OULU UNIVERSITY OF
APPLIED SCIENCES

Tobias Mühlbauer

Resource Cockpit Development

Resource Cockpit Development

Tobias Mühlbauer
Bachelor Thesis
Spring 2023
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author: Tobias Mühlbauer
Title of the bachelor's thesis: Resource Cockpit Development
Supervisor(s): Lasse Haverinen
Term and year of completion: Spring 2023 Number of pages: 52 + 4 appendices

---

The goal of this thesis was to develop a web application called *Resource Cockpit* which allows users to monitor their SAP cloud resources for the management of business processes especially integration flows. Information about the workload and availability of these resources is needed to make business decisions and detect problems in workflows.
This thesis is the first part of a tool packet which will be made available to customers of SAP as part of their ERP solutions.

The front end and GUI of the application were developed with the SAPUI5 framework, which uses a combination of JavaScript with XML views and was guided by user tests. The backend functionality was developed in the Java Spring framework. A particular focus of this thesis was on the implementation of unit tests to ensure reliability during further development. The unit tests for the backend Java modules rely on the built-in framework support of Spring and the dependency injection feature of the framework. The unit tests of JavaScript make use of the QUnit framework. A mock data provider utilizing an in-memory H2 database was developed during this thesis work to improve the accuracy of the tests. Real customer data was analysed to generate mock data resembling reality.

The *Resource Cockpit* was successfully implemented and equipped with an extensive testing framework to support further development.

Keywords: Front-end development, Integration Flow, Java Spring framework, Resource Monitoring, SAPUI5 framework, Unit testing

PREFACE

This Bachelor Thesis was developed in conjunction with my company-oriented projects at SAP / Walldorf / Germany.

”SAP is one of the world’s leading producers of software for the management of business processes, with more than 105,000 employees worldwide, developing solutions that facilitate effective data processing and information flow across organizations.” [1]
“As the market leader in enterprise application software, SAP is supporting companies of all sizes and in all industries run better by redefining enterprise-resource-planning (ERP) and creating networks of intelligent enterprises that provide transparency, resiliency, and sustainability across supply chain.” [2]
An important aspect of this are the cloud services and cloud-based applications provided by SAP for their customers.
”Today SAP has more than 230 million cloud users, more than 100 solutions covering all business functions, and the largest cloud portfolio of any provider.” [1]

As cloud resources become more important, there is also a need to monitor these resources for problems and utilization.

During my time with SAP, I primarily worked on developing an ERP application called *Resource Cockpit* which the users can use to monitor their own SAP resources including cloud resources. My focus during the development was on monitoring automated workflows called *integration flows*. The development of this application included the GUI development, the implementation of functionality and also a focus on usability with user testing.

A major part of the development was spent on the implementation of automated tests which ensure the functionality of the *Resource Cockpit* before deployment to the customers.

Heidelberg, 11.03.2023
Tobias Mühlbauer

CONTENTS

LIST OF FIGURES

GLOSSARY

BPMN	Business Process Model and Notation
EP	Enterprise Portal
ERP	Enterprise-Resource-Planning
ES Repository	Enterprise Services Repository
FICO	Finance and Controlling
Fiori	design Guidelines set by SAP for the development of web applications
H2	relational database; can be embedded directly in Java applications
HCM	Human Capital Management
HR	Human Resources
Integration Flow	automated workflow used to synchronize data between multiple applications or services
MM	Material Management
MVC	Model View Controller; part of SAPUI5 framework
PaaS	Platform as a Service
PI	Process Integration
OpenUI5	open-source version of the SAPUI5 framework; contains the core functionality and most used libraries
QUnit	JavaScript framework for unit tests

SAP	initially called System Analysis Program Development; a multinational enterprise headquartered in Walldorf, Germany
SAP CRM	SAP Customer Relation Management
SAP SRM	SAP Supplier Relationship Management
SAPUI5	JavaScript framework for UI development used by SAP; internal version of the OpenUI5 framework
SD	Sales and Distribution
SOA	Service-oriented architecture
UI	user Interface designed with SAPUI5 framework
XML	Extensible Markup Language as defined in SAPUI5

1 INTRODUCTION

1.1 Goal

The goal of this thesis is to provide customers with a user-friendly interface which can be used to monitor their resources such as database utilization or network traffic. The primary focus of this thesis is on monitoring integration flows. With this information customers can view and understand their database and network activities such as checking if the number of calls on the database is too high. This would lead to a blockage of requests or failed requests.

1.2 Motivation

The stated goal is realised by a web application called *Resource Cockpit* which allows customers to easily understand the status of their resources. There are several requirements which the *Resource Cockpit* must fulfil:

- It must be user friendly and intuitive to use.
- There must be a visual representation of the information.
- Problems must be noticeable at a glance.
- The source of the problem must be highlighted.
- As an SAP application the *Resource Cockpit* must follow the SAP design guidelines and has to be similar to other SAP applications in looks and use.

1.3 Design of the Resource Cockpit

In order to reach my goal, I used tools and frameworks which are SAP standard. The UI is written in the SAPUI5 framework which uses a combination of JavaScript and XML. The functional part of the application is written in the Java Spring framework which comes with built-in unit testing.

1.4 Structure of this thesis

The thesis starts with a Requirement Analysis chapter on which basis the tasks for this thesis were defined. The requirements are split between functional requirements and quality assurance, which includes testing.

The Fundamentals chapter contains an explanation of the design guidelines used to integrate the *Resource Cockpit* into SAP enterprise resource planning and introduces process integration and resulting integration flows as a part of SAP enterprise resource planning. A general understanding of integration flows helps to understand where the *Resource Cockpit* will be located in the overall SAP landscape.

In the Frameworks chapter the most important frameworks for the development of the *Resource Cockpit* SAPUI5 and Java Spring are introduced. Key features of these frameworks which proved valuable during the implementation are highlighted with examples.

The Design chapter gives a broad understanding of the software design ideas behind the *Resource Cockpit* and explains the difference between the production version of the application which the customers will receive and the local version which is used during development and for automated testing.

In the Implementation chapter the implementation of the local application with mock data generation and the implementation of the UI is explained in more detail since these are important results of this thesis.

The Testing and Evaluation chapter explains the different methods used to test the *Resource Cockpit* and ensures the SAP quality standards. This includes unit testing which relies on the local testing environment developed as part of this thesis as well as user tests.

The results a critical view on my own work and a future outlook of this thesis are summarised in the final Results chapter.

2 REQUIREMENT ANALYSIS

In order to enhance the existing options for SAP users to monitor their resources a critical analysis was performed on the current customer access to resources and a list of requirements was gathered and defined. This process preceded the point in time when I joined the project.

2.1 Definition of the functionality of the *Resource Cockpit* application

As customer satisfaction and the highest level of quality are core goals of SAP and every company the development team defined the following requirements for the development of the *Resource Cockpit* application.

- Easy access, including convenient options for filtering and viewing detail information.
- User friendly and self-explanatory usage of the application.
- Visualization of the resource usage in a clear and concise way. Especially it should be easy to spot problems and navigate to desired details.
- The application meets the design guidelines set by SAP.

2.2 Quality testing during development

During the development of the *Resource Cockpit* quality criteria such as usability, functionality and performance were checked in regular meetings and discussions with colleagues. Towards the end of the development a survey (see Appendix 1) was used to formalize this quality evaluation. This survey focuses on the criteria that were previously defined in the requirement analysis and serves as an indicator if the goals of the *Resource Cockpit* are reached.

The tool used for the survey is a freeware tool called Survio [3]. The detailed results of the survey are analysed and presented in the Results Chapter.

The survey was conducted in German as all team members are native German speakers. (see Appendix 1 for the complete survey). For reasons of common understanding, I include an English translation here:

Questions in the survey and *their translation*:

1. Das UI ist intuitive bedienbar.
   The UI can be used intuitively.
2. Die Aufteilung des Resource Cockpit ist übersichtlich.
   The layout of the Resource Cockpit is clear.
3. Die Filter sind zielführend.
   The filters are effective.
4. Die gewünschten Daten sind leicht einsehbar.
   The data you want is easy to see.
5. Gibt es eine Information die zusätzlich angezeigt werden sollte?
   Is there any additional information that should be displayed?
6. Die Graphen stellen die gesuchte Information gut dar.
   The graphs represent the sought information well.
7. Falls die Graphen nicht gut sind, welcher Graphentyp ist deiner Meinung nach besser?
   If the graphs are not good, which graph type do you think is better?
8. Die Ladezeiten der Daten sind angemessen. (Responsiveness)
   The loading times of the data are reasonable. (responsiveness)
9. Die gewünschten Funktionalitäten sind enthalten.
   The desired functionalities are included.

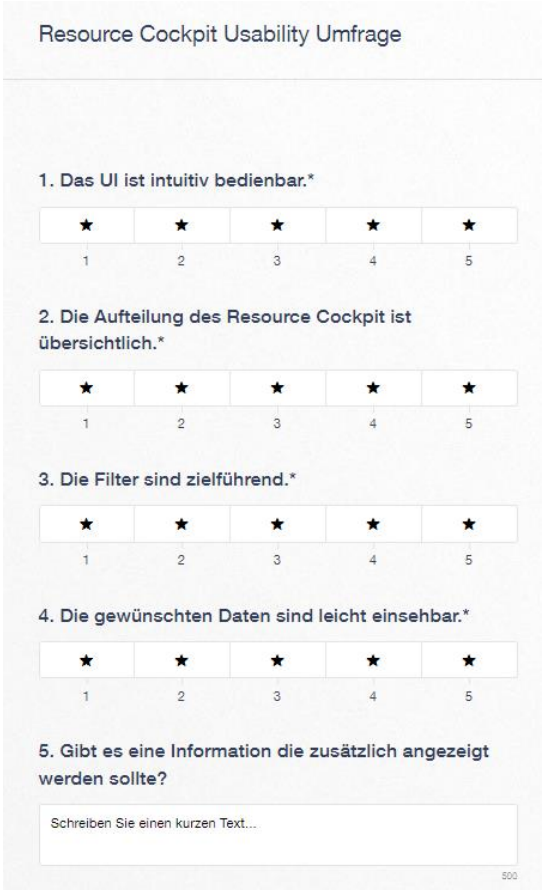


FIGURE 2-1 Excerpt of the Resource Cockpit quality survey

10. Gibt es ein Feature, das dir fehlt?

Is there a feature you are missing?

3 FUNDAMENTALS

3.1 Design guidelines

Currently SAP is building several applications for their business suite with a new and more modern UI design.

The design guidelines for these applications are outlined in the so-called SAP Fiori Design Guidelines [4], which sets the standard for enterprise user experience by removing unnecessary complexity.

The goal is to put users in control of their business tasks by displaying only what they really need and want to see. This core goal is reflected in five design principles: [4]

- Role-based
- Adaptive
- Coherent
- Simple
- Delightful

For example, the “coherent” principle applied on the Resource Cockpit means that the control elements of the application need to behave and look the same as they do in other SAP applications with similar functionality (see figure 3-1).

