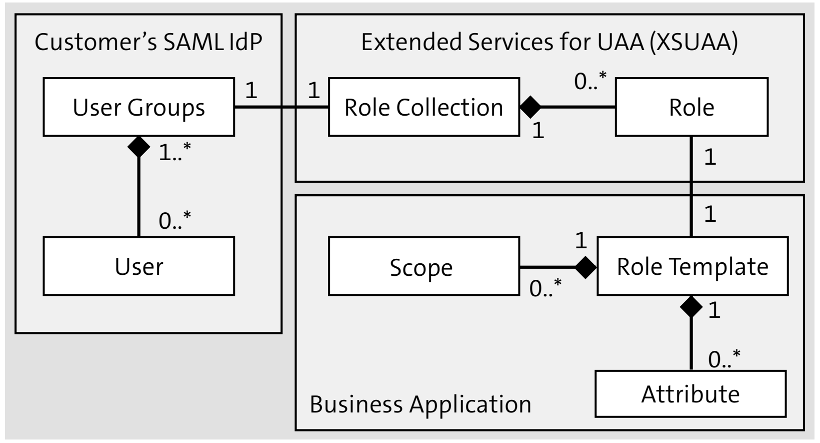


Figure 5.8 Relationships of Authorization Concepts on SAP Cloud Platform

We, as the developers or architects of our business application, can define *role templates*, which may contain multiple (OAuth) scopes. OAuth scopes refer to specific authorizations such as DISPLAY or WRITE permissions, which are checked by the microservice. We'll provide our scope and role template design using the xs-security.json descriptor (Section 5.2.2) when creating the XSUAA service instance.

The consumer of our application (for example, an administrator in a client company) can then instantiate the provided role templates into concrete roles in his subscriber accounts (for example, giving them customer-specific names) and may aggregate multiple roles (for example, from different providers) into role collections that can finally be assigned to individual users or groups of users. In this way, we can achieve, on one hand, fine, granular authorization control for microservices and, on the other hand, compose authorization in a flexible way with more coarse role collections. The idea behind this process is that, for example, a "Business Partner Manager" role collection may span multiple applications while its underlying microservices all have

individual scopes. The role collections can resolve all associated roles and return a union of all associated scopes as part of the JWT issued by the XSUAA.

5.3.2 Defining Scopes and Role Templates

For our Business Partner Address Manager application, let's say we want to define two role templates: the `BusinessPartnerViewer`, which is only allowed to view existing addresses (scope: `ViewAddresses`), and a `BusinessPartnerManager` template, which is also allowed to write new addresses to the system (scope: `ViewAddresses` and `WriteAddresses`). To define these role templates, we'll enhance our existing xs-security.json descriptor as shown in Listing 5.7.

```
{
  "xsappname": "address-manager-<uniqueId>",
  "tenant-mode": "shared",
  "scopes": [{
    "name": "$XSAPPNAME.ViewAddresses",
    "description": "Scope to view business addresses"
  },{
    "name": "$XSAPPNAME.WriteAddresses",
    "description": "Scope to write business addresses"
  }],
  "role-templates": [{
    "name": "BusinessPartnerViewer",
    "description": "Role to view business addresses",
      "scope-references" : [
        "$XSAPPNAME.ViewAddresses"
      ]
    },{
      "name": "BusinessPartnerManager",
      "description": "Role to manage all business addresses",
      "scope-references" : [
        "$XSAPPNAME.ViewAddresses",
        "$XSAPPNAME.WriteAddresses"
      ]
    }
  ]
}
```

Listing 5.7 Enhancing the Application Security Descriptor with Scopes and Role Templates

Note that all scopes must be prefixed with "$XSAPPNAME," which will be replaced during runtime with the actual name of the application.

Finally, we'll need to update our existing XSUAA service instance `my-xsuaa` and make it aware of our design changes using the following command:

```
cf update-service my-xsuaa -c xs-security.json
```

5.3.3 Protecting Our Application

Now that we've introduced scopes and role templates to the XSUAA service instance, we'll need to protect our application accordingly. Basically, two places exist where you can check authorizations: inside the AppRouter and in the backend microservices.

