

FAKULTÄT FÜR INFORMATIK  
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master-Praktikum - Enterprise Software Engineering am Beispiel von SAP

## **Project Documentation**

### **Corporate Sustainability Reporting**

Author:	Manuel Römer Minh Nhu Duong Tim Köneke
Advisors:	Ann-Christin Fleischle Philipp Landler
Submission Date:	20.07.2023



## Table of Contents

<b>Table of Contents .....</b>	<b>III</b>
<b>List of Figures.....</b>	<b>V</b>
<b>List of Tables .....</b>	<b>VI</b>
<b>Motivation.....</b>	<b>7</b>
<b>Requirements Analysis .....</b>	<b>8</b>
<b>Themes / Epics .....</b>	<b>8</b>
<b>User Stories / Tasks .....</b>	<b>8</b>
<b>Approach .....</b>	<b>11</b>
<b>Agile Roadmap .....</b>	<b>11</b>
Requirements Analysis.....	11
Sprint 1 - Setup & Requirements Refinement (29.05.23 – 04.06.23) .....	11
Sprint 2 - PoC Development & Progress Presentation (05.06.23 – 18.06.23) .....	12
Sprint 3 - Finalizing the Core Product (19.06.23 – 02.07.23).....	12
Sprint 4 - Fixes / Challenges / Submission (02.07.23 – 20.07.23).....	13
<b>Software Development .....</b>	<b>14</b>
<b>Testing Methodology .....</b>	<b>14</b>
<b>Software Architecture and Data Models .....</b>	<b>15</b>
<b>Software Architecture .....</b>	<b>15</b>
<b>Semantic Data Model .....</b>	<b>16</b>
<b>Technical Data Model .....</b>	<b>17</b>
<b>Implementation .....</b>	<b>20</b>
<b>SAPUI5 Application .....</b>	<b>20</b>
Project Setup and Requirements .....	20
TypeScript .....	22
Code Generation.....	23
State Management .....	24
Project Structure .....	31
Form Engine .....	31
Form Builder .....	32
<b>ABAP Code .....</b>	<b>33</b>
<b>Project Challenges .....</b>	<b>36</b>
<b>Form Engine .....</b>	<b>36</b>
<b>Form Builder .....</b>	<b>36</b>
<b>Form Engine Performance .....</b>	<b>36</b>

Getting Familiar with the CSRD and ESRS .....	37
Responsibilities .....	38
Project Implementation .....	38
Project Documentation .....	38
Access Information .....	39
User Manual .....	40
Homepage with four options: .....	40
Questionnaire Management Page: .....	40
Form Builder Page: .....	41
Customer Management Page: .....	45
Questionnaire Page: .....	46
Appendix .....	47
User Stories .....	47

## List of Figures

Figure 1: Agile Roadmap .....	11
Figure 2: Application Architecture .....	16
Figure 3: Semantic Data Model .....	16
Figure 4: Technical Data Model (Backend) .....	17
Figure 5: Technical Data Model (Frontend/Form Schema) .....	18
Figure 6: The States of An Asynchronous Operation .....	29
Figure 7: Form Builder Feature Areas .....	33
Figure 8: Expression Tree Editor .....	33
Figure 9: User Story Assignments .....	38

## List of Tables

Table 1: User Stories .....	10
Table 2: Requirements Analysis Stories .....	11
Table 3: Sprint 1 Stories.....	12
Table 4: Sprint 2 Stories.....	12
Table 5: Sprint 3 Stories.....	13
Table 6: Sprint 4 Stories.....	14
Table 7: NPM run Commands .....	21
Table 8: FormSchema OData Endpoints.....	34
Table 9: FormSchemaResult OData Endpoints .....	34
Table 10: Customer OData Endpoints .....	34
Table 11: Project Documentation Assignments .....	38

## Motivation

With global warming turning from a prediction into reality, the general public starts to take more interest in their ecological footprint as well as the companies' they are dealing with. Already back in 1997 the Global Reporting Initiative (GRI) was founded to support organizations by providing standards on what and how to provide environmentally relevant information. Whether a company wanted to report these sustainability reporting standards (GRI standards) has always been optional. These days, governments all around the world try to implement laws and processes for companies to report on their Environmental, Social and Governance (ESG) practices. The interpretation of the EU-parliament regarding the ESG reporting is the Corporate Sustainability Reporting Directive (CSRD). With the CSRD, the EU tries to get the company reporting on environmental, social and governmental matters up to similar level as the tax reporting. The CSRD is supposed to be reported annually as well. While the CSRD lays the groundwork, most of these legal texts are hard to understand, and usually a list of all the Information that had to be collected would be more useful to businesses than the law itself. Exactly for this matter the European Financial Reporting Advisory Group (EFRAG) was tasked with creating the European Sustainability Reporting Standards (ESRS).

The EFRAG stated that the full reporting on the CSRD would take the companies, that are required to hand them in annually, to dedicate two full time workers to that task. While there might be no clear price on environmental change, there surely is one on working hours, especially higher of a certain amount of legal knowledge is required to properly fulfill this task. What information is required from my company? Wouldn't it be great, if there was a simple solution, similar to the software used for the annual tax reporting, allowing the user to navigate through a questionnaire, ultimately resulting in a checklist that describes what standards need to be reported?

## Requirements Analysis

This is the motivation that our project is based on. We were tasked by Capgemini to build a website that allows a consultant to help a customer with their respective CSRD requirements. This would have the consultant ask for company details. These questions would be based on the CSRD, or rather the ESRS, in this case. The software will then provide a checklist-like output recapping which are the actual portions of the ESRS that have to be reported on. The questionnaire could be used to guide towards information that has to be collected from the company as parts of the requirements. Additionally, whatever kind of solution we come up with should be extensible. The law and the CSRD will change and therefore the questionnaire would require updates. These changes should not require an entire rewrite. On the other hand, it was also clarified that we are not supposed to cover the entire CSRD, but rather focus on some sections of the actual documents to build a proper prove of concept. Our solution should also not generate the actual report, as the ESRS mainly requires text-based reports, but only tell him what has to be reported on in the sections that were chosen.

### Themes / Epics

Due to the limited duration of the implementation period and the small number of sprints, there were no real Epics or Themes defined in our project. Based on the “divide and conquer” principle, we preferred to have no large tasks and rather split everything into smaller subtasks (User Stories in this case), which also allowed us to properly split the work between our team members throughout the duration of the project. If an Epic had to be defined, it would most likely be building the application as a whole, the finishing of which was a team effort.

### User Stories / Tasks

We created and implemented the following user stories. Details about the respective stories can be found in the appendix.

ID	Task/User Story
#2	Task: Create Requirements Submission
#3	Task: Create and Submit Sprint Documentation (Sprint 1)
#4	Task: Create and Submit Sprint Documentation (Sprint 2)
#5	Task: Create and Submit Sprint Documentation (Sprint 3)
#6	Task: Create and Submit Sprint Documentation (Sprint 4)
#7	Task: Create and Submit Progress Presentation
#8	Task: Create Final Presentation
#9	Task: Create Project Documentation
#10	Task: Final Project Submission / Customer Handover
#11	User Story: Project Scaffolding

#12	User Story: Setup CI/CD Pipelines
#13	User Story: Understand the CSRD Problem Domain
#16	User Story: Create a Shared Postman Collection
#17	User Story: Form Engine Core
#18	User Story: Develop Initial Set of Form Engine Controls
#19	User Story: Questionnaire Page
#20	User Story: PoC Questionnaire Form Schema
#21	User Story: CSRD Form Schema Implementation
#22	User Story: Create the Form Schema API
#23	User Story: Create the Form Schema Results API
#24	User Story: Integrate the PoC Questionnaire Into the Form Engine / Questionnaire Page
#25	User Story: Connect Questionnaire Page to Backend
#26	User Story: Questionnaire/Customer Management Page
#27	User Story: Home/Landing Page
#28	User Story: Create the Customer API
#29	Task: Customer Meeting (Sprint 1)
#30	Task: Customer Meeting (Sprint 2)
#31	Task: Customer Meeting (Sprint 3)
#32	Task: Customer Meeting (Sprint 4)
#33	Task: Internal Alignment Meeting(s) (Sprint 1)
#34	Task: Internal Alignment Meeting(s) (Sprint 2)
#35	Task: Internal Alignment Meeting(s) (Sprint 3)
#36	Task: Internal Alignment Meeting(s) (Sprint 4)
#37	Task: Sprint 4 Refinement
#38	User Story: Enhance Initial Set of Form Engine Controls
#39	User Story: Find/Define/Outline CSRD Sections for the Final App
#40	User Story: Determine Exact Questionnaire Structure
#51	User Story: Form Engine: Conditional Pages / Submitting



#53	User Story: Allow saving draft from FormSchemaResult and direct navigation to ResultOverview page.
#58	User Story: Visual Questionnaire Improvements
#59	User Story: Form Builder - Next Generation
#60	User Story: FormSchema Management Page
#71	User Story: CSRD Questionnaire - Result Checklist
#72	User Story: Form Schema Result Version Migration
#73	User Story: Testing/Bug-Fixing of the Final Application
#74	User Story: Testing the CSRD Questionnaire

**Table 1: User Stories**

## Approach

### Agile Roadmap

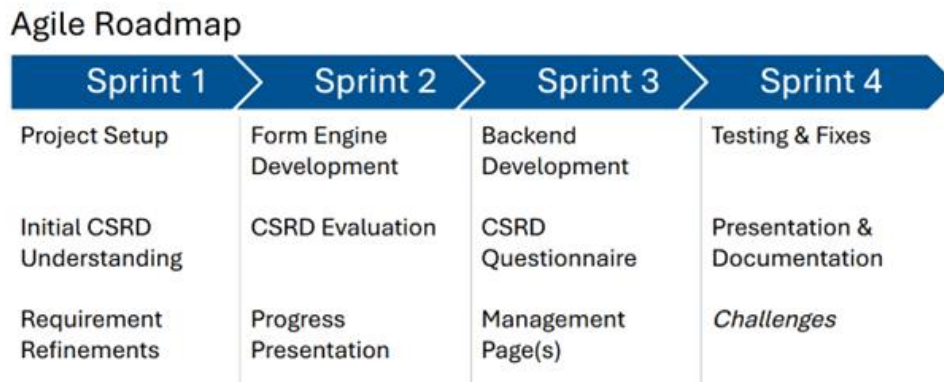


Figure 1: Agile Roadmap

We divided the work into the four sprints with the sub-goals visible in Figure 1: Agile Roadmap.

### Requirements Analysis

The Requirements Analysis phase started after the official Project Kick-Off on the 15<sup>th</sup> of May and ended with us handing in our Requirements Analysis on the 28<sup>th</sup> of May. This phase included multiple meetings with the client to specify their wishes and present our results.

ID	Task/User Story
#2	Task: Create Requirements Submission

Table 2: Requirements Analysis Stories

### *Sprint 1 - Setup & Requirements Refinement (29.05.23 – 04.06.23)*

The first Sprint was meant to lay the groundwork for our project. In this we had to get to know the legal texts and start to build the infrastructure for the application. This included starting work on backend, frontend and building an API – test library in postman. For more information on the tasks / User Stories please consult the appendix or the Sprint 1 documentation.

ID	Task/User Story
#3	Task: Create and Submit Sprint Documentation (Sprint 1)
#11	User Story: Project Scaffolding
#12	User Story: Setup CI/CD Pipelines
#13	User Story: Understand the CSRD Problem Domain
#16	User Story: Create a Shared Postman Collection
#29	Task: Customer Meeting (Sprint 1)

#33	Task: Internal Alignment Meeting(s) (Sprint 1)
-----	--

**Table 3: Sprint 1 Stories**

### ***Sprint 2 - PoC Development & Progress Presentation (05.06.23 – 18.06.23)***

The second Sprint used the groundwork to build a basic application on top. The goal was to develop a proof of concept to demonstrate at the Progress Presentation. The idea was to show off the basic functionality of the form engine and to demonstrate how it could be used to guide users through the CSRD. For more information on the tasks/user Stories, please consult the appendix or the Sprint 2 documentation.