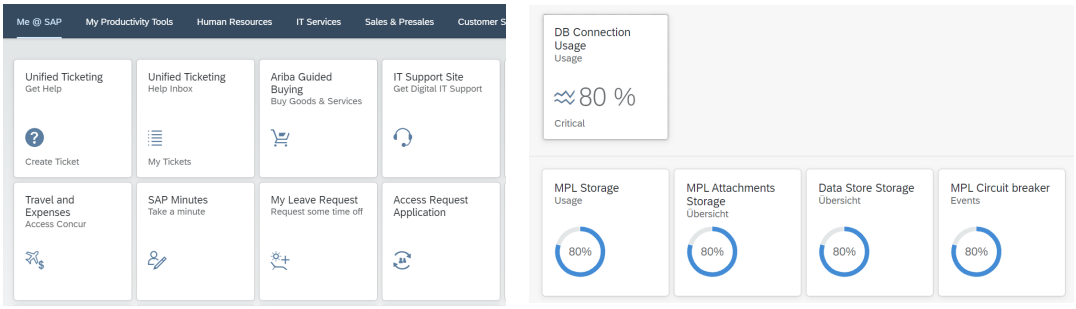


FIGURE 3-1 Example of a SAP HR application (left) and a similar design in the Resource Cockpit (right) – see appendix 2 for full size view

Since the *Resource Cockpit* application will be part of this SAP landscape these design guidelines were applied to the *Resource Cockpit*.

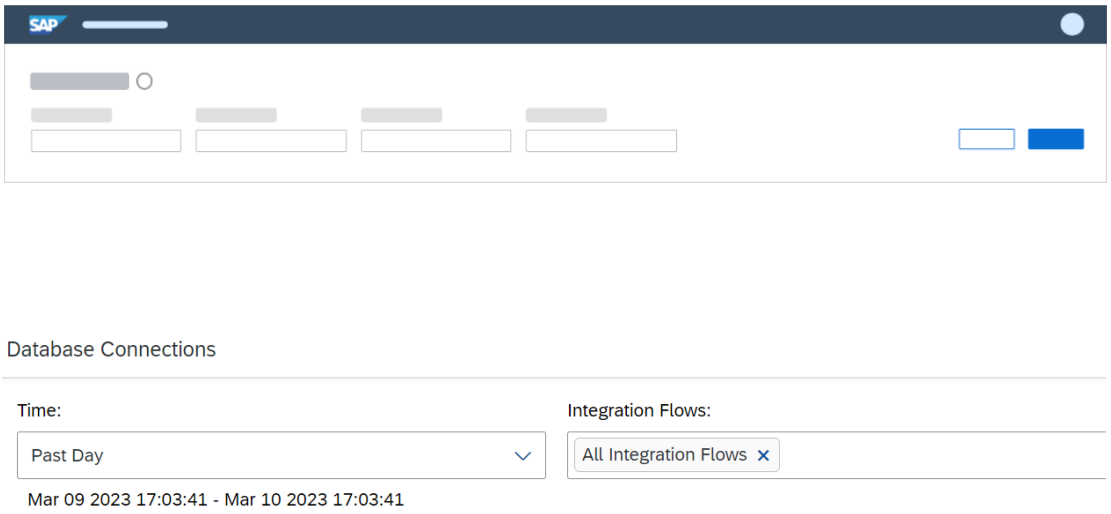


FIGURE 3-2 Design standard set by Fiori (top) and implementation in the Resource Cockpit (bottom)

Since these design guidelines functioned as a starting point (see figure 3-2), the main focus was on considering which type of graph is most suitable to display each respective type of data and how to make sure the user is not overloaded or confused with unnecessary information.

A combination of simple graphs with focused information and a colour scheme to indicate problems turned out to be a good compromise between giving the user all the relevant information without overloading the UI (figure 3-3).

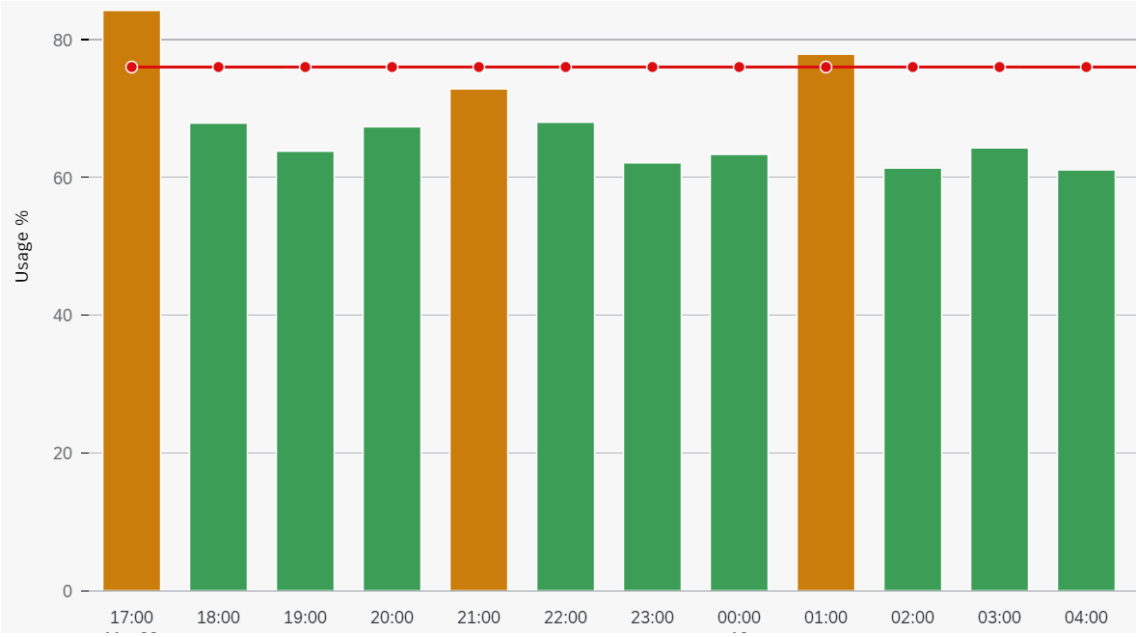


FIGURE 3-3 A graph combining information with problem indicator

Details on the implementation of this are presented in the Implementation chapter.

3.2 Process integration and integration flow

This chapter provides information relevant to the integration flow to help in understanding this automated workflow used to synchronize data between multiple applications or services and the SAP process integration (PI) in general.

As the thesis tasks are directly related to the integration flows between the customer and their sources of data, i.e. two or more heterogeneous applications or systems, the general process integration Landscape needs more attention and needs to be explained in more detail to understand the relevance of my work and the new *Resource Cockpit* application.

3.2.1 SAP enterprise resource planning (ERP)

All businesses, large or small, need to carry out standard business functionalities such as accounting, procurement, project management, risk management, compliance, and supply chain operations. As a solution SAP offers ERP software that organizations use to manage these day-to-day business activities. ERP helps companies manage all their business-related procedures and processes more efficiently and automate processes. [5]

FIGURE 3-4 ERP managing business-related procedures and processes [6]

As shown in figure 3-4, ERP software is an integrated set of technologies and systems in which a centralized computer system collects and holds company-wide and division-specific information, so it can be accessed and used by all the team members who need it. [6]

With an ERP System a company can automate workflows that require a high degree of manual labour and provide the resulting data in real-time across divisions which increases transparency to management and team members. [6] With the data provided by an ERP system a company can reduce costs for repetitive menial labour, detect inefficiencies in their workflows and better scale their business.

Comparing the feature list defined in the requirement analysis for this thesis task (see chapter 2.1) shows that the *Resource Cockpit* developed for this thesis is a requirement for ERP. The *Resource Cockpit* will allow users to monitor their cloud resources in real time and make information about its usage

transparent. With this information the *Resource Cockpit* will support customers when making business decision.

Since an ERP system consists of multiple integrated systems like the *Resource Cockpit* the question arises how these tools can be integrated.

3.2.2 Legacy system

While implementing the SAP ERP in a large business establishment not all sections can be brought under the SAP ERP. Many of the business sections may have their own proprietary tools which are highly complex and may not be possible to be replaced. They run parallel to the SAP System. They are called the Legacy Systems. Therefore, it becomes necessary to integrate the SAP Systems into such pre-existing non-SAP System (or vice versa). This is where the SAP PI comes into play. [5]

3.2.3 Reasons to use SAP PI

Apart from Legacy Systems, in a large business establishment, SAP ERP does not consist of a single system but several integrated systems, i.e. CRM, SRM and FICO etc. To handle such complexities SAP introduced process integration as a platform to provide a single point of integration for all systems without touching existing complex networks of legacy systems [5]. Such a setup is modelled in figure 3-5.

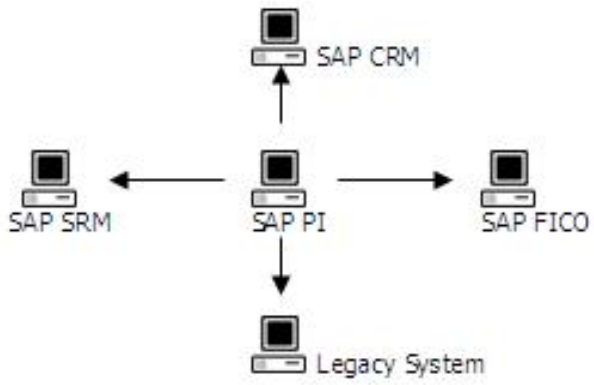


FIGURE 3-5 Example of SAP PI [5]

3.2.4 Process Integration tools

Whenever you want to achieve communication between two or more heterogeneous applications or systems in SAP ERP that exist within or across business landscapes, you work with the Eclipse based process integration Tools. [7] You can have the role of a designer or configurator at different stages of a task flow to perform activities on objects involved in process integration. For each role PI Tools offer dedicated perspectives and interfaces to interact with the objects in the ES Repository[1] and Integration Directory under one platform.

Figure 3-6 below gives an overview of the association between three important levels: User, Eclipse-based process integration Tools and the process integration landscape:

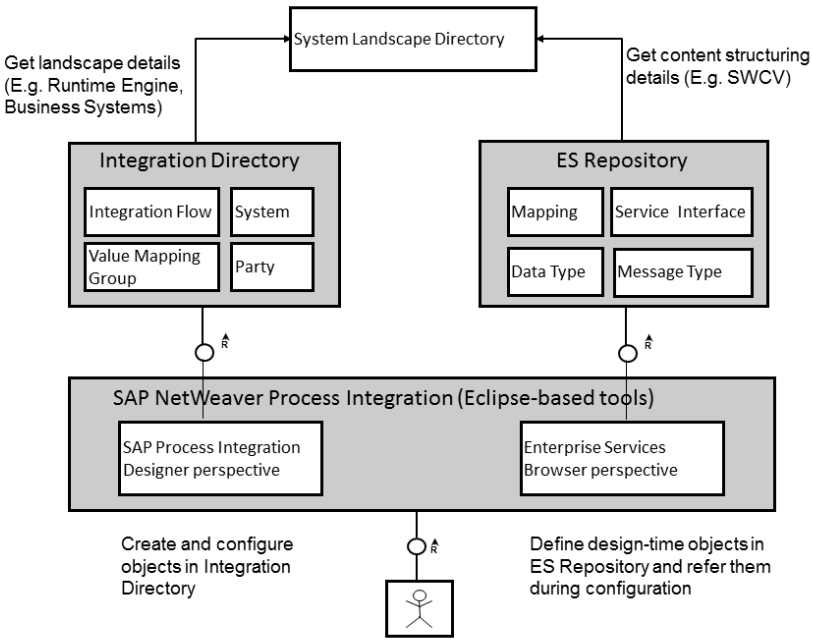


FIGURE 3-6 Association between three important levels: User, Eclipse-based process integration Tools and the process integration landscape [7]

[1] The Enterprise Services Repository in SAP is a central repository where you define, access, and manage SOA assets such as services, data types, and so on. The repository stores the definitions and metadata of enterprise services and business processes.