Checking Scopes inside the AppRouter

First, let's check OAuth scopes inside the AppRouter and protect certain routes with scopes. For example, we could check on our */api* route that we only allow users with the `ViewAddresses` scope to pass; otherwise, the AppRouter should reject the request. However, we won't pursue this option in our example further because these checks are declarative in nature on routes only and might be used as a second line of defense. Therefore, we'll omit this option in the remaining chapter for the sake of simplicity.

```
"routes": [{
  "source": "^/api/(.*)",
  "scopes": ["$XSAPPNAME.ViewAddresses"]
}
```

Checking Scopes Declaratively in Backend Microservices

Checks can also be performed in the backend microservices where the actual business logic resides. We always recommended introducing authorization checks in your backend microservices so that the check is closely related to your business logic and semantics. In some cases, you might even need to check authorizations programmatically inside the business logic, which can be only done inside the microservice itself. To check scopes in our Java-based microservice, we can either use the declarative or programmatic approach.

The declarative approach works either by using the web.xml as before or by using annotations directly in the servlet (Servlet specification 3.0 or later). For servlets, we can rely on the SAP-provided buildpack for Java and let it work out of the box.

With the web.xml approach inside the */application/webapp/WEB-INF/* folder, you can protect resources declaratively outside your code. For example, in the last section, we showed you how to protect the URL pattern under */api/business-partners* with the `ViewAddresses` role, which corresponds exactly to the scope we just defined. In this process, you can add different URL patterns with different authorization constraints to protect your endpoints accordingly, as shown in Listing 5.8. (Please refer to the official Servlet 3.0+ specification for more details.)

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Read business partners ↩
        </web-resource-name>
    <url-pattern>/api/business-partners</url-pattern>
  </web-resource-collection>
  <auth-constraint>
      <role-name>ViewAddresses</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>ViewAddresses</role-name>
</security-role>
```

Listing 5.8 Protecting Java Microservices Declaratively

Alternatively, with the annotation-based approach, you can directly annotate the relevant servlet using the `@ServletSecurity` annotation, for example, inside the `BusinessPartnerServlet` class:

```
@WebServlet("/api/business-partners")
@ServletSecurity(@HttpConstraint(rolesAllowed = ↩
    {"ViewAddresses"}))
public class BusinessPartnerServlet extends HttpServlet
```

However, this code is just a different syntactical way of expressing the same semantics. If you prefer annotations, you can safely remove the protection from web.xml.

No matter which approach you use, you'll need to rebuild and push the application using the command:

```
mvn clean install
cf push
```

Checking Scopes Programmatically in Our Backend Microservices

Authorizations can be also checked programmatically in your backend microservices by leveraging the XS2 security libraries, which are currently distributed by SAP through the SAP Service Marketplace.

> **Note**
>
> At the time of this writing, a Maven Central delivery is not available. Please check if the artifacts are available when you read this chapter; if so, you can skip the instructions in the next paragraph.

To get the latest version of the XS2 security libraries, you'll need to download them from *http://tiny.cc/xs2seclibs* and pick the latest version. At the time of this writing, the latest version is XS_JAVA_1-70001362 (19.03.2018). Please note that you'll require a support user to access the SAP Service Marketplace (see the preface for information on getting one). Once you've downloaded the library, unzip the downloaded package to a temporary directory *<tempDir>*. Afterwards, install the libraries to your local Maven repository as follows:

```
cd <tempDir>
mvn clean install
```

After you've installed the XS2 security libraries, you'll need to reference them as dependencies from *<projectDir>/application/pom.xml*, as shown in Listing 5.9. Note that the depicted dependency versions may be outdated in the code and should be replaced with the latest available versions.