ID	Task/User Story
#4	Task: Create and Submit Sprint Documentation (Sprint 2)
#7	Task: Create and Submit Progress Presentation
#17	User Story: Form Engine Core
#18	User Story: Develop Initial Set of Form Engine Controls
#19	User Story: Questionnaire Page
#20	User Story: PoC Questionnaire Form Schema
#24	User Story: Integrate the PoC Questionnaire into the Form Engine / Questionnaire Page
#30	Task: Customer Meeting (Sprint 2)
#34	Task: Internal Alignment Meeting(s) (Sprint 2)
#39	User Story: Find/Define/Outline CSRD Sections for the Final App

**Table 4: Sprint 2 Stories**

### ***Sprint 3 - Finalizing the Core Product (19.06.23 – 02.07.23)***

The goal of the third Sprint was to finalize the basic functionality of the application. This includes adding features and building the connection to the ABAP backend to store the questionnaire and submit the results. In this Sprint we also defined the sections of the ESRS we use and finalized their questionnaire pages. For more information on the tasks/user stories, please consult the appendix or the Sprint 3 documentation.

ID	Task/User Story
#5	Task: Create and Submit Sprint Documentation (Sprint 3)
#21	User Story: CSRD Form Schema Implementation
#22	User Story: Create the Form Schema API

#23	User Story: Create the Form Schema Results API
#25	User Story: Connect Questionnaire Page to Backend
#26	User Story: Questionnaire/Customer Management Page
#27	User Story: Home/Landing Page
#28	User Story: Create the Customer API
#31	Task: Customer Meeting (Sprint 3)
#35	Task: Internal Alignment Meeting(s) (Sprint 3)
#38	User Story: Enhance Initial Set of Form Engine Controls
#40	User Story: Determine Exact Questionnaire Structure
#51	User Story: Form Engine: Conditional Pages / Submitting

**Table 5: Sprint 3 Stories**

#### ***Sprint 4 - Fixes / Challenges / Submission (02.07.23 – 20.07.23)***

The fourth sprint was planned as a challenges sprint. Since we managed to finish all the tasks of previous sprints on time, we spend this sprint with building additional features like the ability to use an UI to build the questionnaire including all its features. This required a lot of testing and performance optimization. We also used this sprint to build the final checklist page. For more information on the tasks/user stories, please consult the appendix or the Sprint 4 documentation.

ID	Task/User Story
#6	Task: Create and Submit Sprint Documentation (Sprint 4)
#8	Task: Create Final Presentation
#9	Task: Create Project Documentation
#10	Task: Final Project Submission / Customer Handover
#32	Task: Customer Meeting (Sprint 4)
#36	Task: Internal Alignment Meeting(s) (Sprint 4)
#37	Task: Sprint 4 Refinement
#53	User Story: Allow saving draft from FormSchemaResult and direct navigation to ResultOverview page.
#58	User Story: Visual Questionnaire Improvements
#59	User Story: Form Builder - Next Generation

#60	User Story: FormSchema Management Page
#71	User Story: CSRD Questionnaire - Result Checklist
#72	User Story: Form Schema Result Version Migration
#73	User Story: Testing/Bug-Fixing of the Final Application
#74	User Story: Testing the CSRD Questionnaire

**Table 6: Sprint 4 Stories**

## Software Development

We used GitHub for managing our frontend code base. The backend was developed on the UCC's infrastructure.

## Testing Methodology

Our entire project was tested manually. No automated tests were introduced. Manual tests have been conducted after the completion of every large feature set to discover and track inconsistencies and bugs within the respective product increment.

There were generally two categories of manual tests that were frequently done: UI / E2E tests and isolated tests of the OData services. UI tests, as the name suggests, were done by navigating through application via a browser. For the backend tests, we were using a shared Postman collection to query the various endpoints.

While they were not running any tests in the traditional sense, we did leverage GitHub Actions, i.e., CI/CD pipelines, to continuously verify that our UI5 project builds and fulfills some self-imposed coding standards. Specifically, the pipeline, on every push/pull request on the main branch, verifies that the project builds, that it contains no TypeScript warnings or errors, that it produces no errors discovered by ESLint (<https://eslint.org>) and that all code is formatted using the standards imposed by the prettier (<https://prettier.io>) project. For the linting step, we followed a standard set of JavaScript/TypeScript best practices.

These pipelines, in combination with the proper usage of TypeScript did, to a certain degree, lower the requirement for other kinds of automated tests as the proper usage of TypeScript eliminates an entire class of potential errors at compile time. Anecdotally, the major bugs that we *did* find occurred due to misunderstandings in user flows or in places where externally provided types, typically those from UI5, were lacking.

# Software Architecture and Data Models

## Software Architecture

Architecturally, we had to solve one hard issue at the very beginning of the project:

*How do we render a questionnaire UI with **dynamic capabilities** (producing results usable by code, supporting dynamically showing/hiding elements, supporting input validation, etc.) that is **extensible** in the future without having to rewrite parts of the application?*

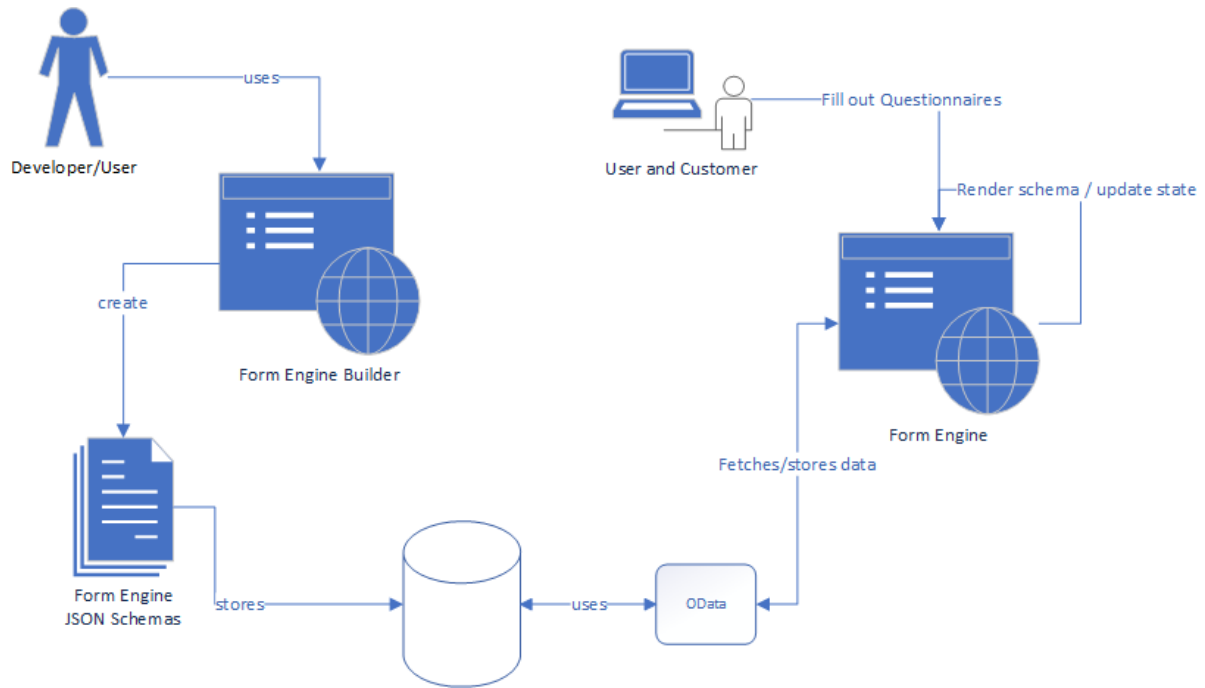
The naïve approach to the task at hand would have been to simply hard code the entire questionnaire in a UI5 view. This was not a solution to us though, because it would violate the second constraint: Extensibility. Assume, for example, that the entire application was hard coded and then the EU decides to release an update of the CSRD. This would result in large parts of the application having to be rewritten from scratch. In addition, you would always require developers for pushing updates to the application. This would not have been a good solution. We needed to find an approach that allowed semi-technical people to update the questionnaire, without any updates to the deployed application.

As typical in software development, the answer is *abstraction*. From a top-level perspective, the questionnaire is a UI that is rendered based on *some input data* (a descriptor of the UI) that also produces *some output data* (the results entered by the user). This is a very powerful view on such a UI, because it means that one only needs three components to build the app and fulfill the above requirements:

1. A data store for the descriptors of the UI.
2. A data store for the results produced by users.
3. A frontend rendering algorithm that translates the UI descriptors into actual UI elements.

We called our implementation of (3) *form engine*. The form engine is a frontend component that receives a structured JSON object, a *form schema* (1), translates it into a browser UI, allows the user to interact with that UI and finally, when the user submits the questionnaire, stores the user's results as another JSON object, the *form schema result* (2).

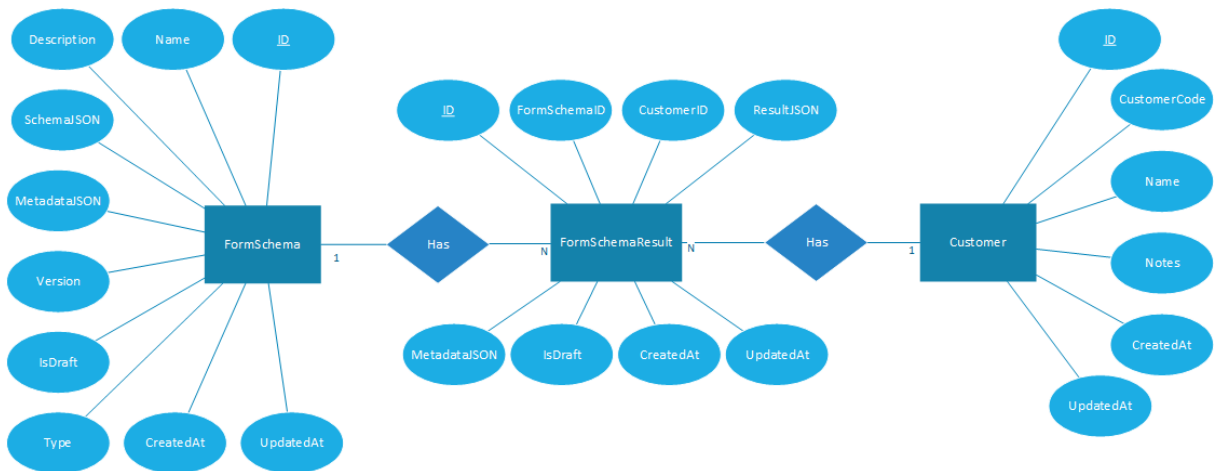
With this foundation settled, the application is practically architected. What remains is an extension with some assistive components and data structures that enable end-to-end user flows and improve the user experience. Figure 2: Application Architecture gives an overview of how all final components play together. At the core stands the form engine. It is a component inside the browser, rendering the questionnaire for a user (which is, due to the requirements, called a *customer*). It renders prebuilt form schemas that are dynamically retrieved from a data store via an OData service and saves the form schema results that the user entered via the very same OData services. The form schemas themselves must also be built – this is made possible via another UI, the form builder, a WYSIWYG editor that allows both developers, but also normal users of the application, to build questionnaires. Finished questionnaires are stored inside the backend, ready to be fetched by the form engine.



**Figure 2: Application Architecture**

An interesting aspect of the above architecture is that it creates a completely generic application which could, theoretically, be used to capture data via questionnaires of any kind. Our scenario will theme the application towards the CSRD. The development of a CSRD form schema is part of the development process.