3.2.5 Working with and creating an integration flow model

The procedure described in figure 3-6 is used to enable a connection between two or more heterogeneous applications or systems by creating an integration flow. These systems can be from different vendors (non-SAP and SAP), of different versions, and implemented with different programming languages (Java, ABAP, etc.).

3.2.6 Elements in an integration flow

The integration flow is a BPMN[2]-based [8], graphical model.

In order to understand the behaviour of this important configuration object we need to take a closer look at the elements in an Integration Flow. Figure 3-7 below shows the graphical representation of an integration flow and its elements:

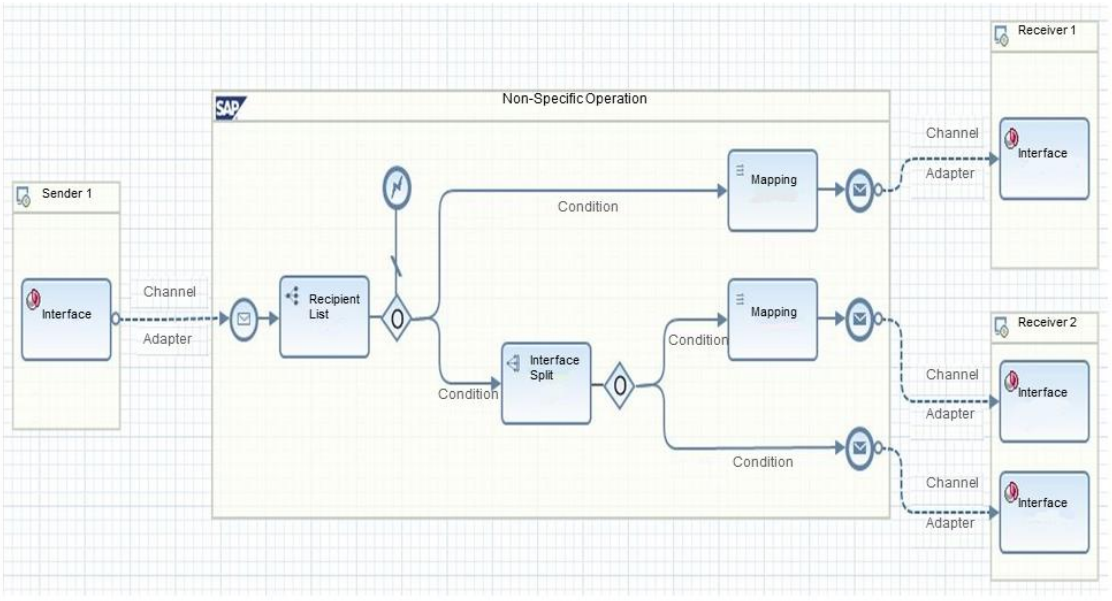


FIGURE 3-7 Elements in an integration flow [7] – see appendix 3 for full size view

[2] Business Process Model and Notation is a modelling system for business processes

In figure 3-7 above, the integration flow allows a sender system to interact with two receiver systems having different communication protocols.

This is the key purpose of integration flows, namely that it allows communicating data between multiple applications or services independent of their respective implementation and across different protocols. In addition, users are not restricted in the system they use to access their resources. In this respect it is possible to think of an integration flow as a network with additional capabilities for different protocols and systems working together.

My *Resource Cockpit* will help users to monitor such integration flows and provides aggregated and centralized information about the history of the integration flows. These flows are subject to similar issues like other networks, for example there can be problems with very high usage of an integration flow or a long allocation of time to a single process. This would block other processes and cause a high number of network failures. Such issues would cause inefficiencies and problems in a customer’s business workflows. With the *Resource Cockpit* these issues can be easily detected which helps customers optimize their business workflows.

4 FRAMEWORKS USED

The *Resource Cockpit* application is written in two frameworks: SAPUI5 for the frontend, which the users get to see, and Java Spring for the backend, which is running in the cloud. This chapter briefly introduces these frameworks and explains the idea behind important functionality used during the implementation of the *Resource Cockpit*. A precise description of the implementation will follow in the Implementation chapter.

4.1 SAPUI5 framework

SAPUI5 is SAP's new development toolkit for frontend development with HTML5 standard using JavaScript und CSS. SAPUI5 applications are optimised to run in modern browsers and on mobile devices.

SAPUI5 is the primary tool to realise the SAP Fiori Guidelines [4], which are the current standard for new SAP applications. As such, SAPUI5 includes libraries and control elements specialised for the development of Fiori applications.

SAP also provides an open-source version of this toolkit called OpenUI5, which contains the core functionality, vast majority of control elements and the most commonly used libraries.

4.1.1 Data binding in SAPUI5

Every application works with data, for example import data from a database or API, process data with internal logic or display data in the UI.

SAPUI5 follows the "Model View Controller" (MVC) design philosophy. This means that the model (data & data sources), view (UI) and controller (application logic) are always strictly separated. Data binding is used to communicate information between models and views. The following figure 4-1 illustrates this idea.

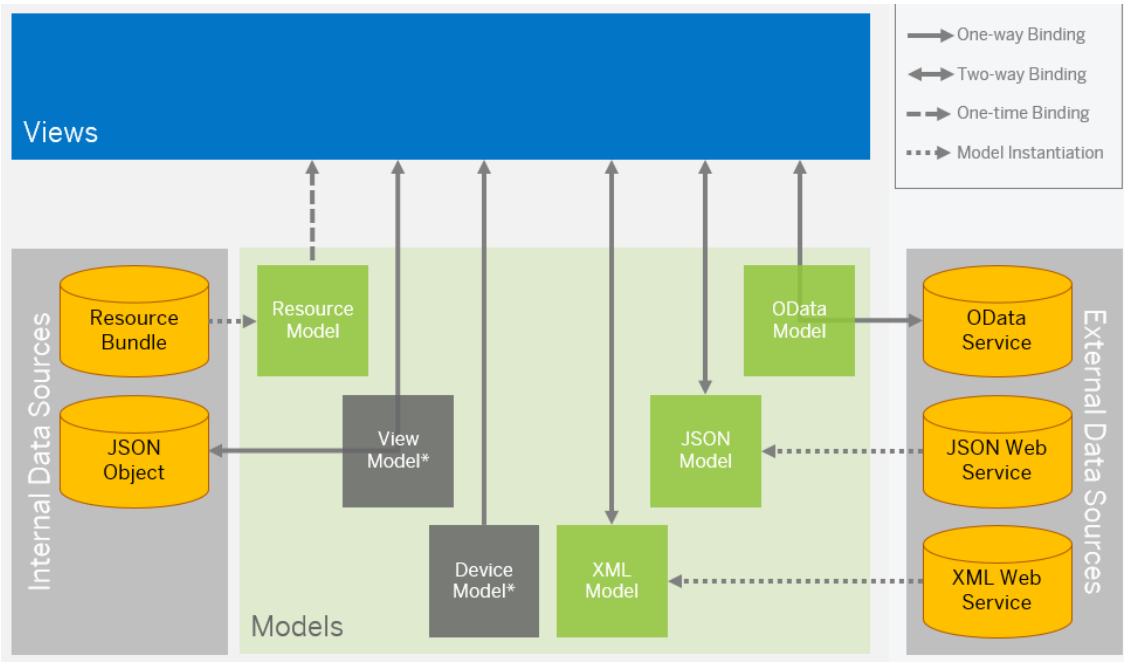


FIGURE 4-1 Databinding in SAPUI5 [9]

In practice, this means that the application logic or control elements (buttons, menus) should never directly manipulate tables or graphs including the underlying data.

Instead, the views are connected to data models with data binding and SAPUI5 will automatically update the view if the data in the model changes and vice versa. The controller should only influence the view by changing the data in the model.

4.1.2 Impact on the development of the *Resource Cockpit*

The clear separation between view, model and controller automatically functions as a powerful realisation of the state management [10] concept and creates cleaner and more readable code [11].

During the development of the *Resource Cockpit* another advantage became apparent: Most Fiori design elements work with almost identical data models. This means changing chart types is often as easy as changing a single line of code in the view without any changes in the controller.

And conversely, when changing the application logic there is never a concern about breaking anything in the UI. This is exemplified in the following figure 4-2:

```
1 <viz:VizFrame id="top-db-connection-usages-heatmap"
2               uiConfig="{applicationSet:'fiori'}"
3               width="100%"
4               height="{/heatmap/chartHeight}"
5               vizType='heatmap'
6               selectData="onDbConnectionUsageHeatmapSelect"
7               vizProperties='{ . . . }'
8  <viz:dataset>
9    <viz.data:FlattenedDataset data = "{/dbConnectionTopUsers}">
```

FIGURE 4-2 Databinding during UI design

To change the chart type in the UI of the *Resource Cockpit* only line 5 in figure 4-2 needs to be changed. The data is stored in the "*dbConnectionTopUsers*" model, which can be seen in line 9. Since the configuration is the same for most Fiori elements (see line 2), the same data will work with different chart types.

4.2 Spring framework

The Spring framework is an open-source framework released in 2002. At its core, Spring is a Java platform application framework. The most important feature is inversion of control with dependency injection. [12]

4.2.1 Inversion of control and dependency injection

The core idea behind Dependency Injection is to make sure an object receives all other objects it depends on. This means, you can use an object without having to worry about constructing it. A common example is to pass all dependencies to the constructor of a class. This already simplifies your code since you do not have to instantiate dependencies with new() or manage them in some global application class. However, these dependencies still need to be created manually at some point with their constructor.

This is where the Dependency Injection Container, the Spring framework, comes into play. Spring automatically keeps track of all dependencies and will automatically construct the dependencies and the object itself.

As an example, consider a class in the *Resource Cockpit* application that handles database requests (see figure 4-3)

Adding the annotation “@Component” (figure 4-3, line 1) tells Spring to create this object during component scanning, which happens when the application is initialized. The annotation “@Autowired” (figure 4-3 line 4) tells Spring to use the specified component. In this way, all dependencies in the project are automatically taken care of by Spring.

```
1 @Component
2 public class ResourceUsageRepo{
3
4     @Autowired
5     DataSource dataSource;
6
7     public Artifact findById(Integer id) throws SQLException {
8        try (Connection connection = dataSource.getConnection()) {
9               // prepared SQL statement, handle data, etc...
10        }
11    }
12 }
```

FIGURE 4-3 Spring Dependency Injection

4.2.2 Impact on *Resource Cockpit* development and unit testing

A big side-effect of a loosely coupled application created by relying on Dependency Injection is that during unit testing any component that should be tested can simply be “@Autowired” into the test without any worries about dependencies. This has allowed the development of comprehensive unit tests for the components of the *Resource Cockpit*.

Additionally, the Dependency Injection means that the *Resource Cockpit* is built in a modular manner: each component is responsible for very specific tasks.