```
<!-- replace with latest available versions -->
<dependency>
    <groupId>com.sap.xs2.security</groupId>
    <artifactId>security-commons</artifactId>
    <version>0.27.2</version>
</dependency>
<dependency>
    <groupId>com.sap.xs2.security</groupId>
    <artifactId>java-container-security</artifactId>
    <version>0.27.2</version>
</dependency>
<dependency>
    <groupId>com.sap.xs2.security</groupId>
    <artifactId>java-container-security-api</artifactId>
```

```
    <version>0.27.2</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
    <version>1.0.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
    <version>2.3.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.sap.security.nw.sso.linuxx86_64.opt</groupId>
    <artifactId>sapjwt.linuxx86_64</artifactId>
    <version>1.0.19</version>
</dependency>
```

Listing 5.9 Additional Dependencies for Programmatic Security Checks

Next, the scopes can be checked programmatically using the SAP S/4HANA Cloud SDK shown in Listing 5.10. Thanks to the abstractions of the SDK, the same code can be reused in both the Cloud Foundry and the Neo environment.

```
try {
  final Scope scope = new Scope("ViewAddresses");
  if(!UserAccessor.getCurrentUser().hasAuthorization(scope)) {
    response.setStatus(HttpStatus.SC_FORBIDDEN);
    response.getWriter().write("Forbidden");
    return;
  }
} catch(UserNotAuthenticatedException | UserAccessException e) {
  e.printStackTrace();
}
```

Listing 5.10 Checking Authorizations Programmatically

Finally, we'll need to rebuild and push our application to SAP Cloud Platform using the command:

```
mvn clean install
cf push
```

Did You Know? XS2 Libraries Enable the Spring Security Framework

The downloaded security libraries aren't just helpful for programmatic authorization checks, they also enable you to secure your microservices with *Spring security* as an alternative to the standard servlet security approach we've presented so far. Moreover, the XS2 libraries allow you to use any buildpack from the community instead of the SAP buildpack for Java. To enable Spring security, the archetype generated with the SAP S/4HANA Cloud SDK contains a spring-security.xml file in the *WEB-INF/* folder of your application, which you can enable by adding the `ContextLoaderListener` and `DelegatingFilterProxy` to your web.xml. The entire approach is described in more detail in a blog post at *http://tiny.cc/sapsecurityblog*.

Securing Applications in the Neo Environment on SAP Cloud Platform

All methods for protecting applications with the declarative servlet security approach as well as the programmatic approach can be equally used in the Neo environment of SAP Cloud Platform. The only difference is that, for Neo applications, the login configuration must be changed from `XSUAA` to `FORM`:

```
<login-config>
    <auth-method>FORM</auth-method>
</login-config>
```

Furthermore, you won't need to add any additional XS2 security libraries to your project.

5.3.4 Assigning Users to Application Roles

Now that we have a new XSUAA configuration in place and have protected our backend microservices, we'll need to assign the corresponding roles to our users. We can assign roles using the SAP Cloud Platform cockpit, for example, using *https://account.hanatrial.ondemand.com/cockpit*.

In the first step, open the **Role Collections** tab under the **Security** ribbon of your Cloud Foundry subaccount as shown in Figure 5.9.

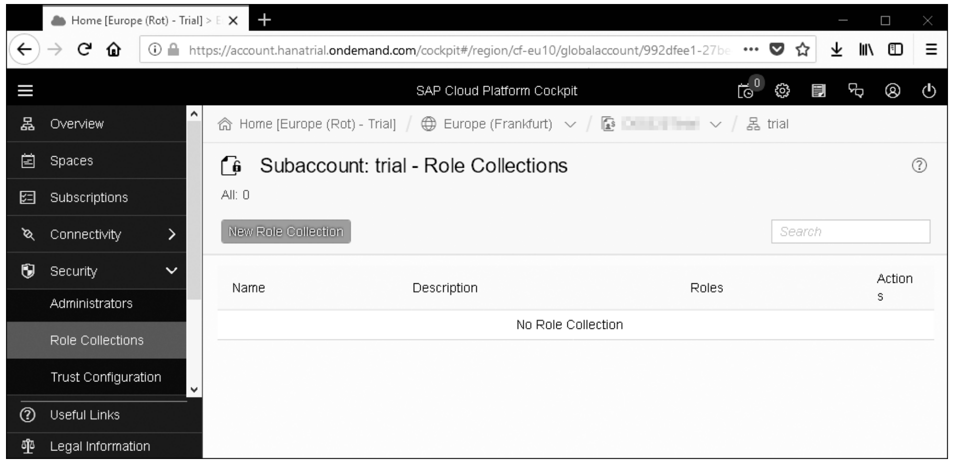
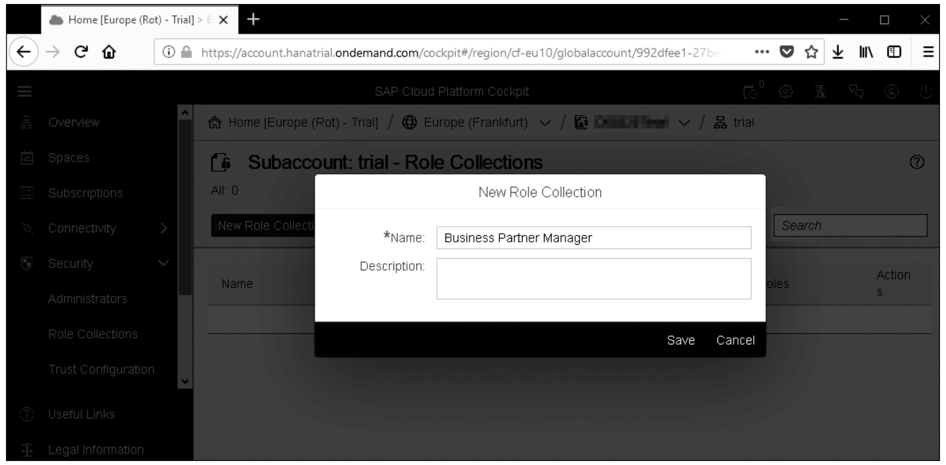

Home [Europe (Rot) - Trial]
https://account.hanatrial.ondemand.com/cockpit#/region/cf-eu10/globalaccount/992dfee1-27b
SAP Cloud Platform Cockpit
Overview
Spaces
Subscriptions
Connectivity
Security
Administrators
Role Collections
Trust Configuration
Useful Links
Legal Information
Home [Europe (Rot) - Trial] / Europe (Frankfurt) / trial
Subaccount: trial - Role Collections
All: 0
New Role Collection
Search
Name
Description
Roles
Actions
No Role Collection

Figure 5.9 The Empty Role Collections Menu to Define A New Collection

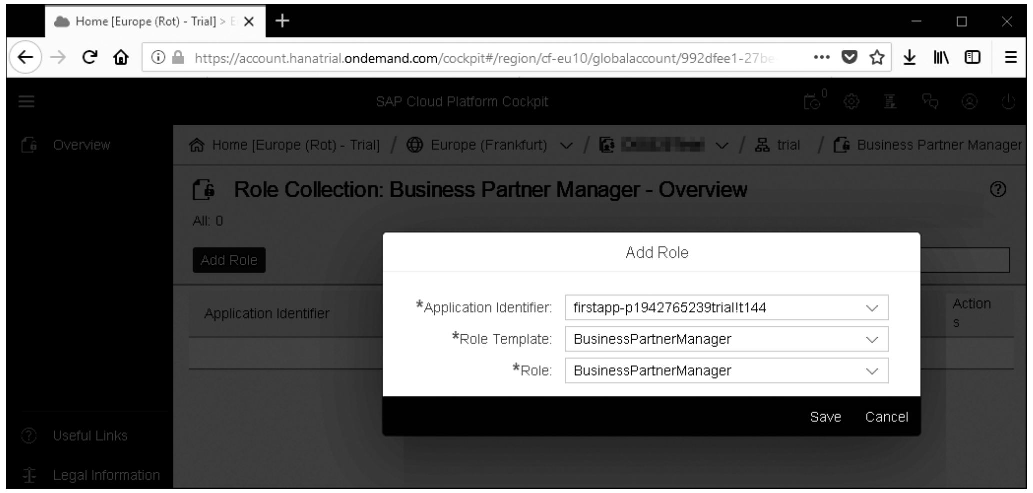
Then, we'll create a new role collection to which you can give an arbitrary name. In our example, we call our role collection "Business Partner Manager" and click **Save**, as shown in Figure 5.10.



*Name: Business Partner Manager
Description:

Figure 5.10 Defining the Business Partner Manager Role Collection

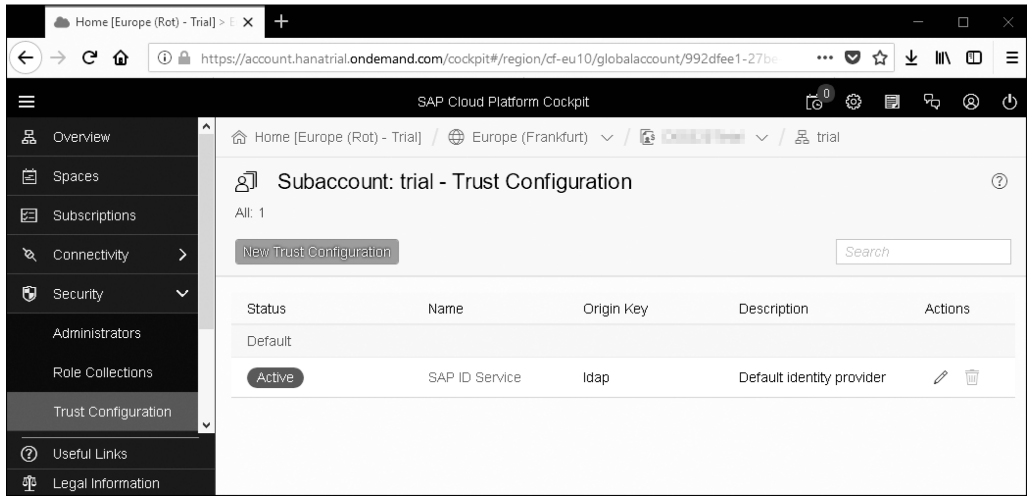
Third, select the newly created **Business Partner Manager** role collection and select **Add Role**. From the menu, select your application presented as the application identifier `xsappname` from your xs-security.json and the corresponding role template we defined earlier (Figure 5.11).


Home [Europe (Rot) - Trial]
https://account.hanatrial.ondemand.com/cockpit#/region/cf-eu10/globalaccount/992dfee1-27b
SAP Cloud Platform Cockpit
Overview
Home [Europe (Rot) - Trial] / Europe (Frankfurt) / [redacted] / trial / Business Partner Manager
Role Collection: Business Partner Manager - Overview
All: 0
Add Role
Application Identifier
Actions
Useful Links
Legal Information
Add Role
Save
Cancel

*Application Identifier: firstapp-p1942765239trial!t144
*Role Template: BusinessPartnerManager
*Role: BusinessPartnerManager

Figure 5.11 Add Role Template to Role Collection

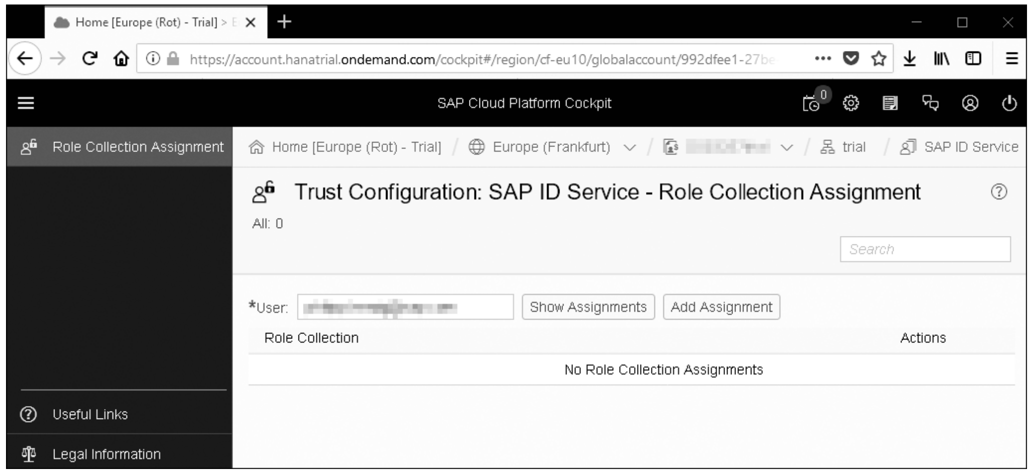