## Semantic Data Model



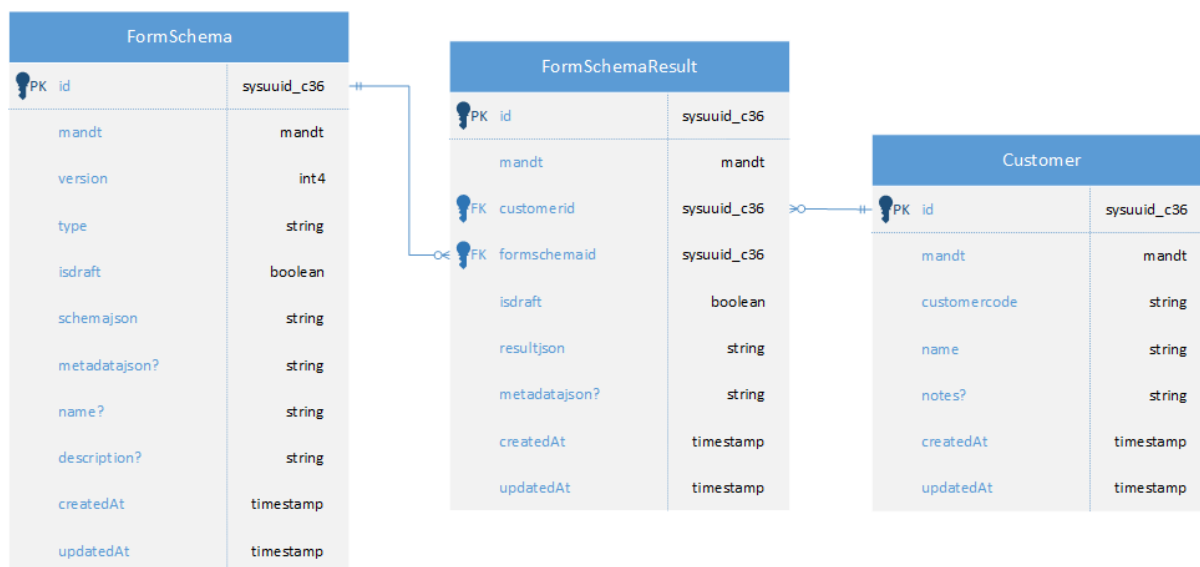
**Figure 3: Semantic Data Model**

As described in Software Architecture, our app only needs two core entities to function: The form schemas and the form schema results. We did introduce an additional, assistive entity: the customer. The relation between those entities can be seen in Figure 3: Semantic Data Model. Both the form schema and the form schema results hold their most important attributes, i.e., the JSON documents containing the schema and results, as simple attributes. In addition, they possess an attribute called `MetadataJSON` which was initially introduced to allow the frontend to store arbitrary, unstructured data. Effectively, this was never used by the application.

Form schemas also possess a `Version` and an `IsDraft` attribute. These two address the complexity of questionnaire updates. For example, assume that a user fills a CSRD questionnaire at version 1 and submits the results. Then, a developer publishes an update (version 2) of that questionnaire, with new pages and new data points to be entered. The user's results are now, technically, obsolete. When the user now wants to view his previously entered data, the app must be able to render the questionnaire of version 1, i.e., the version at the point in time when the user filled the questionnaire. This is only possible if updates to the form schemas are *non-destructive*. Specifically, we ensure that form schemas are generally *immutable* – once published, they are not allowed to change anymore. It is only possible to publish *new versions*, but old versions are still kept. We do allow one exception to this rule: Form schemas in a draft version. When a new schema is in the process of being developed, we expect frequent changes to it until it is finally complete. To avoid cluttering the data store, schemas of a draft form schema *are allowed to change*. Draft form schemas are always assigned a version -1 and are not visible to a user who wants to answer a questionnaire. Once undrafted, a valid version is assigned and the form schema becomes immutable.

Form schema results also have an `IsDraft` attribute, but this attribute, in contrast to form schemas, is just used for visual indicators in the UI. Form schema results are always mutable. Finally, the customer is just a normal entity without any specifics. Customers exist so that filled questionnaires can be associated with someone, from the perspective of the consultant that owns this application.

## Technical Data Model



**Figure 4: Technical Data Model (Backend)**

Figure 4: Technical Data Model (Backend) shows the technical data model for the entities defined described in Semantic Data Model. This model is nearly a 1:1 mapping of the model described above, hence it is not described in any more detail here.

The single point that should be mentioned is that the JSON blobs are stored as strings. For us, this was an easy-to-use approach that ultimately worked out well. It could be considered to use a more appropriate type – or even database layer – here, e.g., JSON columns or a NoSQL database, as those alternatives would allow native querying of this unstructured data.





for example, have an effect that only shows the element when a specific condition (i.e., an expression) evaluates to true. An example, in JSON format, can be seen here – an arbitrary element is only shown, when the value of another input is equal to Hello world!:

```
{
  "effect": "show",
  "condition": {
    "type": "eq",
    "left": {
      "type": "value",
      "id": "otherElement"
    },
    "right": "Hello world!"
  }
}
```

Expression trees are recursive and can, theoretically, be nested indefinitely. A simple example using two layers can be seen here:

```
{
  "effect": "show",
  "condition": {
    "type": "and",
    "left": {
      "type": "eq",
      "left": {
        "type": "value",
        "id": "otherElement"
      },
      "right": "Hello world!"
    },
    "right": {
      "type": "gt",
      "left": {
        "type": "value",
        "id": "numberElement"
      },
      "right": 50
    }
  }
}
```

This expression tree asserts the previous condition, but additionally verifies that the value of a number input with the ID “numberElement” has a value greater than 50.

Expression trees are used in various locations inside the form schema: Effects on elements and pages, validation rules on dynamic elements and ultimately as referenceable conditions inside the schema. The latter, also called “ref conditions”, are essentially condition variables that can be reused by other elements, thus avoiding duplication of the contained logic.

## Implementation

This chapter describes several technical implementation details of our application’s frontend and backend. It cannot serve as a replacement for reading the application’s code, as the code is ultimately the “hard truth” of *what* is implemented *how*. However, we will give an overview about cornerstones that require either an introduction or a detailed explanation to be understandable. Ideally, after reading this chapter, you are able to navigate through our codebase and understand it.

### SAPUI5 Application

When compared to other projects, our implementation might stand out when comparing the amount of frontend code to backend code. As evident from the data models, the scope of our backend is tiny when compared to the frontend. This is an advantage of the architecture that we chose: Iterating on the frontend is a much faster process when one doesn’t need to rely on the completion of associated changes in the backend. It does, however, result in the frontend needing to handle the vast amount of user flows and data management, leading to a larger and, if not kept in check, more complicated project. We anticipated this situation upfront and took appropriate measures to ensure that development within the UI5 codebase became and remained a smooth process. Those measures might deviate from the way a “traditional” UI5 application is written, but they ultimately were the reason that enabled us to deliver this project. Understanding our frontend setup and its architecture is crucial for understanding the implementation. This chapter will therefore start by explaining the project setup itself (i.e., the “why”) and only then it will progress into implementation details of interesting keystones (i.e., the “how”).

Note that this chapter assumes that you know about the application from the perspective of a user, i.e., what its UI looks like and what user flows and features it supports. If you haven’t used the application yourself yet, we recommend doing so, or alternatively, reading the User Manual first.

### *Project Setup and Requirements*

Please read this section carefully if you intend to run the frontend part of the project locally.

For running the project locally or for continuing development on it, we expect the following system configuration:

- A machine running Windows
- An evergreen version of a Chromium browser and/or Firefox
- The latest LTS Node.js version

The rationale for those requirements is as follows:

Local development happened on Windows machines only. Although it should be possible, we cannot *guarantee* that it is possible to develop the frontend project on non-Windows operating systems, due to a lack of access to those machines.

The reason for requiring an evergreen browser version is that the code makes use of semi-modern JavaScript APIs like `Array.findLastIndex`. To our knowledge, UI5 does not, out of the box, polyfill those APIs and neither do we. This makes using a modern browser a requirement. Running an outdated browser version could therefore cause instabilities within the application. Realistically, if your browser of choice is allowed to automatically update, you should be alright.

At last, a modern version of Node.js is required for two reasons: Managing the project via NPM and for running small scripts when the project is started.

Overall, the project is a normal UI5 frontend application using NPM. The project can be managed, started and deployed via the `npm run` scripts listed within the `package.json` file. We modified the default scripts of the project template to support our requirements, resulting in the following `npm run` scripts:

Script	Description
<code>start</code>	Starts the project locally on <code>localhost:8080</code> and launches a new browser window on that address. Before starting, the command auto-generates code that is required for running the application. This is the recommended way to start the project.
<code>watch</code>	Helper script for <code>start</code> . Watches for any file changes, both within
<code>watch:app</code>	Helper script for <code>watch</code> . Watches application files and issues a re-build when those change.
<code>watch:types</code>	Helper script for <code>watch</code> . Watches UI5 controls and regenerates TypeScript types when the public API of a control (i.e., properties, aggregations, etc.) change.
<code>generate</code>	Helper script for <code>start</code> and <code>build</code> . Runs the code generation process once.
<code>generate:types</code>	Helper script for <code>generate</code> . Generates TypeScript types for the public API of a control (i.e., properties, aggregations, etc.) change.
<code>generate:json-schema</code>	Helper script for <code>generate</code> . Runs the <code>./tools/generateJsonSchema.js</code> script to generate a JSON schema code fragment that is required by the application.
<code>build</code>	Builds the entire project. This includes code generation.
<code>deploy</code>	Builds and automatically deploys the project to the configured target system.
<code>undeploy</code>	Undeploys the project from the configured target system.
<code>tsc</code>	Runs the TypeScript compiler on the code base. This is used for ensuring that the code base contains no TypeScript errors.
<code>format</code>	Formats the entire project using prettier.
<code>lint</code>	Runs ESLint on the code base. This is used for ensuring that no linting errors exist in the project.
<code>prepare</code>	A lifecycle hook which is automatically executed whenever dependencies are installed. Installs git commit and push hooks to automatically format the code on commit and to automatically lint/test the code on push.  You should not run this manually.

**Table 7: NPM run Commands**

For local development, only `npm start` and `npm run build` are usually required. In special cases, `npm run generate` might need to be run when making changes to the form schema types while the dev server is running.

## TypeScript

Speaking in absolutes, when given the choice between starting a greenfield project in either JavaScript or TypeScript, the choice should never be JavaScript. There are, of course, granularities and exceptions to this statement, but in general, it applies - the errors that TypeScript catches at compile-time and the improved editor support alone typically make it worth it. There already exists plenty of discussion on the internet supporting this statement and we don't want to reiterate what is explained much better online. However, there are some specific benefits unique to our project and/or the UI5 TypeScript integration that are worth highlighting.

As already mentioned, a large portion of our application's data management flows are happening on the frontend. This means that interaction with and modification of data is done in the frontend. Modifying data correctly is crucial for maintaining its integrity and having compile-time error discovery via types assists tremendously in avoiding them. Take, for example, the form schema, which was already introduced in Software Architecture and Data Models – the form schema is crucial for correctly rendering the UI of a questionnaire. Processing the form schema is handled entirely on the client-side. Since the backend is not doing any validation of the schema, the frontend itself is responsible for ensuring that the schema is both built and processed appropriately. Types and compile-time error checking is a tremendous benefit here, discovering issues, typos and potential logical errors in mere seconds, whereas a plain JavaScript implementation would require thorough runtime testing.

Another benefit is the dramatic increase of development speed. Especially in the context of UI5, several APIs are not necessarily intuitive to developers new to the framework. Having auto-complete and type suggestions by the editor enabled us to iterate on the project with a higher velocity, as mental context switches, e.g., due the need to check online documentation, became less of a necessity.

At last, the adoption of TypeScript has practical benefits on the density and readability of code. A prime example for this is how module imports are handled in a UI5 JavaScript vs. a UI5 TypeScript project. At its core, UI5 uses a non-standard AMD API for loading modules. A typical example in a JavaScript project looks like this:

```
sap.ui.define(
  ['MyDependency1', 'MyDependency2'],
  function (MyDependency1, MyDependency2) {
    // Module code.

    return {
      // ...exported members.
    };
  }
)
```

This API is verbose and error prone (error prone, because it requires the user to manually align the order of imports in the dependencies array and the module function). Further, without additional plugins, editors like VS Code do not understand this syntax and cannot help the developer with features like auto-completion. Compare this with the TypeScript/ESM equivalent:

```
import MyDependency from 'MyDependency';
import MyDependency2 from 'MyDependency2';
```

```
// Module code.

export default {
  // ...exported members.
}
```

This syntax is much clearer to read, is supported by editors and, most importantly, is aligned with the established ESM standard of the modern web. With the help of Babel, this syntax is transformed to UI5's module API during compile time, thus giving us the best out of both worlds: Perfect interoperability with UI5 and a modernized, editor-supported coding environment.

Using TypeScript with UI5 requires some initial setup. UI5 itself is written in plain JavaScript. This means that, by default, there is no type support for UI5's API surface. Further, the above mentioned transpilation of TypeScript features like ESM imports to UI5's dialect is not in-built into the TypeScript compiler. This means that, in order for TypeScript to work properly with UI5, additional steps apart from installing the TypeScript compiler and creating a `tsconfig.json` file must be taken. Fortunately, SAP, by now, provides first-class support in the form of multiple type definition NPM packages and a Babel addon which handles the aforementioned transpilation process. For additional information about TypeScript support in UI5, you can have a look at the following link: <https://sap.github.io/ui5-typescript/>

In our case, we integrated TypeScript as a three-staged process. First, we installed a UI5 type definition package to inform TypeScript about UI5's API surface. Here, you can effectively choose between the following two packages:

- `@sapui5/types` – ESM type definitions
- `@sapui/ts-types` – legacy type definitions

In our case, we used `@sapui5/types`. Using legacy types for a new project wouldn't have made much sense.