This helps keep the code understandable and expandable for colleagues and future work on the *Resource Cockpit*.

5 DESIGN

The design of the *Resource Cockpit* follows the standard design of a web application deployed in the cloud. However, other tasks of this thesis, primarily the testing, required a local environment using mock data.

5.1 Production environment

The main purpose of the *Resource Cockpit* is to enable customers to monitor their data streams in the SAP cloud, for example to get information about the utilization of their resources. As the data is kept within the SAP cloud, the customer needs to access it from the outside via a web browser. The *Resource Cockpit* is located in the SAP cloud and will query a separate API for the required metadata and logs, stored in a central database, which can also be used by different applications. This separation of functionality allows containerized deployment in the Cloud Foundry[3] which has many advantages such as flexible deployment, scaling and management (see figure 5-1).

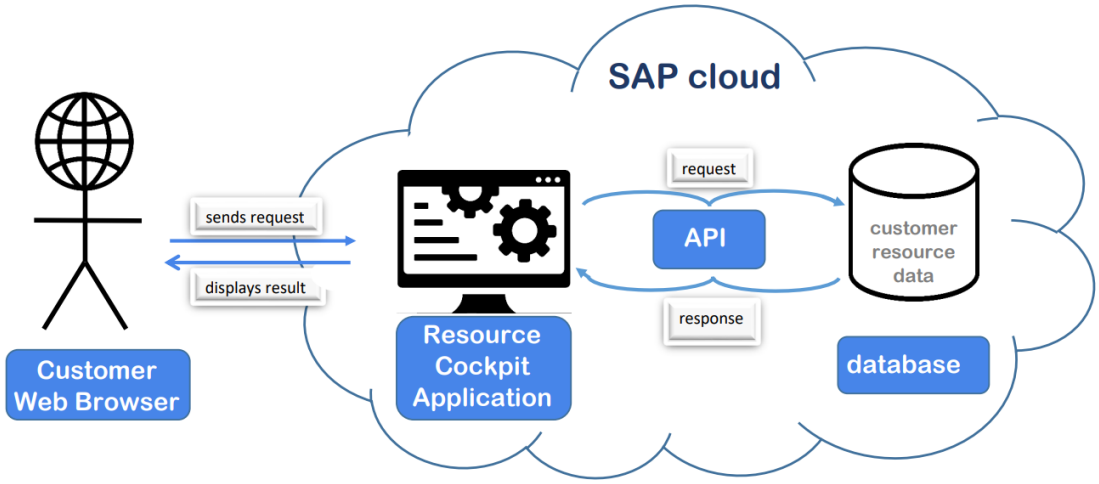


FIGURE 5-1 Resource Cockpit, production environment[4]

[3] Cloud Foundry is a cloud service provider service provider: https://www.cloudfoundry.org/
[4] Icons are used under free-to-use license from: https://uxwing.com/desktop-application-app-icon/

5.2 Local testing environment

During testing and UI design the *Resource Cockpit* is instead running locally, i.e. on localhost. The local environment has a similar design as the production environment; however, the API requests are instead redirected to a local version of the API and an embedded H2 database is used to store mock data (see figure 5-2). The next chapter contains a more detailed explanation of the implementation and generation of mock data.

The main advantage of this architecture is that it prevents development and tests from making unnecessary calls to the production API in the cloud. This has allowed the development and utilization of unit tests in the *Resource Cockpit*.

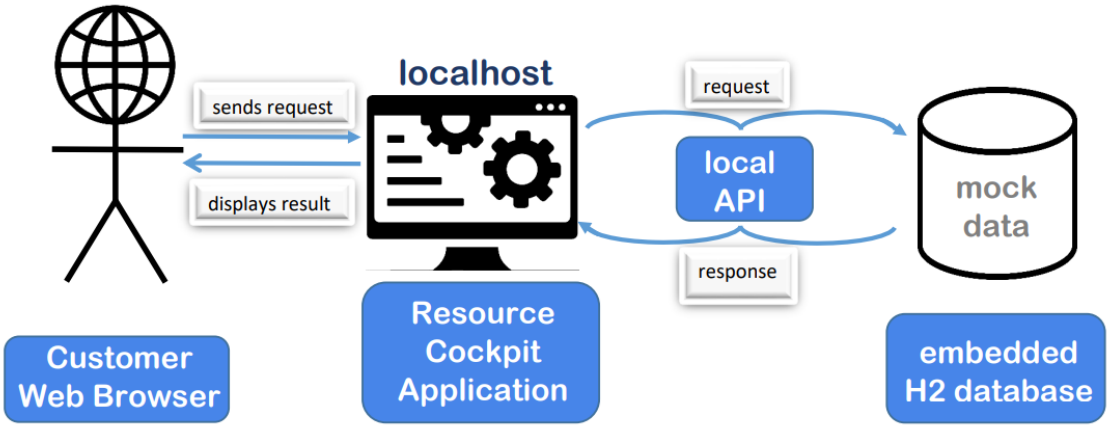


FIGURE 5-2 Resource Cockpit, local environment for testing

Since the local version of the *Resource Cockpit* is an important and interesting part of my thesis work, the implementation is explained in more detail in the following chapter.

6 IMPLEMENTATION

During development it became clear that unit tests are needed to ensure the functionality of the *Resource Cockpit* after updates or changes. However, querying the API in the cloud for every unit test was not an option due to request limitations. It would also have created a situation where unit tests might fail because of network problems and similar issues. This is not desirable since the unit tests should strictly test only the components of the *Resource Cockpit.*

As an alternative the *Resource Cockpit* has a local version which redirects all data requests to a local API. To ensure useful unit tests, this local API must respond with mock data resembling a response of the real API. Additionally, this local API has to store the mock data in a database to better emulate a real request and catch any errors in the database connection.

As a positive side effect, the implementation of the mock data allows local development of the UI without the need for the API in the cloud.

6.1 Local testing environment

The first step of creating the local testing environment is the development of a separate back-end service which utilizes a local database instead of the live data from the API in the cloud, as well as developing a module which generates mock data for this local service.

6.1.1 Development of local data provider

To achieve this goal, I created an additional web application in the project structure but excluded it from the production version. The local application should be available to developers but users should not get to see it. It should also not be uploaded to the Cloud Foundry platform, which is the PaaS we use to host the production application. The definition of this local application can be seen in the following figure 6-1.

```
1 Import ...
2
3 @SpringBootApplication
4 @EnableConfigurationProperties
5 @ServletComponentScan("com.sap.it.rc")
6 @ComponentScan(basePackages = {"com.sap.it.rc",
7        "com.sap.cloud.security"}, excludeFilters = {
8        @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value =
9        XsuaaServiceConfigurationDefault.class)})
10 @EnableTransactionManagement
11 @EnableMBeanExport(registration = RegistrationPolicy.IGNORE_EXISTING)
12 public class ResourceCockpitLocalWebApplication extends
13 SpringBootServletInitializer {
14    public static void main(final String[] args) {
15        SpringApplication.run(ResourceCockpitLocalWebApplication.class,
16        args);
17    }
18 }
```

FIGURE 6-1 Definition of local application

The “@ComponentScan” (figure 6-1 line 6) makes sure that all Spring Components are loaded in the local version of the *Resource Cockpit*, as explained in the Frameworks chapter. This means the local application uses the same components as the production version and adds functionality for a local database and mockdata generation.

With this local application in place, it is possible to create a request controller for a separate handling of the routing (see figure 6-2):

```
1 @RequestMapping(path = "/api/v1/resourceusage", produces =
2 "application/json")
3 public Resource getResourceUsage(@RequestParam("type") String type,
4        @RequestParam("from") @DateTimeFormat() Date from,
5        @RequestParam("to") @DateTimeFormat() Date to,
6        @RequestParam(value = "time", required = false, defaultValue =
7          "daily") ResourceUsageProvider.TimeGranularity time,
8        @RequestHeader Map<String, String> headers) {
9
10    if (localDataProvider != null) {
11        localDataProvider.createTestData(from, to, time); {
12    }
13    return executeRequest(type, from, to, time, headers); {
14 }
```

FIGURE 6-2 Example of route handling

Instead of a call to a database API in the SAP Cloud environment the request is handled locally. First test data is created and stored in an embedded H2 database[5] by the "*LocalDataProvider*" object (figure 6-2, line 11), then the data is read from this H2 database (figure 6-2, line 13). It is important to point out that the request (figure 6-2, line 13) is handled by the component as it would be in the production environment but it is executed on the embedded H2 database instead, which allows the automated tests to check the database request handling without live data. This is less resource intensive and allows development without access to internal SAP systems.

Implementing it in this way, instead of responding with mock data directly without the database step in between, improves the unit tests which are included in the *Resource Cockpit*.

The local request controller can also serve static files in place of any other API call. This has simplified the development of the local request controller in cases where there is no need to generate mock data (see figure 6-3):

[5] H2 database is a relational databse system, which can be directly embedded into Java applications: https://github.com/H2database/H2database

```
1 @RequestMapping(path = "/Operations/KnownArtifactsListCommand",
2                 produces = "application/xml")
3 public org.springframework.core.io.Resource getKnownArtifacts() {
4    try {
5        return new ClassPathResource("sample-data/packageNames.xml");
6    } catch (Exception e) {
7        throw new RuntimeException(e);
8    }
9 }
```

FIGURE 6-3 Route handling with static response

In this example, the production application would call an API to get all currently deployed integration flows. This information is needed to match IDs with names. While this list can and does change my team decided that in this case a static list of names is sufficient for testing purposes.

With a separate local application is it also easy to set any configuration values to be used during local development and automated tests.

```
1 @Configuration
2 public class LocalConfig {
3    @Value("${resource-cockpit.local.random-seed:123}")
4    public long randomSeed;
5    @Value("${resource-cockpit.local.distribution:MIXED}")
6    public String distribution;
7
8    /* more config values removed for brevity */
9 }
```

FIGURE 6-4 Local Configuration values

The “@Configuration” annotation (figure 6-4, line 1) allows Spring to “@Autowire” the configuration file automatically in any place it is needed.

6.1.2 Providing mock data

The next step of development was to create a data provider which generates the mock data of the simulated API response (see figure 6-5).

```
1 public void createTestData(Date from, Date to,
2 ResourceUsageProvider.TimeGranularity time) {
3
4    if (time == ResourceUsageProvider.TimeGranularity.hourly){
5        localConfig.setTimeToSendHours(1);
6    }
7    setConfigParams(from, to);
8
9    localResourceUsageRepo.deleteOldResourceData();
10   List<ResourceUsagePayload> payloadList =
11     createResourceUsagePayloads();
12   for (ResourceUsagePayload payload : payloadList) {
13     resourceUsageRepo.persistResourceUsage(payload);
14   }
15 }
```

FIGURE 6-5 Creating mock data

The three main steps when generating mock data are as follows:

1. First, any existing data is deleted from the database (figure 6-5, line 9) which is important for automated tests since they create data several times.
2. Then mock data is generated to match the incoming request (see figure 6-5, line 10-11). Datapoints are generated for every hour and for every node to match real data.
3. Finally, the generated mock data is stored in the embedded H2 database (figure 6-5 line 13), ready to be queried.

Aside from several information like process id, system id, tenant id and timestamp, the most interesting part of the mock data are the usage values which are generated from an array of values which is created semi-randomly to resemble a real situation.