In the next step, we'll need to assign our user to the newly created **Business Partner Manager** role collection to grant the `ViewAddresses` and `WriteAddresses` scopes. Therefore, we'll select the **Trust Configuration** from the **Security** menu and select the **SAP ID Service** from the list as shown in Figure 5.12.


Home [Europe (Rot) - Trial]
https://account.hanatrial.ondemand.com/cockpit#/region/cf-eu10/globalaccount/992dfee1-27b
SAP Cloud Platform Cockpit
Overview
Spaces
Subscriptions
Connectivity
Security
Administrators
Role Collections
Trust Configuration
Useful Links
Legal Information
Home [Europe (Rot) - Trial] / Europe (Frankfurt) / [redacted] / trial
Subaccount: trial - Trust Configuration
All: 1
New Trust Configuration
Search

Status	Name	Origin Key	Description	Actions
Default				
Active	SAP ID Service	ldap	Default identity provider	

Figure 5.12 Trust Configuration in SAP Cloud Platform with the SAP ID Service as Default Identity Provider

In the opening dialog, enter your email into the user field and click **Show Assignments** (Figure 5.13).

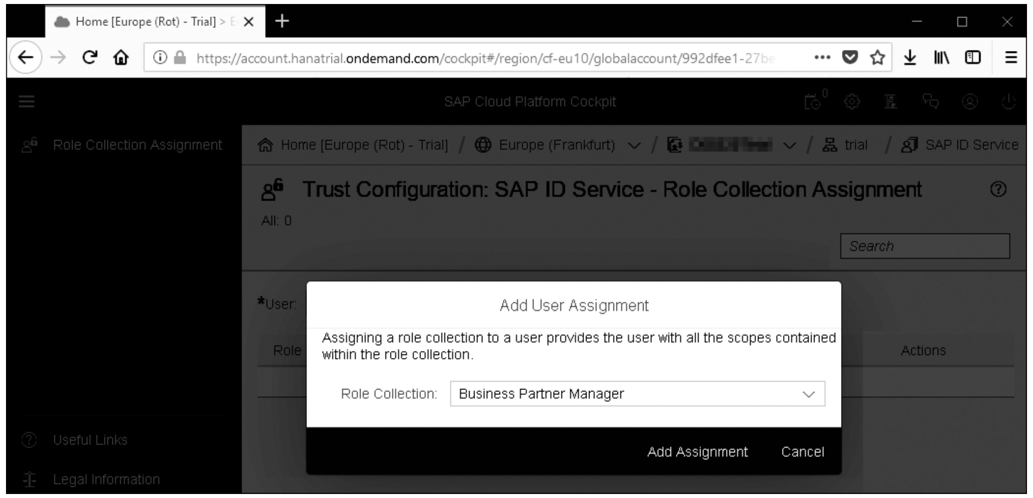

Home [Europe (Rot) - Trial]
https://account.hanatrial.ondemand.com/cockpit#/region/cf-eu10/globalaccount/992dfee1-27b
SAP Cloud Platform Cockpit
Role Collection Assignment
Home [Europe (Rot) - Trial] / Europe (Frankfurt) / [redacted] / trial / SAP ID Service
Trust Configuration: SAP ID Service - Role Collection Assignment
All: 0
Search
Show Assignments
Add Assignment
Useful Links
Legal Information

*User: [redacted]

Role Collection	Actions
No Role Collection Assignments	

Figure 5.13 Selecting User Role Assignments

Then, click **Add Assignment** and choose the **Business Partner Manager** role collection from the menu to assign it to your user (Figure 5.14).


Home [Europe (Rot) - Trial]
https://account.hanatrial.ondemand.com/cockpit#/region/cf-eu10/globalaccount/992dfee1-27b
SAP Cloud Platform Cockpit
Role Collection Assignment
Home [Europe (Rot) - Trial] / Europe (Frankfurt) / [redacted] / trial / SAP ID Service
Trust Configuration: SAP ID Service - Role Collection Assignment
All: 0
Search
*User
Role
Actions
Useful Links
Legal Information
Add User Assignment
Add Assignment
Cancel

Assigning a role collection to a user provides the user with all the scopes contained within the role collection.

Role Collection: Business Partner Manager

Figure 5.14 Assigning Role Collection "Business Partner Manager" to a User

Based on this role collection assignment, we can now use the application as before using our authenticated and authorized user while other users will no longer have access.

As noted earlier, you may need to call the AppRouter's logout functionality once to invalidate any existing sessions and retrieve a new JWT with the updated scope information.