As the second step, we installed the Babel addon via the `ui5-tooling-transpile` package. This package can be added to the UI5 CLI pipeline by adding a new middleware and builder step to the `ui5.yaml` configuration.

At last, we installed the `@ui5/ts-interface-generator` package which is a necessity when your project contains custom controls. Control definitions/classes in a UI5 TypeScript project, by default, clash with the TypeScript compiler due to how UI5 declares the types of metadata, like properties. This package remedies that by providing a CLI that automatically parses the source code of custom controls and generates type definitions for those controls that satisfy the TypeScript compiler. It even provides a watch mode, meaning that it updates the generated code on the fly when changes are made.

With the above setup, the project properly compiled the TypeScript code. We additionally enhanced the error-discovery process during development by installing ESLint addons specifically suited for TypeScript development which further reduced the possibilities of introducing errors.

## ***Code Generation***

The previous section already touched on how some parts, specifically, type definitions for our controls, are automatically generated. We further introduced another code generation layer for solving a very specific problem: The lack of reflection in JavaScript.

To understand the issue at hand, one must familiarize oneself with the form schema and the form builder feature. The form builder requires an in-depth understanding of the form schema, because it is responsible for generating a valid schema on behalf of the user. To do so, it is crucial that the UI knows about the granularities of the form schema. Take, for example, the “Add Element” dialog which is shown when a user wants to add a new element to an existing form schema page. The dialog needs to show a list of all elements that are supported by the form schema. There are generally two ways to provide such a list in code: (1) the list can be hard-coded and (2) the list is automatically generated. For most of our features, we use approach (1), together with smart type declarations. If the schema changes, the TypeScript compiler notifies us that we must also update parts of the UI. For one specific code location, the amount of code that we would have had to hard-code became too much though. Therefore, we needed to tackle approach (2), i.e., auto-generate parts of the UI by analyzing the form schema. This is a difficult task, because JavaScript does not provide any native type information. Further, the form schema definition itself is written in pure TypeScript interfaces, meaning that all information related to it are completely erased when the application is compiled. We needed a workaround: code generation. With the help of the `ts-json-schema-generator` package, it is possible to generate a JSON schema from TypeScript types. JSON schema is a standard that describes the shape of a JavaScript object. It is typically used for verifying that an object matches an expected shape, i.e., for validation, but it can also be used for defining the metadata that we are looking for: The JSON schema matching our form schema types contains a list of all interfaces and attributes that the form schema supports. It is therefore a good replacement for reflection in languages like C# or Java, because it provides us with type-based metadata that we can leverage to dynamically generate parts of the UI.

We are invoking the code generation once whenever the project is started or built. The process is handled by a Node.js script located in `./tools/generateJsonSchema.js`.

## ***State Management***

Every UI is a visual representation of an internal state (e.g., a “busy” state is mapped to a loading spinner). This makes the management of the UI’s state the primary concern when writing an application. In the modern frontend toolbox, this led to a plethora of state management libraries like Redux, MobX, Zustand and many others. State management is no longer treated as a side activity, but instead as the central concern of a frontend application.

While UI5 does not provide any opinionated approach for state management, the framework does promote the usage of the MVC pattern. In the MVC pattern, the model is typically responsible for managing the data and business logic of the application – a role that is, roughly, equal to the management of application/UI state. UI5 also supports first class data binding, making the development of views a much more streamlined process if the application’s data is kept inside a UI5 model.

In many CRUD applications, proper state management can be discarded by moving the vast amount of logic to a backend server. This is not the case for our project though – most logic lives in the frontend. This makes proper state management incredibly important for the project’s success. We therefore decided early on to use a state-focused approach for writing our application. When the project began, we thought about whether we wanted to use an off-the-shelf state management library like Redux, but in the end, we decided against that. First, we didn’t know whether it was allowed within the scope of the practical course – we therefore decided that we wanted to build the application in a “pure way”, i.e., by only using the core UI5 libraries.

Second, even if we were to use a 3<sup>rd</sup> party library, we'd have to write integration code anyway (Redux, for example, doesn't allow native data binding to UI5 due to a lack of official bridge packages). Ultimately, this resulted in us building a tiny and yet very effective state management layer that is built on UI5's native toolbox. The implementation is heavily inspired by the `zustand` library, whose documentation (which is accessible at <https://github.com/pmndrs/zustand>) is a recommended read from our side – if you understand `zustand`, you will also understand our state implementation.

The vast majority of our frontend's code base is based upon this implementation. In order to understand our project, it is important to understand how our state management solution works. Let's consider an example: A UI that displays a counter and two buttons for incrementing and decrementing the counter:

```
<Text text="Count: {/count}" />
<Button text="Increment" press="onIncrement" />
<Button text="Decrement" press="onDecrement" />
```

The button's events are hooked to the controller and the counter text is data bound to an arbitrary model's `/count` path. The presence of this model is already the introduction of state management: Instead of tracking the current count inside the controller and imperatively updating the UI whenever a button is pressed, the data is stored inside a model – a state container that automatically updates the UI when its held data changes. This is where it gets interesting – this model is not simply defined as a new JSON model, but instead uses our custom state management abstraction:

```
interface CounterState {
  count: number;
}

const counterState = createState<CounterState>(() => ({
  count: 0,
}));
```

In this code fragment, we are defining the shape of the state to be managed (which is only the current count value) and then create a new `counterState` via the `createState` function. This is the state container - an object, that provides several APIs for interacting with and modifying the state:

```
// Get the current value:
console.log(counterState.get().count); // 0

// Set the current value and log the updated value:
console.log(counterState.set({ count: 123 }).count); // 123

// Subscribe to *any* change of the state:
counterState.subscribe((next, previous) => {
  console.log(`The entire state changed from
    ${JSON.stringify(previous)} to
    ${JSON.stringify(next)}.`);
});

// Subscribe to *specific* changes of the state (in this case, count):
counterState.watch(
  (state) => state.count,
  (next, previous) => {
    console.log(`The count changed from ${previous?.count} to
```



```

        ${next.count}.\`);
    },
);

```

The state also provides one very important field: `model`. `model` is a UI5 `JSONModel` instance whose value is updated whenever a batched state change occurred. This means that the contents of the state container are eventually reflected into a UI5 model to which the view can bind (the state also supports two-way binding, meaning that changes to the model are reflected back into the state). In the above example, this means that the view fragment above can be made functional via the following controller:

```

class DemoController extends Controller {
    override onInit(): void {
        this.getView()?.setModel(counterState.model);
    }

    onIncrement(): void {
        counterState.set((state) => ({ count: state.count + 1 }));
    }

    onDecrement(): void {
        counterState.set((state) => ({ count: state.count - 1 }));
    }
}

```

With this controller in place, the user is able to interact with the UI. The implementation is not perfect though – we are still leaking details of the state management, namely, changing the count, into the controller. A perfect implementation would instead look like this:

```

interface CounterState {
    count: number;
    increment(): void;
    decrement(): void;
}

const counterState = createState<CounterState>(({ set }) => ({
    count: 0,
    increment: () => set(({ count }) => ({ count: count + 1 })),
    decrement: () => set(({ count }) => ({ count: count - 1 })),
}));

class DemoController extends Controller {
    override onInit(): void {
        this.getView()?.setModel(counterState.model);
    }

    onIncrement(): void {
        const { increment } = counterState.get();
        increment();
    }

    onDecrement(): void {
        const { decrement } = counterState.get();
        decrement();
    }
}

```

The difference here is that the “business logic”, i.e., the modification of the state, is handled *inside* the state itself. This transforms the purpose of the controller from a class that knows about the details of the state (i.e., “I need to update the count from the previous value to the next increment”) to a simple dispatcher between the UI and model (i.e., “the increment button was clicked, therefore I must forward this event to the state management”).

In the complex areas of our application (e.g., the form builder), we are making heavy use of this pattern, where the entirety of the business logic and UI data is moved inside a state container. This results in an interesting side-effect: The core logic of the application is, for the most part, decoupled from the underlying UI5 framework. In theory, one could, with comparatively little amount of work, move away from UI5 to another UI layer while keeping most of the state management code the same. This shows the true power of the MVC pattern – an isolated model implementation leads to cleanly decoupled layers. For the sake of completeness, we should mention that we *do not* strictly follow this pattern everywhere: For the application’s CRUD-areas, e.g., the questionnaire management, the controllers also handle parts of the state management. Finalizing the app was faster that way and there was little benefit to be had from fully embracing the pattern everywhere.

### State Slices

A powerful aspect of the state implementation is that the state is just a simple JavaScript object. This enables composability of a state container via slices. Slices are a subset of a state container’s state that can be independently managed in separate files, thus reducing the cognitive complexity required for understanding the code.

The concept is easier to understand with an example. Assume that an application consists of two views which are overall different but share one common control: A text that displays a message. This message text shares the same state implementation in the two views. The naïve approach for implementing the states of those views would be to simply copy and paste the message into the two corresponding states:

```
interface View1State {
  message: string;
  updateMessage(): void;
  // Other properties, unique to view 1.
}

interface View2State {
  message: string;
  updateMessage(): void;
  // Other properties, unique to view 2.
}

const view1State = createState<View1State>(() => ({
  message: 'Hello World',
  updateMessage: () => { /* Code to update message */ },
  // Other properties, unique to view 1.
}));

const view2State = createState<View2State>(() => ({
  message: 'Hello World',
  updateMessage: () => { /* Code to update message */ },
  // Other properties, unique to view 2.
}));
```

This is not a good implementation - code duplication should generally be avoided. A better approach is to extract the shared parts of the state into a slice:

```

interface MessageSlice {
  message: string;
  updateMessage(): void;
}

interface View1State extends MessageSlice {
  // Other properties, unique to view 1.
}

interface View2State extends MessageSlice {
  // Other properties, unique to view 2.
}

const createMessageSlice = (state) => ({
  message: 'Hello World',
  updateMessage: () => { /* Code to update message */ },
});

const view1State = createState<View1State>(({ state }) => ({
  ...createMessageSlice(state),
  // Other properties, unique to view 1.
}));

const view2State = createState<View2State>(({ state }) => ({
  ...createMessageSlice(state),
  // Other properties, unique to view 2.
}));

```

This way, the shared part is managed in a single location and can easily be modified and understood. Note that a slice typically receives the `state` container as a parameter, thus being granted full access to the state API.

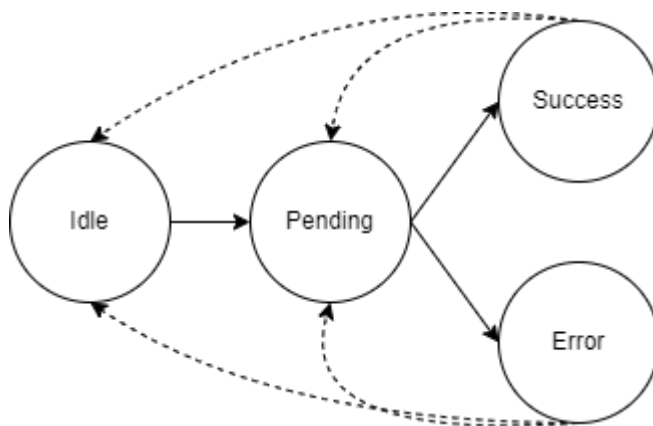
In our project, we make use of slices in two locations: (1) the form engine context and (2) the form builder. (1) is a more complicated version of the above example: Our app has multiple views which render a form engine. Each of those views needs to maintain the engine's state. Therefore, we needed to extract the state management of the form engine into a separate slice. (2), i.e., the form builder, uses slices for a different reason: To manage complexity. The form builder's state is large and complicated. Maintaining it in a single file would make the code incredibly hard to understand. Therefore, we split it into multiple, semi-independent slices (and files) which, by themselves, are smaller and easier to understand/maintain.

### *Asynchronous, Server-Side State*

When building a UI, you sooner or later need to deal with asynchronous states. A prime example is server-side state, e.g., data that is first loaded from a server, processed locally by the user and then updated on the server. Asynchronous state behaves differently than synchronous state, because suddenly a delay must be considered, e.g., the connection time to/from the server. In addition, since the remote parties are typically other machines, most asynchronous operations can fail, e.g., due to a drop of the user's network connection. In general though, every asynchronous operation can be reduced to four possible states, as visible in Figure 6: The States of An Asynchronous Operation:

- *Idle*: The operation has not been started yet.
- *Pending*: The operation is running (e.g., fetching data from a server).
- *Success*: The operation succeeded (e.g., the server returned data).