In the next sub-chapter, I explain how I decided on the pattern of the mock data and why there are multiple patterns.

After the local data provider was working, two questions came up in the team:

1. Is it possible to use the local application also with live data?
   This was important for using the local application as a testing ground for changes to the production application.
2. Is it possible to prevent mock data generation if the application is not running with an embedded H2 database but instead with the local postgres database, which was the default database during development?

The answer to both these questions is the dependency injection feature of the Spring framework:

By mapping the data provider Spring Bean[6] to the “local-H2” profile and declaring it as an optional dependency in the request controller mock data is only created if the application is running with the H2 database (see figure 6-6 line 1 & line 4).

```
1 @Profile({"local-H2"})
2 @DependsOn({"liquibase"})
3 public class LocalDataProvider {
```

```
4 @Autowired(required = false)
5    LocalDataProvider localDataProvider;
```

FIGURE 6-6 Spring Bean annotations to run local data provider only with H2 profile.

In conclusion, the local application is loosely connected into the larger application, especially the automated tests, with the Spring @Autowired feature, explained in the Frameworks chapter (see figure 6-7).

[6] In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. [16]

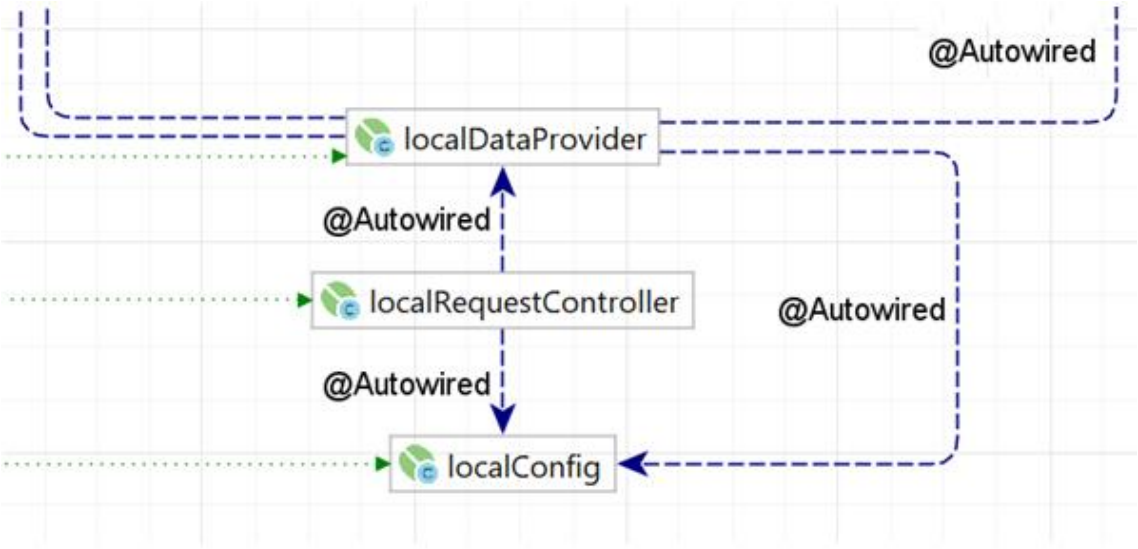


FIGURE 6-7 Application structure

After setting up the testing environment to use mock data, it is important to look into generating mock data which represents the real data.

6.1.3 Creating realistic mock data

As a first step I decided to investigate several live tenants[7] and analyse the structure of the real data.

During this analysis, I found that generally all tenants can be categorized into three categories:

1. Tenants which have a relatively homogenous usage of the integration flows and this usage is fairly small.
2. Tenants which have a relatively homogenous usage of the integration flows and this usage is fairly close to the recommended maximum.
3. Tenants which have very uneven distribution of integration flow usage.

[7] Every tenant belongs to a SAP customer, who is using resources of the SAP cloud.

Further study of the tenants with uneven distribution, i.e. category 3 tenants, also showed the following similarities:

- The “system”[8] channel will have by far the biggest usage of resources. This is around 10% to 15% of the total available resources.
- There are 5-10 integration flows with a constant medium usage of around 5% to 8% of the total available resources.
- There are around 10-15 integration flows with a constant but fairly small usage of 1% to 3% of the total available resources.
- Most other integration flows have only sporadic usage. These integration flows are idle most of the time and only use resources in peaks. These peaks are of varying intensity, but mostly around 5% to 10% of the total available resources.

With this differentiation of the possible usage pattern, I was able to develop realistic mock data. I have developed my local data provider in such a way that it will generate mock data in one of the three patterns above depending on which system variables are set. For convenience, I use a set of local variables which are injected with the Spring dependency injection on start-up. This way the local configuration can also easily be overwritten if needed.

Setting it up in this way was important since it fulfils two development requirements, which I set:

1. It is easy to start the *Resource Cockpit* in a local mode using mock data while working on the UI.
   It is not possible to develop useful and clear UI elements like graphs or tables without having data to display.

[8] ”system” is listed in the list of integration flows, but it is strictly speaking not an integration flow.

2. The local data provider is used in the Spring unit tests. Therefore, it is important that all variables can be set dynamically from code.

During meetings with colleagues an additional requirement came up:

3. There should be some randomness in the data but with the ability to use a fixed seed in case it is necessary to reproduce the same data again.

To fulfil this last requirement, I added a random number generator to my code and another system variable for the seed of this generator.

6.2 Implementation of the UI

The controller of the frontend, which handles the functional part according to the SAPUI5 MVC [13] design, is written in JavaScript and handles all database request, data filtering, calculating aggregated data, etc..

The controller is separated from any design of the UI. Instead, the UI design is specified in XML views following the SAPUI5 convention.

6.2.1 Controller

The data behind the UI elements is saved in connected data models as explained in the Frameworks chapter. Instead of directly accessing the UI, the controller only changes the data models (line 6 to 9 in figure 6-8 below). The SAPUI5 framework will then automatically update the view.

```
1 press: function(e){
2   let dFrom = new Date(selectedContext.timestamp);
3   let dTo = new Date(dFrom.getTime() + (1000*3600*24));
4   oPopOver.close();
5   oPopOver.destroy();
6   oModel.setProperty("/state/selectedTimeKey", 'CUSTOM');
7   oModel.setProperty("/state/dateFrom", dFrom);
8   oModel.setProperty("/state/dateTo", dTo);
9   oModel.setProperty("/state/timeUnit", 'hourly');
10
11  oController.navigateToDbConnections();
```

FIGURE 6-8 Example of using a data model

Additionally, whenever the application needs to load new data (in this example when the user changes the timeframe during which he wants to see data), this is handed to the router attached to the application (line 11 in figure 6-8).
This has simplified the development as only the initialization of the page needs to be handled manually. SAPUI5 can then handle updates automatically.

6.2.2 View

The SAPUI5 framework contains libraries developed for the Fiori guidelines [4]. Elements from these libraries are used with minimal modification in the *Resource Cockpit*.
This has the advantage of automatically fulfilling the design requirements of SAP applications having to adhere to the Fiori standards, such as usability and accessibility, e.g., colour-blind friendly schemas (figure 6-9).

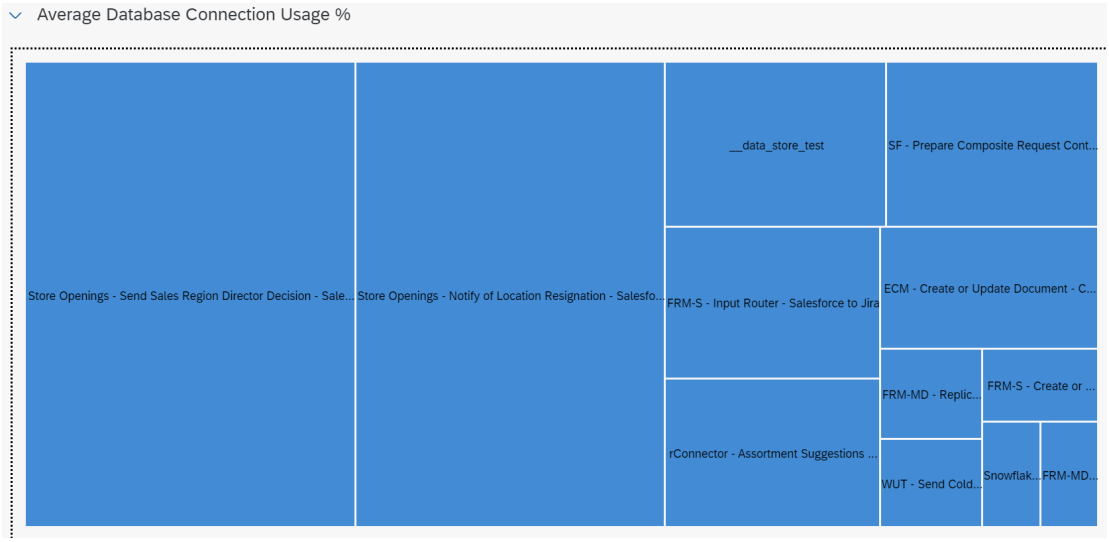


FIGURE 6-9 Example of an element that can be used ”out of the box” – see appendix 4 for full size view

Many of these elements are however as minimalistic as possible. While this leads to a clean and less cluttered design, it has the disadvantage of missing some information.

To improve the predefined design elements the *Resource Cockpit* has additional pop-ups with detailed information and context actions (figure 6-10).

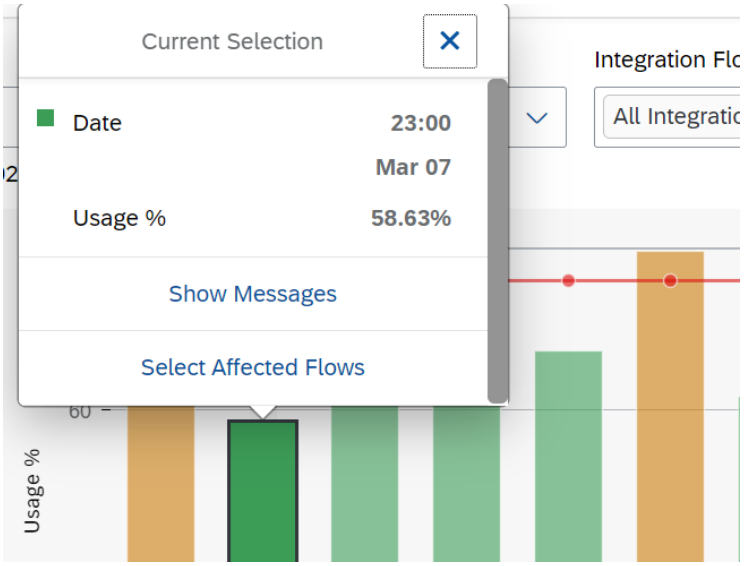


FIGURE 6-10 A Pop up showing more information and context actions

This way, the *Resource cockpit* keeps the clear and concise look of a Fiori application but also solves some restrictions of these predefined elements.

7 TESTING AND EVALUATION

Testing of the *Resource Cockpit* is split into three parts. Firstly, the unit tests integrated in the Spring Framework. These tests are used to test the Java components which make up the backend of the *Resource Cockpit*.