5.4 Protecting against Common Web Application Threats

Besides protecting our application with authentication and authorization, the AppRouter, as well as the SAP S/4HANA Cloud SDK, provides additional capabilities to protect your application against standard attack vectors in web or cloud applications. We'll touch on some of these potential attacks in this final section. Note that enabling the features we describe do not mean your application is secure. Additional attack vectors are possible and require mitigation when building your application. For example, typical vulnerabilities such as SQL, Lightweight Directory Access Protocol (LDAP), or other path injections; cross-site scripting attacks; or the use of vulnerable third-party dependencies in your application code must be checked by different tools or methods. These additional aspects are beyond the scope of this book. We recommend that you consult the Open Web Application Security Project (OWASP) for more information on this topic (*https://owasp.org*).

5.4.1 Cross-Site Request Forgery

In simple terms, cross-site request forgery (CSRF) or, more specifically, *session riding* is an attack where an attacker convinces a user to follow a link that does something that the user did not intend to do (for example, deleting all records from a database if the user is the application's administrator). As a prerequisite to this attack, the user must be logged into the application, which can easily happen when session timeouts are unreasonably long, when login information has been stored by the user inside the browser for convenience, or login information is stored by the application for a long period of time in persistent cookies.

To prevent this attack, the client- and server-side code must share a secret unknown to an attacker. This secret is called the *CSRF token*.

Fortunately, in the Cloud Foundry environment of SAP Cloud Platform, the AppRouter enables CSRF protection for any HTTP method except `GET` and `HEAD` on routes

that require authentication, that is, where the `authenticationType` attribute has not been set to `none`. If a route is CSRF-protected but the client-side application does not send a valid CSRF token as part of the `x-csrf-token` header, the AppRouter will reject the request with a 403 HTTP error code. To retrieve a valid CSRF token, the client-side code must first send a request with the HTTP header key-value `x-csrf-token: fetch`.

If you need to change this default behavior, you may use the `authenticationType` or `csrfProtection` attributes on routes in your xs-app.json, as shown in Listing 5.11.