- *Error*: The operation failed (e.g., the server replied with an error).



**Figure 6: The States of An Asynchronous Operation**

UIs, and consequently the UI’s state implementation, need to handle each of those states when dealing with asynchronous operations. This can also be used to enrich the user experience, e.g., by showing loading spinners while an operation is pending or detailed error information when an operation failed.

Our state implementation provides an easy-to-use abstraction for two kinds of asynchronous operations: (1) Querying data and (2) Mutating data. The implementation is heavily inspired by two popular libraries from the React ecosystem: `react-query` and `swr`. From a top-level perspective, both queries and mutations represent the four states of an asynchronous operation. They provide information about the operation’s current state (e.g., an `isPending` flag which allows the UI to display a busy indicator via data binding) and access to either the result or the error. The difference between a query and a mutation is that a query automatically starts in the *pending* state (data should be loaded as soon as possible), whereas a mutation starts in the *idle* state, i.e., it must be explicitly triggered (for example, via a user interaction like a click on a “Delete” button). The following code fragment shows an example of how our state implementation can be used to fetch data from a server via a query and delete an entity from a server via a mutation:

```

interface AsyncOpState {
  // The first type parameter is an argument that the fetch function receives.
  // The second type parameter is the return value of the fetch function.
  query: QueryState<string, ApiEntity[]>;
  mutation: AsyncState<string, ApiEntity>;
}

const state = createState<AsyncOpState>(({ state }) => ({
  // Important: `key` must always be set to the name
  // of the query/mutation property within the state.
  query: createQuery({
    state,
    key: 'query',
    getArgs: () => '/api/entities',
    fetch: async (url: string) => await apiGet(url),
    onSuccess: (entities) => console.log(entities),
    onError: (err) => console.error(err),
  }),
  mutation: createAsync({
    state,
    key: 'mutation',
  })
})

```

```

    fetch: async (id: string) => await apiDelete(`/api/entities/${id}`),
    onSuccess: (entity) => console.log(entity),
    onError: (err) => console.error(err),
  }),
});
});

```

The above code is all that we require for representing asynchronous operations. Queries have a special function `getArgs` which provide the required arguments for starting the query. This function is required because queries are started automatically – the API therefore needs a way to dynamically get the query’s current arguments. If the function returns `undefined` or `null`, the query does not start. This can be used to pause queries if a certain precondition is not met. Overall, both queries and mutations provide, among others, the following state properties to which other parties, including the UI5 view, can use for data binding and custom logic:

```

const { query } = state.get();
console.log(query.isIdle);      // true if the operation has not yet started.
console.log(query.isPending);  // true if the operation is running.
console.log(query.isError);    // true if the operation failed.
console.log(query.isSuccess);  // true if the operation succeeded.
console.log(query.data);       // The fetched data, if any.
console.log(query.error);      // The error, if any.

```

Queries can be restarted/renewed via `query.fetch()`. Mutations can be invoked via `mutation.fetch(args)`, where `args` is an arbitrary argument required by the mutation (e.g., the ID of an entity to be deleted).

In our project, we make heavy use of queries and mutations whenever we are using server-side operations, i.e., OData APIs. There is one exception: If we simply need to display a list of entities (e.g., in the customers list on the customer management page), we use a standard UI5 OData list binding. Though whenever we require custom or complex processing of the fetched data, we pull this data into our state containers via queries.

### *Routing State*

Our views make heavy use of routing parameters. Take, for example, the URL of the page that is displayed when an existing questionnaire is edited by a given customer: `#/questionnaire/csr/customer/457D655F-65DC-1EDE-8685-5A9075CB548D/?formSchemaResultId=457D655F-65DC-1EEE-88E6-398533075912`

This URL contains two route parameters: the `customerId` and the `formSchemaType`. It also leverages an optional query parameter for determining whether an existing form schema result should be edited: `formSchemaResultId`. In the manifest, this is encoded as the following route pattern: `questionnaire/{formSchemaType}/customer/{customerId}/: ?query:` From these three pieces of information, the page is able to determine and load the entire data that is required for functioning. A user could, therefore, bookmark the page, do a new navigation to it and it would load correctly.

These pieces of information, i.e., the URL parameters, must therefore be considered part of the application’s state, i.e., merged into the state container. We did this via a special router state slice:

```

interface RouterStateDemo extends RouterState<
  { customerId: string; formSchemaType: string },
  { formSchemaResultId: string }

```

```

> {
  logState(): void;
}

const routerState = createState<RouterStateDemo>(({ get }) => ({
  parameters: {},
  query: {},
  logState: () =>
    console.log('Parameters and query: ', get().parameters, get().query),
}));

// Then, inside a controller's onInit function:
connectRouterState(routerState, this.getRouter());

```

This is a special slice, because it needs to be hooked to the UI's router manually via the `connectRouterState` function. This is a requirement, because for updating the state, one needs to attach to the router's events to be notified about route changes. Whenever a route change occurs, the handlers attached in `connectRouterState` properly update the state's `parameters` and `query` fields with all values present in the URLs (query) parameters.

At this point, we should also mention that, whenever the user navigates away from a page (e.g., from the customer page to the home page), the previous page's state is *reset*. This addresses a specific behavior in UI5: The fact that controllers are objects with a singleton lifetime *without an ability or option to change this*. Once a controller is initialized, the very same instance is reused until the application exits completely. This leads to behavior that is consistent with most other pages on the web. Typically, when you navigate back and forth between pages, you expect those pages to start from a clean state. We wanted this behavior for our application, too. Therefore, our `BaseController` resets the states via the `reset()` function whenever a user navigates away from a page.

## Project Structure

With the above foundations settled, it should already be possible to understand most parts of the frontend's codebase. What remains for the general area is an overview of the project structure, i.e., where which type of files are located. In principle, we are using a normal UI5 project structure. We use the typical `webapp`, `controller`, `css`, `il8n`, and `view` folders. In addition, we introduced the following folders inside `webapp`:

- `api`: Contains functions for calling the app's associated OData services. These functions make use of an OData model, but provide a modernized, Promise-based API. We created one function per CRUD operation per OData entity set. Whenever an API call is made, we use the functions inside this folder.
- `assets`: Contains images that are used by the app, i.e., static assets.
- `controls`: The folder for custom controls that don't fit into any specific feature folder.
- `formengine`: A feature folder, containing all code files that are required for rendering the form engine.
- `state`: Contains the state containers that are used by the views/controllers. Per controller, we have at least one associated state container file inside this folder.
- `utils`: Contains utility functions and members of arbitrary sizes and scopes.

## Form Engine

Conceptually, the form engine was already covered in Software Architecture and Data Models. The technical implementation deserves its own section though, because it deviates from a “traditional” UI5 view implementation. From the perspective of a developer, the form engine needs to fulfill the following tasks:

1. It needs to be reusable by multiple different views.
2. It needs to render the UI, based on a JSON schema.
3. It needs to produce a JSON output based on the user’s actions.

This list of requirements results in a single, feasible implementation option: A custom UI5 control. Controls can be (re-)used by multiple views and support properties which allow data binding (and thus passing data in and out). This custom `FormEngine` control is located in the `webapp/formengine/FormEngine.ts` file. The control itself contains relatively little code – less than 100 lines. This is because it is, comparatively, a “dumb” control. In essence, it only performs the following tasks: It defines the UI5 control metadata and properties and calls an external `render` function whenever it determines that it must re-render the UI. This means that the control **does not** handle tasks like page management, input validation, conditional evaluation of expression trees, submission of the form, etc. Those tasks are handled externally via a state slice, specifically, a `FormEngineContext`.

The `FormEngineContext` is a state slice that can be integrated into an arbitrary view’s state container, e.g., the form builder’s or the questionnaire page’s state container. Moving the management of the form engine’s internal from the control into a state slice is what enables those containers to perform actions or display data based on the form engine’s internals, e.g., modifying the form schema in the case of the form builder or showing validation errors when “Submit” is pressed. The consequence is that the `FormEngine` control becomes a stateless control – its only purpose and task is to receive its current state (i.e., the context) from the outside and render that state whenever it changes.

Rendering the UI is a relatively straightforward process: The `FormEngine` control maintains a single `VBox` control, a container for the elements to be rendered on the current page. Whenever it determines that it must re-render the current page, all current elements are removed from the container, the new elements are created and added to the container. Finally, the container is re-rendered by UI5. The rendering of the current page, i.e., the translation of the elements in the current page’s form schema to the corresponding UI5 controls, happens in `Rendering.ts`. We maintain one rendering function per form schema element type. In addition, we extracted several shared properties like `required` or `description` into shared wrapper rendering functions. This file is also the file that performs *validation* and the *evaluation of the expression trees*, albeit via helper functions located in separate files.

### **Form Builder**

The form builder is a complex UI with many different functionalities and behaviors. Explaining it in detail here would exceed the scope of the documentation. Despite this, there are a few points that are worth mentioning:

Due to its size, the form builder is split into various feature slices. We chose a UI-driven split. As visible in Figure 7: Form Builder Feature Areas, we split independent areas of the UI into feature areas. Each area is represented as its own view (e.g., `FormBuilderPageArea.xml`), controller (e.g., `FormBuilderPageArea.controller.ts`) and state slice (e.g., `FormBuilderPageArea.ts`). Because UI5 does not support properties on views, this separation is only possible with a shared, global state variable. This means that all

those different views operate together as a single entity – because they all share the same, underlying model.

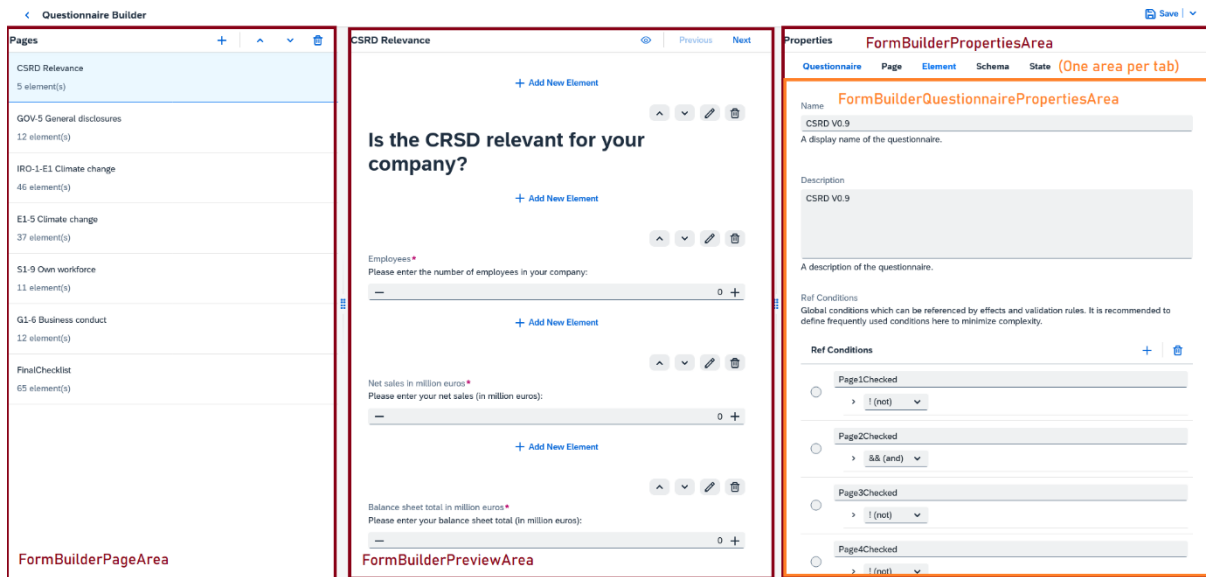


Figure 7: Form Builder Feature Areas

Another interesting feature of the form builder is the expression tree editor. An example is the control seen in Figure 8: Expression Tree Editor, which shows the expression tree editor within an element effect. The expression tree editors are complex UI fragments that are reused in four different locations. Therefore, we again needed to create a custom UI5 control. This control lives in `webapp/controls/ExpressionTreeEditor.ts`. It is responsible for dynamically generating the tree element based on an expression tree defined in a form schema. Unlike the `FormEngine` control, the `ExpressionTreeEditor` is self-contained, meaning that it handles its entire lifecycle by itself, without the assistance of any *external* state management features.