Secondly, the unit tests for the JavaScript frontend of the application. These tests utilize the QUnit framework [14] and test the functionality of the frontend, i.e. the controller of the SAPUI5 application.

Finally, usability testing was performed with experts; in this case a survey among my colleagues to get input on design decisions.

7.1 Unit testing with Spring

The unit tests for the backend of the *Resource Cockpit* utilize the framework support of Spring Boot. After including the necessary dependencies, the main one being the spring-boot-starter-test [15], it is possible to bootstrap the entire application context with the “@SpringBootTest” (figure 7-1, line 1) annotation. This means, any Spring Bean that was picked up during component scanning with “@ComponentScan” (see figure 6-1, line 6) can be “@AutoWired” (figure 7-1, line 4) into the test.

```
1 @SpringBootTest
2 class LocalDataProviderTest {
3
4    @Autowired
5    LocalDataProvider dataProvider;
6
7    @Test
8    void generateUsageValuesEvenDistributionTest() { // Test Code
9    }
```

FIGURE 7-1 Example of a test with the Spring framework

Spring will then load test-specific parts of the application automatically, which is one of the best parts and main advantages of working with this framework.

In this way, Spring provides a very powerful framework to implement unit tests for all components of a Spring application.

7.2 QUnit tests

Unit tests for the JavaScript frontend are realized with the QUnit framework.

The purpose of these tests is to check the functionality of the frontend and detect problems with data handling etc.

FIGURE 7-2 Resource Cockpit UI and Results of QUnit tests

By including the results of the unit tests (red box in figure 7-2) in the page of the application, it is easy to detect any problems or programming errors which were introduced while developing additional functionality.

7.3 Test cases and results of unit tests

Spring

For the Spring components the unit tests are focused on checking the functionality of every component. Since a Spring application is modular in nature and every component should only have a single, well-defined function with a predictable output it is possible to directly compare the output with the expected value.

```
1 @Test
2 void generateUsageValuesEvenDistributionTest() {
3     /* Set up, create config & generate requestParams ...
4        Removed from screenshot for brevity */
5
6     double[] values = dataProvider.generateValues(requestParams);
7     assertTrue( Arrays.stream(values).sum() <
8         dataProvider.calculateTotalAvailable(requestParams) );
9 }
```

FIGURE 7-3 Example of a Spring unit test

Figure 7-3 is an example of a test for the mock data provider described in Chapter 6. In line 6 mock values are generated and in line 7+8 is a test if the sum of the generated values is below the maximum available.

This test is used to check two conditions: Firstly, it checks if the data provider component is working and returning values. If it isn't, then line 6 will already throw an exception. Secondly, it checks if the data provider returns sensible mock data. The second condition was added after some mistakes with loop iterators and too big span of randomness caused the mock values to go over 100% of the total available which does not work with the frontend UI.

The Spring unit tests proved most valuable for detecting problems with interfaces between modules. For example, a module loading data from a database or API might no longer work on a different system (Windows or Mac) or due to other changes to the database or API even if the module itself was not changed.

Qunit

The unit tests for the frontend UI are focused on *user stories* instead of pure functionality of components. For example, a user story could be “The user clicks on button A and expects to see xyz”. The test will simulate this user interaction, i.e. the button press, and check if the expected outcome is visible. However, the implementation of tests for the frontend UI is more challenging for two reasons.

Firstly, the controller for the frontend is not modular but controls the entire UI with multiple functions. This means there is not a single test case for the controller but multiple test cases to test all functionality which makes these tests more complex. In addition, some test cases need to test a combination of inputs. For example, the user might apply a filter before or after loading data and both cases need a separate test case even though the “load data” function is the same.

Secondly, the controller does not have an output aside from the UI, so we can’t compare return parameters with an expected value. Instead, we have to simulate the user input (line 2 in figure 7-4) and then check the state of the UI which is stored in the model (lines 3-7 in figure 7-4). The test function is then called from the QUnit module (line 10-13 in figure 7-4).

```
1 function hideFilterbarButtonTest(assert, sValue, fExpectedValue) {
2    controller.onHideFilterButtonPress();
3    let filterbarVisible =
4        oModel.getProperty("/filterbar/filterbarVisible");
5
6    assert.strictEqual(filterbarVisible, fExpectedValue, "hide
7        filterbar");
8 }
9
10 QUnit.test("Should hide filterbar on button press", function (assert)
11 {
12   hideFilterbarButtonTest.call
13     (this, assert, false /*no input for this  test*/, false);
14 });
```

FIGURE 7-4 Example of a Qunit test

7.4 User feedback and usability survey

While the unit tests cover the functionality of the frontend and backend of the *Resource Cockpit*, one issue that remains is the usability of the application. Since this issue cannot be covered with automated tests, the design and control elements of the UI were instead guided by feedback from SAP customers and discussions with colleagues.

This feedback can be primarily put into two categories:

1. Advice from colleagues, especially graphic designers, on colour schemas, designs and UI elements that work well or are important to consider.
2. Feedback from (possible) users on what does *not* work.

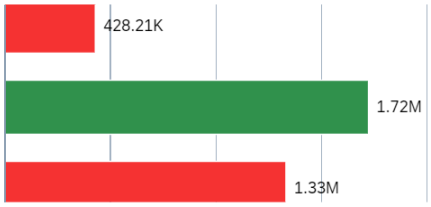


FIGURE 7-5 graph rejected by a designer

This design (see figure 7-5) was rejected based on feedback from a graphical designer colleague. It is not inclusive for red-green colour blindness.

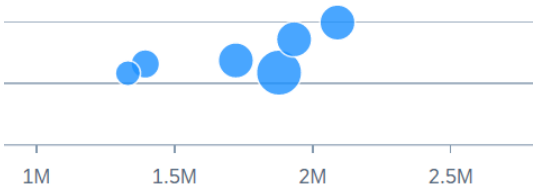


FIGURE 7-6 graph rejected by a user

Other designs (see figure 7-6) were rejected by users even though they are new and highlighted in the Fiori guidelines. Users are not familiar with them and do not know how to interpret it.

To summarize my own findings: It takes users time to get used to new designs. Often they prefer designs they are already familiar with so the risk to misinterpret them is very low.

Usability survey

In order to get specific feedback on the *Resource Cockpit* and the features an online survey was implemented with ten questions distributed to the members of my team, using their responses as feedback from expert users. 8 colleagues submitted their answers.

The survey included seven questions which used a common 1-5 star rating (5 being the best mark and 1 being the lowest mark) as well as 3 questions for free-text answers.

FIGURE 7-7 sample of a star rating

The distribution of the answers is displayed in the following graphs, where the x-axis represents the number of stars and the y-axis the number of answers.
The tool used for the survey is a freeware tool called Survio [3].
In order to get a detailed view of the feedback the answers are now presented individually:

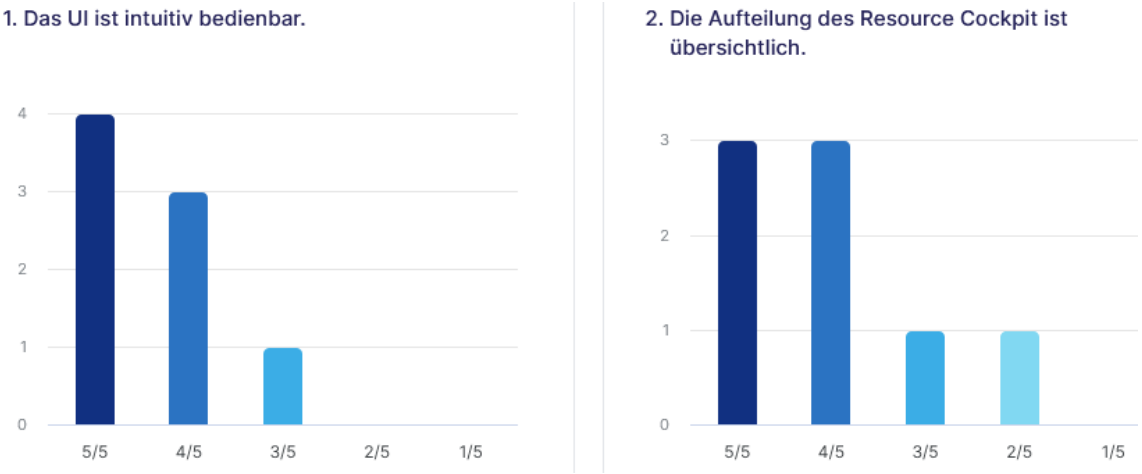
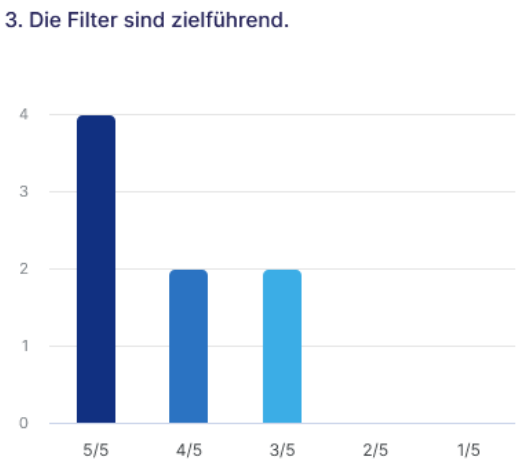


FIGURE 7-8 Q1 & Q2 of the survey

Q1: *The UI can be used intuitively*

Q2: *The layout of the Resource Cockpit is clear*



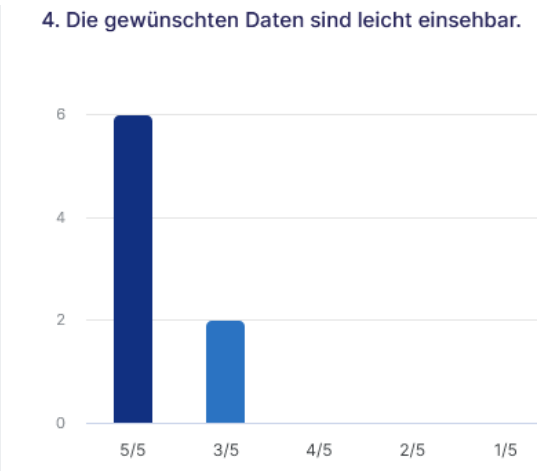


FIGURE 7-9 Q3 & Q4 of the survey

Q3: *The filters are effective*

Q4: *The data you want is easy to see*

5. Gibt es eine Information die zusätzlich angezeigt werden sollte?

ANTWORT	ANTWORTEN	VERHÄLTNIS
	6	75%
Detail view not obvious to reach	1	12.5%
Graph not good on small screen	1	12.5%

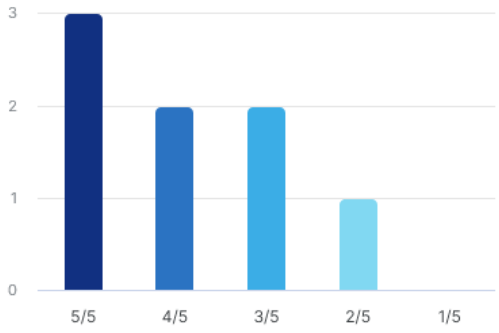


FIGURE 7-10 Q5 & Q6 of the survey

Q5: *Is there any additional information that should be displayed?*

Q6: The graphs represent the sought information well.