```
"routes": [{
  "source": "^/api/(.*)",
  ...
  "csrfProtection": false,
  "authenticationType": "none"
}
```

Listing 5.11 Changing CSRF Protection in the AppRouter

Additionally, every microservice created using archetypes of the S/4HANA Cloud SDK also contain a CSRF protection filter defined in the web.xml file, as shown in Listing 5.12.

```
<filter>
  <filter-name>RestCsrfPreventionFilter</filter-name>
  <filter-class>
      org.apache.catalina.filters.RestCsrfPreventionFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>RestCsrfPreventionFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listing 5.12 CSRF Protection in Java Services

If you are using the AppRouter and have protected your backend microservice as described in this chapter, you can safely remove this protection. However, if you're exposing your service directly without the AppRouter (e.g., in the Neo environment), you should keep this protection in place to protect your application from being vulnerable to this attack.

5.4.2 Clickjacking

Clickjacking refers to an attack vector where the attacker embeds the vulnerable application, for example, using an HTML iframe, and overrides the visible intended actions with unintended actions. For instance, the user may intentionally click the playback button in a video application but instead unintentionally triggers a purchase in some retail application.

To prevent this attack, recent browsers can be instructed to block remote content embedded with iframes using the `X-FRAME-OPTIONS` HTTP header.