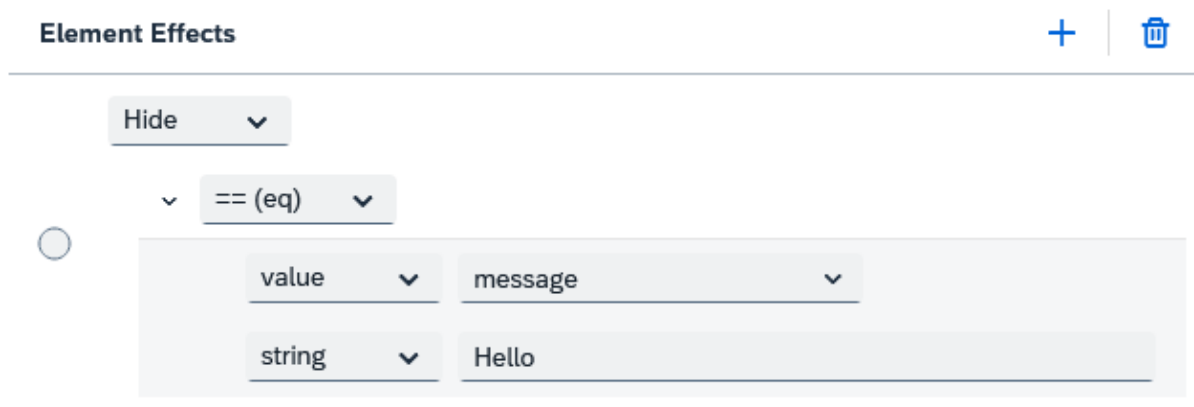


Figure 8: Expression Tree Editor

At last, the form builder is the feature that depends on the JSON schema code generation that was mentioned in Code Generation. The file that makes use of the generated JSON schema is located in `webapp/state/FormBuilderElementPropertiesArea.ts`.

## ABAP Code



Our backend, specifically, our OData services, have been implemented in ABAP. The OData entities are 1:1 representations of the data model(s) described in Software Architecture and Data Models. Per entity, we provide 5 CRUD endpoints: GET All, GET by Id, POST, PUT/PATCH and DELETE. The endpoint implementations are, for the most part, nothing special – they perform their task by running appropriate queries on the associated database table(s). Some endpoints do have special behaviors though. For a detailed, per-endpoint description of this behavior, consult the following tables:

<b>FormSchema</b>	
<b>Operation</b>	<b>Behavior</b>
GET All	Does not return the <code>SchemaJson</code> and <code>MetadataJson</code> fields. This is done for performance – returning multiple entities with large string fields would consume a lot of bandwidth. If the UI needs to access those fields, they can still be retrieved via the GET by Id operation.
GET by Id	<i>Does return</i> the <code>SchemaJson</code> and <code>MetadataJson</code> fields.
POST	Handles versioning of newly created form schema entities. If the new form schema is a draft, it is assigned the version -1; otherwise, it is assigned to next highest version number.
PUT/PATCH	Similarly to POST, allows undrafting form schemas. If <code>IsDraft</code> changes to <code>true</code> , the next highest version number is assigned. Further, the <code>SchemaJson</code> property is not updated if the schema is undrafted.
DELETE	Deletes associated form schema result and customer entities.

**Table 8: FormSchema OData Endpoints**

<b>FormSchemaResult</b>	
<b>Operation</b>	<b>Behavior</b>
GET All	Does not return the <code>ResultJson</code> and <code>MetadataJson</code> fields. This is done for performance – returning multiple entities with large string fields would consume a lot of bandwidth. If the UI needs to access those fields, they can still be retrieved via the GET by Id operation.
GET by Id	<i>Does return</i> the <code>ResultJson</code> and <code>MetadataJson</code> fields.
POST	<i>Standard implementation.</i>
PUT/PATCH	<i>Standard implementation.</i>
DELETE	<i>Standard implementation.</i>

**Table 9: FormSchemaResult OData Endpoints**

<b>Customer</b>	
<b>Operation</b>	<b>Behavior</b>
GET All	<i>Standard implementation.</i>
GET by Id	<i>Standard implementation.</i>
POST	<i>Standard implementation.</i>
PUT/PATCH	<i>Standard implementation.</i>
DELETE	Deletes associated form schema result entities.

**Table 10: Customer OData Endpoints**

All “GET All” endpoints listed above support OData filters. All POST/UPDATE/PATCH endpoints only allow updating an allowlisted set of properties and verify that foreign keys exist before creating/updating entities. All POST endpoints automatically generate a UUID as the entity’s new Id.

As a data store for the above endpoints, we defined the following tables per entity:

- FormSchema: Z00T1\_FSCHEMA
- FormSchemaResult: Z00T1\_FSCHEMARES
- Customer: Z00T1\_CUSTOMER

We additionally defined a single data element Z00T1\_SSTRING (“short string”) which holds an arbitrary string with a length of 255 characters. This string is used for text properties like descriptions or names.

The OData service project was configured via the SEGW transaction and is called T\_00\_T1\_SS23\_CSRD. The corresponding Web-API can be accessed via the relative /sap/opu/odata/sap/Z\_00\_T1\_SS23\_CSRD\_SRV path.

## Project Challenges

### Form Engine

Implementing the form engine was obviously a challenge. It was an idea that we had before the first sprint even started, but whether it would work out the way we had it in mind was not clear. In hindsight, it did. We will not go into any more details about the form engine though, as it has received plenty of discussion in the previous chapters.

### Form Builder

The WYSIWYG form builder was a challenge that we set ourselves for the last sprint – sprint 4. In sprint 2, we already possessed an initial and crude version of the form builder. It supported two code editors that allowed the user to directly enter the JSON schemas and see how they are rendered by the form engine. This version was initially only developed for ourselves to be able to quickly test and experiment. During the progress presentation, we received questions whether it could be possible to create the form schemas via a dedicated UI/UX flow. We considered these questions and, at the end of sprint 4, were able to deliver such a UI by reworking the initial version of the form builder.

The entire form builder was a challenge on top of the original scope/requirements that the customer set (which was, essentially, only the creation of a CSRD questionnaire). The implementation could, in hindsight, definitely be considered an epic regarding the time that had to be invested.

### Form Engine Performance

While developing the new version of the form builder, we discovered performance issues in the form engine when building large pages. Inside the form builder, taking any action (such as deleting an element on the page) took multiple seconds – a very noticeable lag that made using the form builder on such large pages nearly impossible. The problem did, however, only occur within the form builder. It was not an issue within the questionnaire page.

An investigation via multiple profiling sessions led to the conclusion that the culprit was the form engine which was running its rendering process multiple times after a user action was performed. The internal UI5 rendering process itself takes a relatively large amount of time (we measured up to ~150ms per render cycle for pages with ~120 elements). This, multiplied by an exemplary 10 immediate render cycle invocations, results in a UI lag of ~1.5+ seconds.

Once this was discovered, we applied three different measures to remedy the issue:

1. Only trigger render cycles inside the form engine when the form engine's context changes (and not for any other property changes) by introducing an incrementing `renderVersion` inside the context.
2. Move the creation of the UI5 control tree into the `onBeforeRendering` hook of the `FormEngine` control to delay the costly creation of the control tree until it becomes unavoidable.
3. Introduced batched UI5 `JSONModel` updates into our state implementation, i.e., only update the associated `JSONModel` of a state container *once* after all other subscribers/watchers are done executing.

With the above actions, the form engine inside the form builder now only renders once, per user action taken. Any remaining UI lag is attributed to the UI5 rendering process, which is outside of our control.

The above changes also improved the overall performance of the application in other areas, even though there was no noticeable issue.

### **Getting Familiar with the CSRD and ESRS**

While developing the UI was a big challenge, there was another, not as obvious challenge: Getting familiar with the CSRD and understanding the ESRS. Ultimately, part of the task was the creation of a questionnaire covering real parts of the CSRD/ESRS. To be able to create such a questionnaire, we needed to read through large portions of the ESRS, a collection of roughly 20 PDF files total, with page numbers varying between 20-60, on average. These files were legal-like documents that are, for students with a technical background, hard to assess and understand. Reading through them, and consequently, converting these into a questionnaire, is a task that took a large amount of time and effort and should, therefore, not be underestimated, despite not producing any “visible” results, apart from the questionnaire itself.

## Responsibilities

### Project Implementation

	Sprint 1 29.05.2023 - 04.06.2023	Sprint 2 05.06.2023 - 18.06.2023	Sprint 3 19.06.2023 - 02.07.2023	Sprint 4 03.07.2023 - 20.07.2023
All	<ul style="list-style-type: none"> <li>#29 Customer Meeting</li> <li>#33 Internal Alignment Meeting</li> <li>#13 Understand CSRD Problem Domain</li> </ul>	<ul style="list-style-type: none"> <li>#7 Create and Submit Progress Presentation</li> <li>#11 Customer Meeting</li> <li>#34 Internal Alignment Meetings</li> <li>#39 Find/Define/Outline CSRD Sections for the Final App</li> <li>#17 Form Engine Core</li> </ul>	<ul style="list-style-type: none"> <li>#31 Customer Meeting</li> <li>#35 Internal Alignment Meetings</li> </ul>	<ul style="list-style-type: none"> <li>#32 Customer Meeting</li> <li>#36 Internal Alignment Meetings</li> <li>#8 Create Final Presentation</li> <li>#9 Create Project Documentation</li> <li>#10 Final Project Submission/Customer Handover</li> <li>#37 Sprint 4 Refinement</li> </ul>
Minh Nhu	<ul style="list-style-type: none"> <li>#16 Create a Shared Postman Collection</li> </ul>	<ul style="list-style-type: none"> <li>#13 Develop Initial Set of Form Engine Controls</li> <li>#18 Questionnaire Page</li> </ul>	<ul style="list-style-type: none"> <li>#28 Create the Customer API</li> <li>#30 Home/Landing Page</li> <li>#26 Questionnaire/Customer Management Page</li> </ul>	<ul style="list-style-type: none"> <li>#53 Allow saving draft from FormSchemaResult and direct navigation to Result Overview Page</li> <li>#59 Form Builder: Next Generation</li> <li>#60 Form Schema Management Page</li> <li>#72 From Schema Result Version Migration</li> <li>#73 Testing/Bug-Fixing of the Final Application</li> </ul>
Manuel	<ul style="list-style-type: none"> <li>#11 Project Scaffolding</li> <li>#12 Setup CI/CD Pipelines</li> </ul>	<ul style="list-style-type: none"> <li>#24 Integrate the PoC Questionnaire into Form Engine/Questionnaire Page</li> </ul>	<ul style="list-style-type: none"> <li>#25 Connect Questionnaire Page to Backend</li> <li>#22 Create Form Schema API</li> <li>#23 Create Form Schema Results API</li> <li>#51 Form Engine: Conditional Pages/ Submitting</li> </ul>	<ul style="list-style-type: none"> <li>#53 Allow saving draft from FormSchemaResult and direct navigation to Result Overview Page</li> <li>#58 Visual Questionnaire Improvements</li> <li>#59 Form Builder: Next Generation</li> <li>#72 From Schema Result Version Migration</li> <li>#73 Testing/Bug-Fixing of the Final Application</li> </ul>
Tim	<ul style="list-style-type: none"> <li>#3: Create and Submit Sprint Documentation</li> <li>#11 Project Scaffolding</li> </ul>	<ul style="list-style-type: none"> <li>#4 Create and Submit Sprint Documentation</li> <li>#20 PoC Questionnaire Form Schema</li> </ul>	<ul style="list-style-type: none"> <li>#5 Create and Submit Sprint Documentation</li> <li>#21 CSRD Form Schema Implementation</li> <li>#40 Determine Exact Questionnaire Structure</li> <li>#38 Enhance Initial Set of Form Engine Controls</li> </ul>	<ul style="list-style-type: none"> <li>#6 Create and Submit Sprint Documentation</li> <li>Final Questionnaire Structure</li> <li>#71 CSRD Questionnaire – Result Checklist</li> <li>#74 Testing the CSRD Questionnaire</li> </ul>

Figure 9: User Story Assignments

### Project Documentation

Section	Assignee
Motivation	Tim Köneke
Requirements Analysis	Tim Köneke
Approach	
Agile Roadmap	Tim Köneke
Software Development	Manuel Römer
Testing Methodology	Manuel Römer
Software Architecture and Data Models	
Software Architecture	Manuel Römer
Semantic Data Model	Minh Nhu Duong
Technical Data Model	Minh Nhu Duong
Implementation	
SAPUI5 Application	Manuel Römer
ABAP Code	Manuel Römer, Minh Nhu Duong
Challenges	Manuel Römer, Tim Köneke
Responsibilities	Minh Nhu Duong
Access Information	Minh Nhu Duong
User Manual	Minh Nhu Duong

Table 11: Project Documentation Assignments

## Access Information

- **Package:** Z\_00\_T1\_SS23\_CSRD
- **Frontend:** ABAP-LAB-SS23-FRONTEND
- **User:** „normal“ system user accounts e.g., DEV-0xx
- **Transport Request:** S72K901368
- **URL:** [https://s72lp1.ucc.in.tum.de:8100/sap/bc/ui5\\_ui5/sap/z\\_00\\_t1\\_ss23\\_fe/index.html?sap-client=300](https://s72lp1.ucc.in.tum.de:8100/sap/bc/ui5_ui5/sap/z_00_t1_ss23_fe/index.html?sap-client=300)

# User Manual


## Homepage with four options:


- Fill out a questionnaire (always uses the highest version of the questionnaire/form schema)
- Customer Management Page
- Questionnaire (Form Schema) Management Page
- Reset Backend


## CSRD Reporting


Manage the customers and filled questionnaires.