7. Falls die Graphen nicht gut sind, welcher Graphentyp ist deiner Meinung nach besser?

ANTWORT	ANTWORTEN	VERHÄLTNIS
	6	75%
Box Graph	1	12.5%
Graph does not scale well with small screen size	1	12.5%

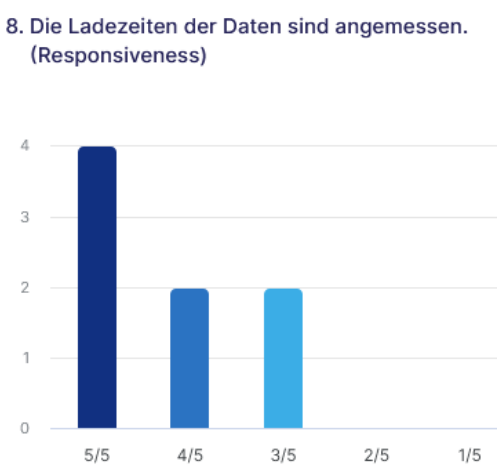
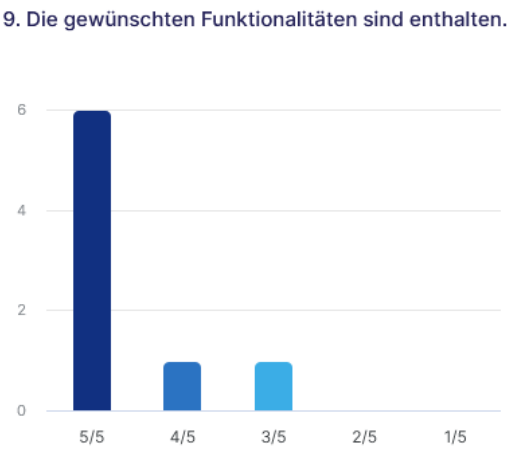


FIGURE 7-11 Q7 & Q8 of the survey

Q7: If the graphs are not good, which graph type do you think is better?

Q8: The loading times of the data are reasonable.



10. Gibt es ein Feature, das dir fehlt?

ANTWORT	ANTWORTEN	VERHÄLTNIS
	8	100%

FIGURE 7-12 Q9 & Q10 of the survey

Q9: *The desired functionalities are included.* *Q10: Is there a feature you are missing?*

Since most answers to the star question were 3 or above stars and there are no suggestions in Q10, I conclude that the users are generally satisfied with the UI of the *Resource Cockpit.* Of particular interest are the text answers pertaining to the scaling of the UI. This suggests further attention in a next iteration step of the *Resource Cockpit.*

8 RESULTS

8.1 Current state

As the result of this thesis work, a web application called *Resource Cockpit* was implemented which users can use to get an overview of their cloud resources. Especially current workload and problems can be detected easily.

Based on the survey presented in Chapter 7.4, the *Resource Cockpit* meets the original goal of exposing such information to SAP customers in a user-friendly manner.

Additionally, there is now an extensive framework for unit tests included in the application which will support further development of the application such as the addition of more resource types.

8.2 Impact of the *Resource Cockpit* for SAP and SAP customers

The *Resource Cockpit* can be used to monitor and visualize the utilization of SAP resources like integration flows used by enterprise customers in their workflows. This makes it easier for enterprise customers to understand and evaluate the use and value they get out of these resources. The *Resource Cockpit* helps with detecting and understanding problems and inefficiencies in their own workflows which is a topic all companies are concerned about.
With an easy-to-use way to visualize the benefits of SAP resources to enterprise customers it is also easier for SAP to market and advertise those resources to potential enterprise customers. The user-friendly and easy to understand graphical UI of the *Resource Cockpit* helps someone not familiar with SAP products understand and properly utilize SAP resources. This makes it easier for the marketing team to advertise and sell to potential customers.

8.3 Lessons learned

During this thesis I could apply a lot of the knowledge learned during my studies, especially courses on web applications. Naturally, this real work has provided me with a much more in-depth understanding of the topic. Especially the Java Spring framework is important outside of the SAP environment as well. Alongside programming skills, I have also learned a lot about SAP technologies like integration flows and ERP.

A painful lesson I made is on the implementation of testing. In retrospect it would have been easier to start with test driven development instead of implementing unit tests only at the end. I had to re-do parts of the code to make them testable properly. The result is a cleaner and better structured code.

8.4 Future outlook

At the time of writing this report, the unit tests for the Java components making up the backend of the application have full code coverage and are complete. However, the unit tests for the JavaScript frontend only cover the most important parts of the application and should be expanded.

Also, there are no integration tests at this point which will become important when the *Resource Cockpit* is integrated into the SAP landscape.

Based on the answers to the survey Q5 and Q7 some attention needs to be given to proper scaling of the *Resource Cockpit* on small screens.

In the future the *Resource Cockpit* will be released in an alpha state to customers, who use it in their real workflows. There should be several iterations before a final release to address customer issues and to include additional resource types for monitoring.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Stefan Opderbeck and his entire RET team at SAP Waldorf for welcoming me into their team so warmly throughout the Company Oriented Projects period. In particular, Mitko Kolev, to whom I was technically assigned, always had time to help me familiarize myself with the tasks. Especially at the beginning of my project work when some content was still new to me his help was invaluable. In addition to the many technical contents that I learned during this time, the value of a good and harmoniously working team became clear to me. Thanks! I felt very comfortable with you.

And of course, I would like to thank Lasse Haverinen, who coached me academically through this thesis and always had time for me when I needed his advice. A lot of the content I learned in his courses helped me to support the SAP team during my Company Oriented Projects and thesis work.

REFERENCES

1. SAP, „What is SAP?,“ [Online]. Available: https://www.sap.com/about/company/what-is-sap.html. [Zugriff am 14 01 2023].

2. SAP, „SAP Company Information,“ [Online]. Available: https://www.sap.com/about/company.html. [Zugriff am 14 01 2023].

3. „Survio tool,“ [Online]. Available: https://my.survio.com/.

4. SAP, „SAP Fiori Design Guidelines,“ 2023. [Online]. Available: https://experience.sap.com/fiori-design/. [Zugriff am 02 02 2023].

5. R. B. De, „SAP Community (Blog),“ 2013. [Online]. Available: https://blogs.sap.com/2013/05/21/sap-pi-for-beginners/. [Zugriff am 02 02 2023].

6. UpKeep, „Enterprise Resource Planning (ERP),“ 2019. [Online]. Available: https://www.upkeep.com/learning/enterprise-resource-planning-erp-software . [Zugriff am 02 02 2023].

7. SAP, „Understanding Integration Flow,“ [Online]. Available: https://help.sap.com/doc/saphelp_nw73ehp1/7.31.19/en-US/76/94c168d14e4baa8a7c3b5dea209ad4/frameset.htm. [Zugriff am 14 01 2023].

8. A. Grosskopf, G. Decker und M. Weske, The Process. Business Process Modeling Using BPMN, Meghan-Kiffer Press, 2018.

9. SAP, „SAPUI5 Documentation - Data Binding,“ SAP, 06 10 2022. [Online]. Available: https://sapui5.hana.ondemand.com/sdk/#/topic/68b9644a253741e8a4b9e4279a35c247.html. [Zugriff am 01 02 2023].

10. D. Bugl, Learning Redux, Packt Publishing, 2017.

11. R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson, 2008.

12. C. Walls, Spring Boot in Action, Manning, 2016.

13. P. Modderman, C. Goebels, D. Nepraunig und T. Seidel, SAPUI5 The Comprehensive Guide, SAP PRESS, 2020.

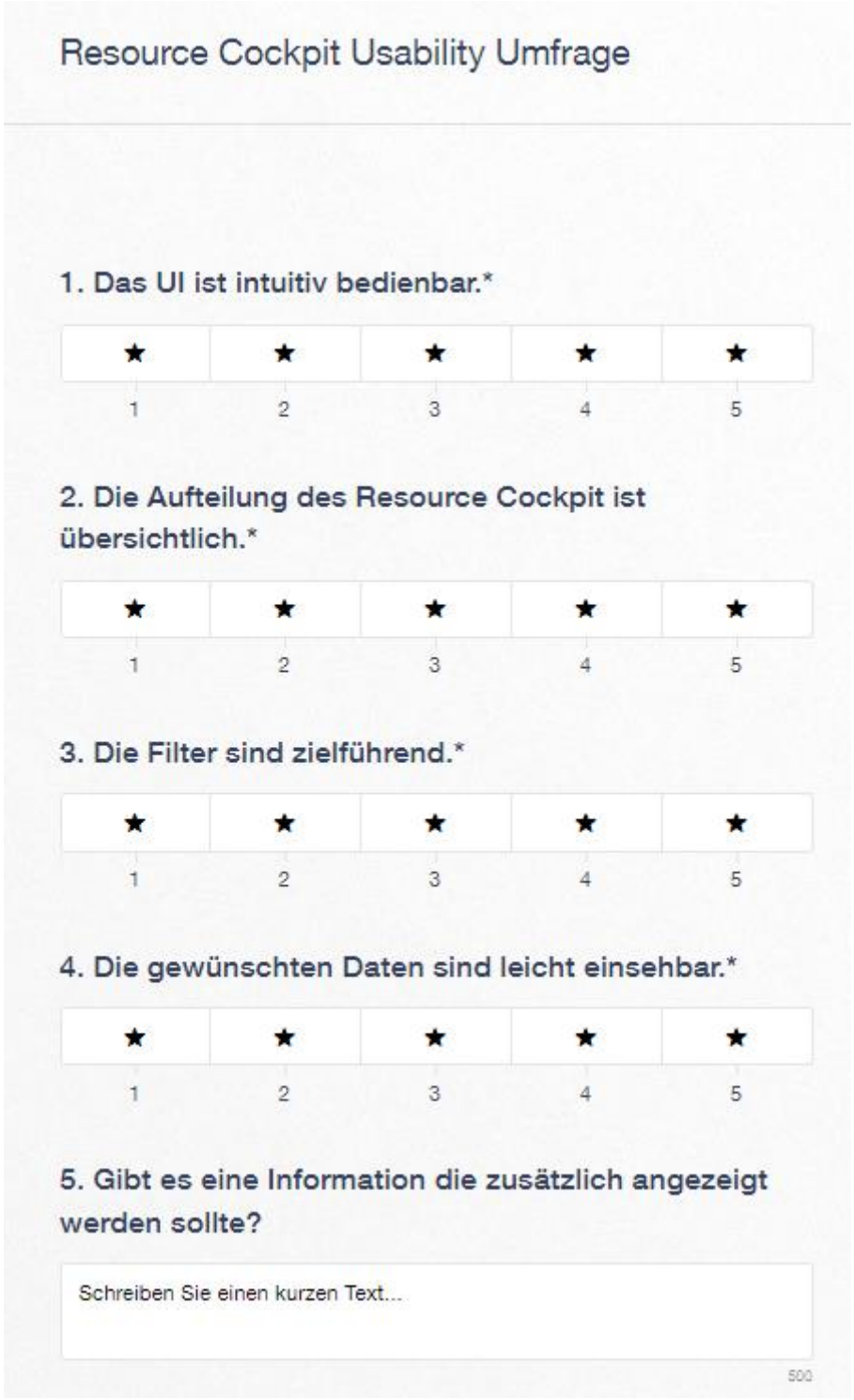
14. QUnit, „About QUnit,“ QUnit, [Online]. Available: https://qunitjs.com/about/. [Zugriff am 10 03 2023].