If you're using the AppRouter, the default setting of `X-FRAME-OPTIONS` is `SAMEORIGIN`, that is, the browser allows iframes only if the framed content originates from the same domain as the embedding application. You can either turn off this behavior by setting the AppRouter's environment variable `SEND_XFRAMEOPTIONS` to `false`, or you can override the value by specifying the `httpHeaders` environment variable (for example, setting it to `DENY` to completely forbid the use of iframes).

In backend microservices generated using the SAP S/4HANA Cloud SDK, the lines from web.xml, shown in Listing 5.13, introduce the same behavior.

```
<filter>
  <filter-name>HttpSecurityHeadersFilter</filter-name>
  <filter-class>
    com.sap.cloud.sdk.cloudplatform.security.servlet ↩
        .HttpSecurityHeadersFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>HttpSecurityHeadersFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listing 5.13 HTTP Security Headers in Java Services

You can remove these lines if your microservices are not directly accessible, for example, using the AppRouter. However, when your service is directly accessible, for example, when exposing Java-based services directly in the Neo environment of SAP Cloud Platform, you should keep this filtering setting in place.

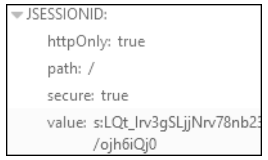
In addition, you may use the Content Security Policy (CSP) header, which is currently recommended by the W3C (*https://www.w3.org/TR/CSP2/*) and supported by all modern browsers. We won't discuss this header further in this book, but recommend that

you look into it yourself and use the standard HTTP header settings introduced in this chapter, since such policies are specific to each application.

5.4.3 Securing Session Cookies

In all SAP Cloud Platform environments, session IDs stored in cookies are used on the client to continuously identify a user across several stateless server interactions without requiring re-logins. Thus, these cookies are a primary target for attackers because they grant access to applications using the identities of compromised users. Therefore, sessions and their cookies must be protected in productive environments.

Fortunately, in the Cloud Foundry environment of SAP Cloud Platform, the AppRouter uses secure attributes by default when instructing browsers to store session cookies by specifying the `httpOnly` and `secure` options in the `Set-Cookie` response header, as shown in Figure 5.15. This process happens in the redirection phase (Step ❺ in Figure 5.2).



```
JSESSIONID:
    httpOnly: true
    path: /
    secure: true
    value: s:LQt_lrv3gSLjjNrv78nb23
           /ojh6iQj0
```

Figure 5.15 Example Set-Cookie Instruction Returned from AppRouter after Successful Login

The `secure` option instructs the browser to only send the session cookie over encrypted connections (that is, HTTPS). The `httpOnly` options instructs the browser that the cookie can be only accessed on real HTTP requests, for example, any JavaScript access is forbidden.

For testing purposes without encrypted connections, you might need to turn off this option by setting the AppRouter's environment variable `SECURE_SESSION_COOKIE` to `false`.

If you are using the Neo environment on the SAP Cloud Platform, no default security option exists. Thus, we recommend that you add the code shown in Listing 5.14 to your application's web.xml file.

```
<session-config>
  <cookie-config>
    <secure>true</secure>
    <http-only>true</http-only>
  </cookie-config>
</session-config>
```

Listing 5.14 Securing Session Cookies on SAP Cloud Platform Neo

5.4.4 Secure HTTP Headers

Finally, the backend microservices generated with the SAP S/4HANA Cloud SDK include `HttpSecurityHeadersFilter` in web.xml, which sets default headers for security. Besides the `X-FRAME-OPTIONS` headers we mentioned earlier in Section 5.4.2, `HttpSecurityHeadersFilter` also enforces the `Strict-Transport-Security` header, which instructs the browser to only allow cloud application access via the HTTPS protocol. In other words, no unencrypted HTTP traffic without TLS is allowed.

5.5 Summary

This concludes our chapter on how to protect a microservice-based application on SAP Cloud Platform. Based on these lessons, we'll turn, in the next chapter, to how to safely integrate our example application with SAP S/4HANA.

Contents

Index