**Actions**

**Fill Questionnaire**  
Fill out a new questionnaire  
  
Here you can fill out a new CSRD questionnaire.

**Customer Management**  
Questionnaire and Customer  
  
Here you can manage customers and their questionnaires that were filled in the past.

**Questionnaire Management**  
Create and edit FormSchemas  
  
Here you can create and edit your FormSchemas.


**Reset Backend**  
Clears all backend tables  
  
Resets all tables in the backend to a new, demo-like state.


## Questionnaire Management Page:

- Create new questionnaire (form schema) from scratch or duplicate an existing one
- Overview of all existing questionnaires (form schemas)
  - Every questionnaire can be duplicated and edited
  - Only title/description of undrafted questionnaires can be edited
  - Drafted questionnaires templates can be edited and deleted


**Management**  
Manage the questionnaires.


**Actions**


**Create Questionnaire**  
Create a new questionnaire.  
  
Here you can create a new questionnaire with the form builder.

**Duplicate Questionnaire**  
Create an editable version of an existing, undrafted questionnaire.  
  
Here you can create a questionnaire based on an existing one.

**Existing Questionnaires** ⓘ

**CSRD PoC**  
Description: The CSRD questi...  
Type: csrd  
Version: 0  
Created: Jul 17, 2023, 1:34:10 PM

**Demo Form Schema** **Draft**  
Description: A form schema s...  
Type: csrd  
Version: -1  
Created: Jul 17, 2023, 1:34:10 PM

**CSRD PoC New** **Draft**  
Description: The CSRD questi...  
Type: csrd  
Version: -1  
Created: Jul 17, 2023, 4:12:55 PM

## Form Builder Page:

- Left Side: Page Area
- Middle: Preview Area
- Right Side: Properties Area of Questionnaires, Page, Element, Schema and State

The screenshot shows the 'Questionnaire Builder' interface. On the left, the 'Pages' panel shows a list with '0 element(s)'. The middle 'Page Preview' panel shows a '+ Add New Element' button. The right 'Properties' panel has tabs for 'Questionnaire', 'Page', 'Element', 'Schema', and 'State'. The 'Questionnaire' tab is active, showing fields for 'Name' (with a placeholder 'Name' and description 'A display name of the questionnaire.'), 'Description' (with a placeholder 'Description' and description 'A description of the questionnaire.'), and 'Ref Conditions' (with a description 'Global conditions which can be referenced by effects and validation rules. It is recommended to define frequently used conditions here to minimize complexity.' and a 'Ref Conditions' section with a '+', a trash icon, and 'No data' text).

- Properties of Questionnaire - Ref Conditions
  - Reusable conditions for the whole questionnaire

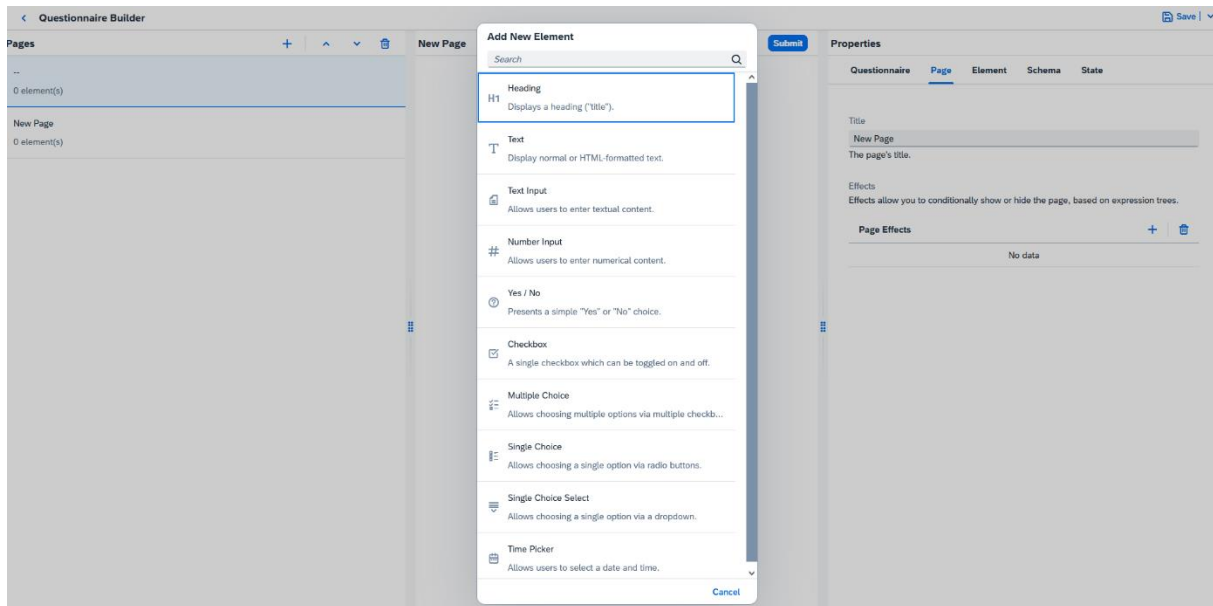
The screenshot shows the 'Ref Conditions' section. At the top, there's a header 'Ref Conditions' with a '+' icon and a trash icon. Below it, there are two conditions, each with a radio button and a dropdown menu.

The first condition is 'Page1Checked'. It has a dropdown menu with '!(not)' selected. Below it, there are two input fields: 'value' and 'page1done'.

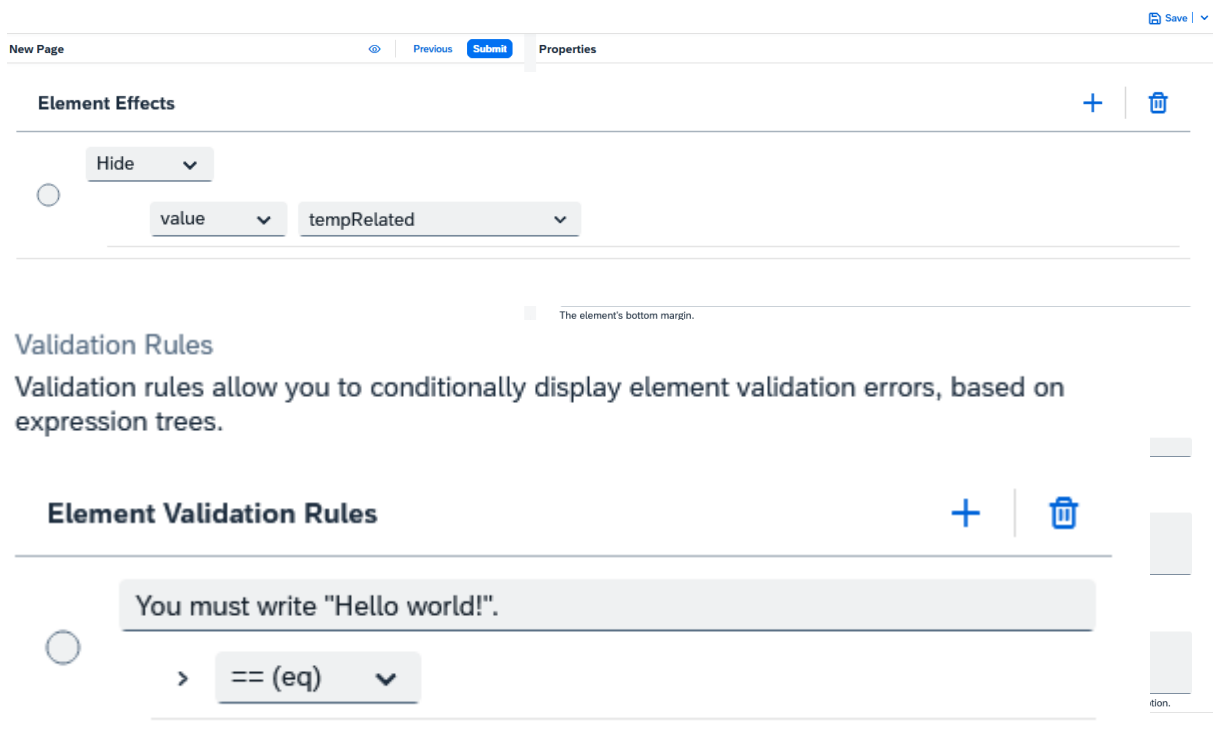
The second condition is 'Page2Checked'. It has a dropdown menu with '&& (and)' selected. Below it, there are two input fields: 'value' and 'page2done'. Below these, there is another dropdown menu with '> (gt)' selected. Below it, there are two input fields: 'value' and 'numEmployees'. At the bottom, there is a 'number' dropdown menu, a minus sign, and the value '250' with a plus sign.



- How to use the Form Builder
  - Start with adding a new page in Page Management (+)
  - Then add new elements from the selection dialog



- Afterwards, set properties for the selected element e.g. ID, Margins, Label, ...



- There is also the possibility to add effects and validation rules
- Example how effects/conditions are used for whole pages (applies to single controls as well):
  - Here we have our “DemoFormSchema”. Within the first page “All Controls”, we have a choice control where you can choose your favorite animal.
  - This control has the ID “animal” - this ID is used inside the conditions.

< Questionnaire Builder

Pages + ^ v 🗑️

All Controls 9 element(s)

The Panda 2 element(s)

Expression-Test 12 element(s)

All Controls

Tip: It has no effect what you choose here.

+ Add New Element

Your favorite animal  
If you had to pick one, what is your favorite animal?

☐ Cat 🐱 ☐ Dog 🐶

☐ Panda 🐼 ☐ Whale 🐳

+ Add New Element

Properties

Questionnaire Page **Element** Schema State

ID

animal

A unique identifier of the element.

Top Margin

The element's top margin.

Bottom Margin

The element's bottom margin.

Mandatory Element

- The options of the choice control are as follows:

Options \*

Options + 🗑️	
<input type="checkbox"/> ID	Display
<input type="checkbox"/> cat	Cat 🐱
<input type="checkbox"/> dog	Dog 🐶
<input type="checkbox"/> pandö	Panda 🐼
<input type="checkbox"/> whale	Whale 🐳

- Now the next page “The Panda”, this page will only be shown if in the choice control above, “Panda 🐼” is selected.

The screenshot shows the 'Questionnaire Builder' interface. On the left, a 'Pages' sidebar lists 'All Controls' (9 elements), 'The Panda' (2 elements, highlighted with a red box), and 'Expression-Test' (12 elements). The main area is titled 'The Panda' and contains a 'Panda Page' section. This section has a title 'Panda Page' and a description 'This page is only visible if you chose the Panda. 🐼'. There are three 'Add New Element' buttons and navigation controls (up, down, edit, delete) for each section.

- The Effect: the page is shown if the value (ID of the options of the “animal” control) is equal (eq) to “pandö”

## Properties

The screenshot shows the 'Properties' panel for the 'The Panda' page. The 'Page' tab is selected. The 'Page Effects' section is highlighted with a red box. It contains a 'Show' dropdown menu, a radio button, and a condition editor. The condition editor shows '== (eq)' with 'value' and 'animal' in the first row, and 'string' and 'pandö' in the second row. The 'value' and 'string' dropdowns are also highlighted with a red box.