15. Maven Repository, „Spring Boot Starter Test,“ Maven, [Online]. Available: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test. [Zugriff am 10 03 2023].

16. N. T. Nguyen, „What is a Spring Bean?,“ Baeldung, 17 06 2021. [Online]. Available: https://www.baeldung.com/spring-bean. [Zugriff am 05 02 2023].

APPENDICES

Appendix 1: Usability Survey



Resource Cockpit Usability Umfrage

1. Das UI ist intuitiv bedienbar.*

★ ★ ★ ★ ★

1 2 3 4 5

2. Die Aufteilung des Resource Cockpit ist übersichtlich.*

★ ★ ★ ★ ★

1 2 3 4 5

3. Die Filter sind zielführend.*

★ ★ ★ ★ ★

1 2 3 4 5

4. Die gewünschten Daten sind leicht einsehbar.*

★ ★ ★ ★ ★

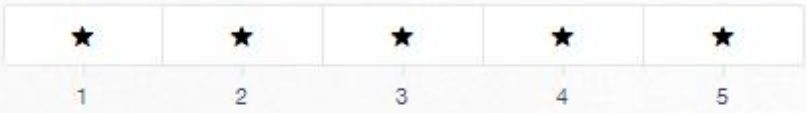
1 2 3 4 5

5. Gibt es eine Information die zusätzlich angezeigt werden sollte?

Schreiben Sie einen kurzen Text...

500

6. Die Graphen stellen die gesuchte Information gut dar.*



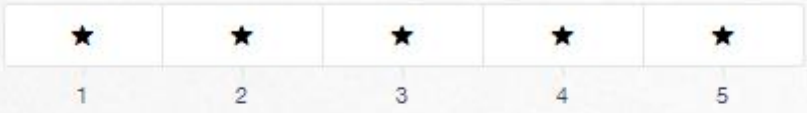
7. Falls die Graphen nicht gut sind, welcher Graphentyp ist deiner Meinung nach besser?

Schreiben Sie einen kurzen Text...

500

8. Die Ladezeiten der Daten sind angemessen. (Responsiveness)*

9. Die gewünschten Funktionalitäten sind enthalten.*

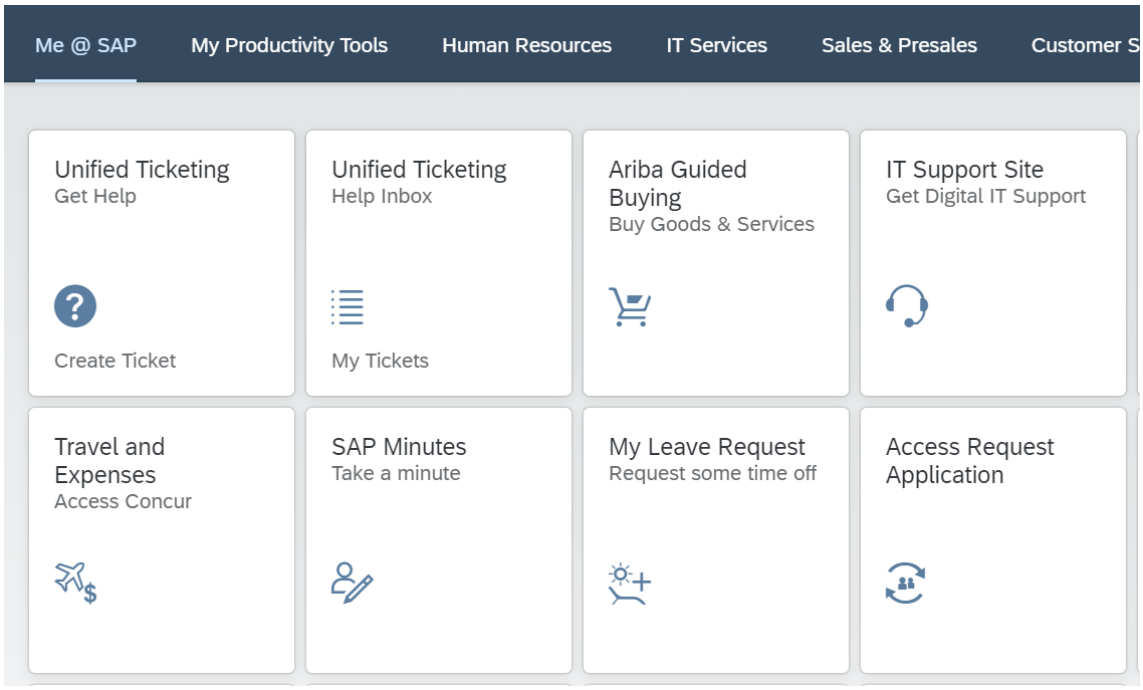


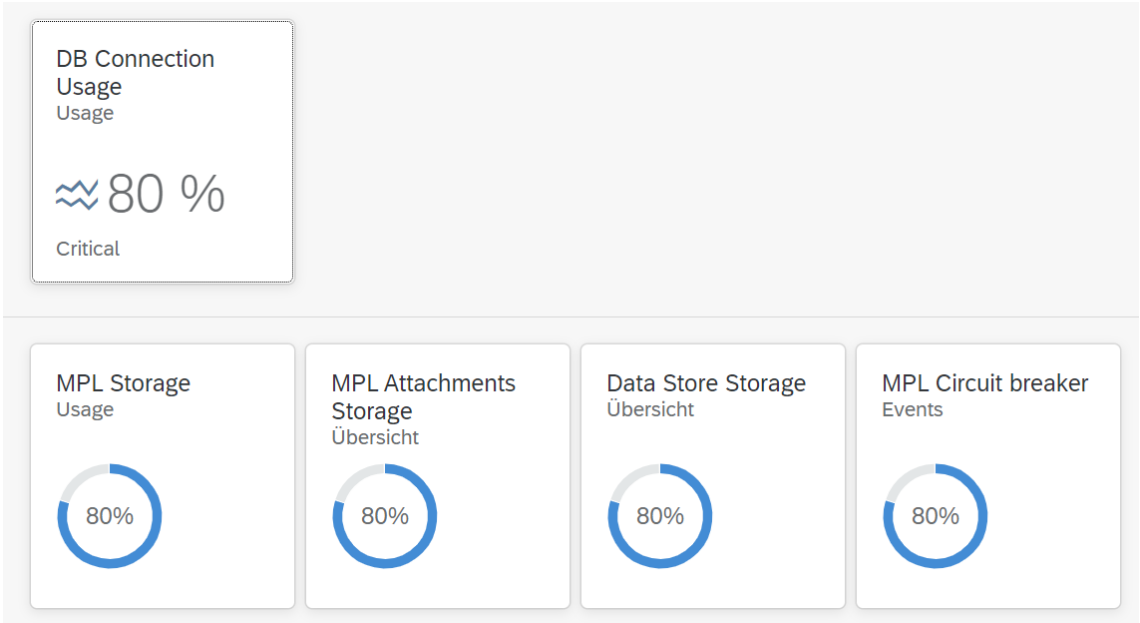
10. Gibt es ein Feature, das dir fehlt?

Schreiben Sie einen kurzen Text...

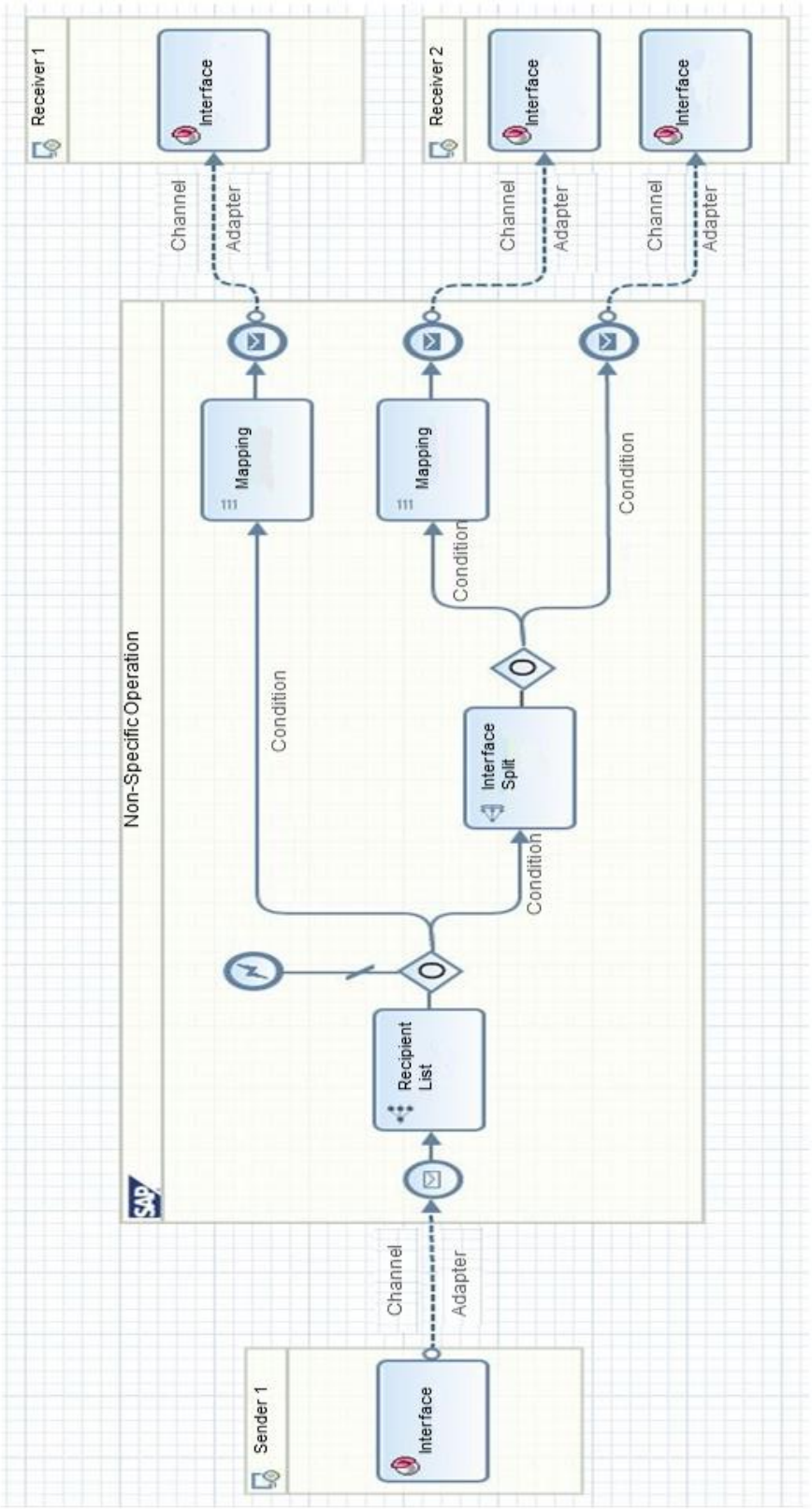
500

Appendix 2: UI comparison of an SAP HR application and the *Resource Cockpit* UI





Appendix 3: Elements in an integration flow



Appendix 4: Treemap chart from the *Resource Cockpit*