- Same expression/condition as JSON:

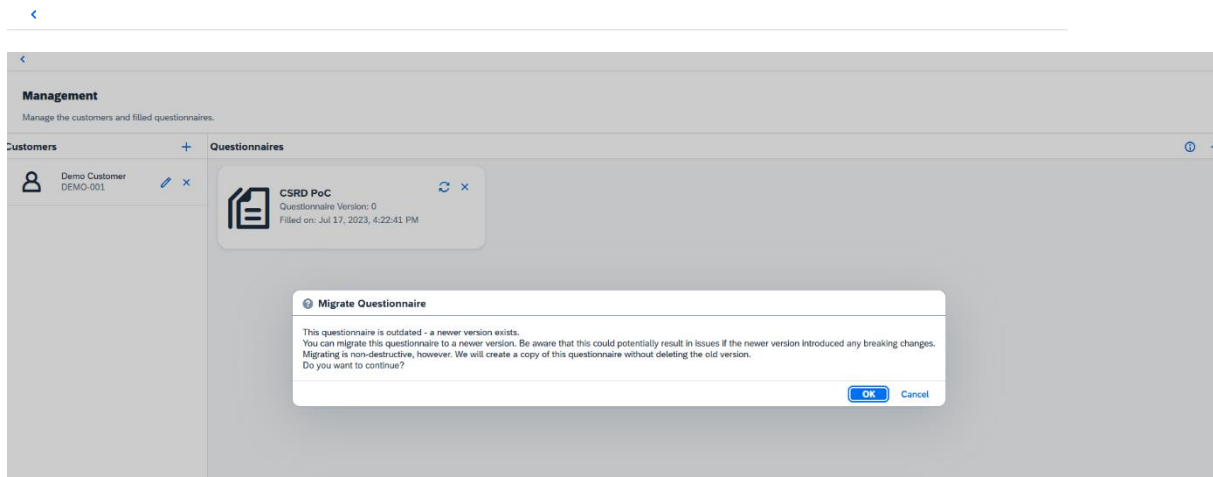
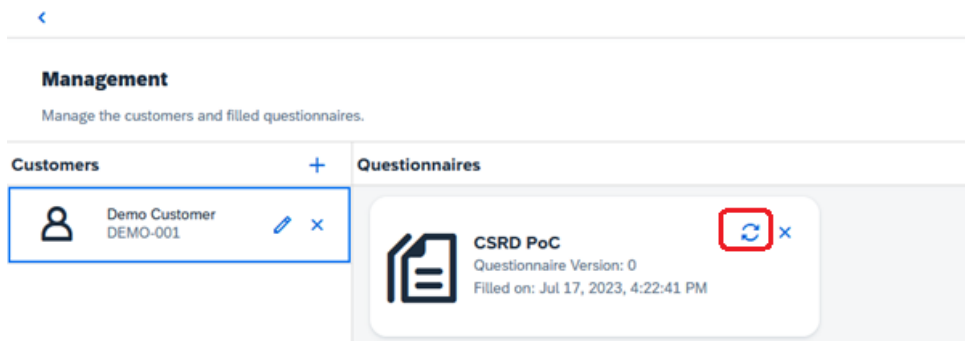
```

134 {
135   "title": "The Panda",
136   "effects": [
137     {
138       "effect": "show",
139       "condition": {
140         "type": "eq",
141         "left": {
142           "type": "value",
143           "id": "animal"
144         },
145         "right": "pandö"
146       }
147     },
148     {
149       "type": "heading",
150       "text": "Panda Page"
151     },
152     {
153       "type": "text",
154       "text": "This page is only visible if you chose the Panda. 🐼"
155     }
156   ]
157 }
158
159

```

## Customer Management Page:

- You can add new customers and fill out questionnaires for the existing customers
- Customers can be edited and deleted, if customer is deleted all questionnaires will also be deleted
- Questionnaires can be migrated, edited and deleted
- Migration is possible if a newer version of the questionnaire exists
  - However, it may break the already filled out questionnaire
  - To prevent data loss, the questionnaire will always be duplicated



## Questionnaire Page:

[<](#)

Save

### Questionnaire

CSRD Relevance

#### Is the CSRD relevant for your company?

Employees \*

Please enter the number of employees in your company:

—

0

+

Net sales in million euros \*

Please enter your net sales (in million euros):

—

0

+

Balance sheet total in million euros \*

Please enter your balance sheet total (in million euros):

—

0

+

[<](#)

Previous

Next

[<](#)

Save

### Questionnaire

Final Checklist

#### GOV-5 - Risk management and internal controls over sustainability reporting

The undertaking shall disclose the main features of its risk management and internal control system in relation to the sustainability reporting process(es).

- ☐ 34 (a): The scope, main features and components of the risk management and internal control processes and systems in relation to sustainability reporting
- ☐ 34 (b): The risk assessment approach followed, including the risk prioritisation methodology
- ☐ 34 (c): The main risks identified, actual and potential, and their mitigation strategies including related controls
- ☐ 34 (d): Description of how the undertaking integrates the findings of its risk assessment and internal controls as regards the sustainability reporting process into relevant internal functions and processes
- ☐ 34 (e): Description of the periodic reporting of the findings to the administrative, management and supervisory bodies
- ☐ GOV-5 done?

#### IRO-1-E1 – Description of the processes to identify and assess material climate-related impacts, risks and opportunities

[<](#)

Previous

Submit

## Appendix

### User Stories

#### #2 Task: Create Requirements Submission

The Requirements had to be submitted until 28.05.2023 - 23:59 on PowerPoint slides.

#### Create and Submit Sprint Documentation

#3 Task: Create and Submit Sprint Documentation (Sprint 1)

#4 Task: Create and Submit Sprint Documentation (Sprint 2)

#5 Task: Create and Submit Sprint Documentation (Sprint 3)

#6 Task: Create and Submit Sprint Documentation (Sprint 4)

Documentation for each Sprint was a basic requirement for this course based on the given template.

#### #7 Task: Create and Submit Progress Presentation

The PowerPoint slides for the Progress Presentation had to be submitted until 15.06.2023 - 23:59.

#### #8 Task: Create Final Presentation

The PowerPoint slides for the Progress Presentation had to be submitted until 20.07.2023 - 23:59.

#### #9 Task: Create Project Documentation

This document is the result of this task and a basic requirement of this course.

#### #10 Task: Final Project Submission / Customer Handover

We must submit the following things until 20.07.2023, EOD:

- Deploy: ABAP code in the form of one or multiple transport requests
- Deploy: Frontend in the form of one or multiple transport requests
- Do: Release all transport requests
- Submit via mail:

Zipped frontend code

ABAP package name

All transport request IDs  
Entry points of application  
Credentials to login

#### #11 User Story: Project Scaffolding

As a developer, I want to be able to develop the CSRD application. To do this, I need a frontend and a backend project in which I can develop the features.

#### #12 User Story: Setup CI/CD Pipelines

As a developer, I want to have CI/CD pipelines integrated into the frontend's GitHub repository to ensure that changes are continuously deployed to the target server and that bugs are quickly caught.

The pipelines should...

- automatically build the code and fail PRs on build errors
- automatically lint the code and fail PRs on lint errors

To fully leverage the safety that pipelines provide, all team members must only be able to make code changes via PRs.

#### #13 User Story: Understand the CSRD Problem Domain

As a user, I want the CSRD reporting application to capture the CSRD legislature. The application should guide me through parts of the questionnaire hidden in the law.

As a developer, I must be able to understand those laws in order to be able to develop the application correctly. To do so, I should read through those CSRD documents that apply to the scope of the requirements.

#### #16 User Story: Create a Shared Postman Collection

As a developer, I want to be able to quickly test the OData services that we are about to develop. For this purpose, we should have a shared Postman collection that can be accessed by all team members.

#### #17 User Story: Form Engine Core

As a developer, I want to be able to render arbitrary, yet typed, JSON schemas to a UI based on the SAPUI5 framework. For this purpose, we need a rendering mechanism that transforms the JSON schema into reactive UI Components that allow data mutations and reactions. This concept is called form engine.

The form engine will require several core functionalities that can, in other user stories, be enhanced. These core functionalities include:

- Types for the schema
- A rendering mechanism that transforms the schema to UI components
- An internal state model that is reactive and causes the engine to rerender on changes
- The ability to conditionally show/hide UI controls based on arbitrary conditions in the JSON schema (so called rules)
- Maybe: The ability to validate arbitrary user input by default and schema-defined validation rules

To be able to use such a form engine within SAPUI, the best route is to create a custom control. These integrate well with the framework and allow for reusability in different views.

#### #18 User Story: Develop Initial Set of Form Engine Controls

As a user, I want the CSRD questionnaire to display the questions with a variety of browser input fields, e.g., Input, Checkbox, RadioButton, etc. To support such inputs, they must be supported by the form schema and engine.

Once the form engine core exists (see #17), support for additional form schema elements ("controls") can be implemented. The list of elements that we want to support heavily depends on the requirements, but the following list might provide a good starting point:

- Text Inputs
- Yes/No Input
- StepInput / NumberInput
- Multi-Choice Inputs ( $\geq 2$  choices)
- Single-Choice Inputs ( $\geq 2$  choices)

#### #19 User Story: Questionnaire Page

As a user, I want to be able to fill in the CSRD questionnaire on a dedicated page. The page should contain the following features:

- Information about the current questionnaire page.
- The UI rendered by the form engine (see User Story: Form Engine Core #17) where the actual questionnaire is rendered.
- Controls to navigate between the questionnaire pages (e.g., "Forward"/"Backward" buttons).
- Optional: Metadata about the questionnaire progress (e.g., "Page X out of Y").



#### #20 User Story: PoC Questionnaire Form Schema

As a user, I want to be able to enter data in a questionnaire. For that to happen, the form engine (#17) needs to be fed with a form schema that outlines the questions to be asked and the final report to be generated.

We should develop a form schema that asks sufficient questions for a progress presentation demo. The schema may, but does not necessarily have to, use questions coming from the CSRD. If we don't have a deep enough understanding at this point, we can also use mocked example questions.

#### #21 User Story: CSRD Form Schema Implementation

As a user, I want to be able to enter information about the CSRD. As part of #39 and #40, we built the required understanding about what the questionnaire should look like. We should now convert these results into an actual form schema that can be delivered by the backend.

#### #22 User Story: Create the Form Schema API

As a user, I want the application to automatically use the newest CSRD report version. To achieve this, the frontend must be able to fetch the schemas from the backend.

This will be made possible via an OData service that provides a FormSchemaSet entity that has the following attributes:

- Id: string - a GUID
- Type: string - a unique type of the schema, e.g., "csrd"
- Version: number - e.g., 1, 2, ... Autoincremented.
- SchemaJson: string - the form schema, formatted as JSON string
- MetadataJson?: string - arbitrary metadata used by the frontend, formatted as JSON string
- CreatedAt: string - the creation date
- UpdatedAt: string - the last modified date

The OData service must provide the following conceptual endpoints (the URLs might vary):

- GET /FormSchemaSet('name') - to get schemas by name
- GET /FormSchemaSet('id') - to get schemas by ID

#### #23 User Story: Create the Form Schema Results API

As a user, I want the entered questionnaire data to be saved by the backend for later analysis. To achieve this, the app must be able to store the data in the backend.

This requires the creation of a new OData entity called FormSchemaResult/FormSchemaResultSet with the following attributes:

- id: string - a GUID
- formSchemaId: string - a GUID; references the form schema that was used to create this result
- result: string - a JSON string containing the unstructured result data
- metadata?: string - arbitrary metadata used by the frontend, formatted as JSON string
- createdAt: string - the creation date
- updatedAt: string - the last modified date

The OData service should offer the following endpoints:

- GET /FormSchemaResultSet
- GET /FormSchemaResultSet('id')
- POST /FormSchemaResultSet
- PATCH /FormSchemaResultSet('id')
- DELETE /FormSchemaResultSet('id')

#### #24 User Story: Integrate the PoC Questionnaire Into the Form Engine / Questionnaire Page

As a student, I want to be able to demonstrate the developed questionnaire (#20) in the progress presentation. For this purpose, we should setup the application such that it uses the hardcoded, client-side PoC questionnaire to render the form engine on the page created in #19.

#### #25 User Story: Connect Questionnaire Page to Backend

As a user, I want the questionnaire page to fully interact with the backend, so that all data is automatically fetched and saved without requiring any work from my side.

#### #26 User Story: Questionnaire/Customer Management Page

As a user, in addition to newly filling out questionnaires (i.e., creating a new questionnaire result, starting from scratch), I want to be able to see a list of the questionnaires which were filled in the past on a dedicated page. Via that list, I want to be able to view, edit, update and delete such questionnaires. For the page to work, we must logically also implement customer management, since questionnaires are a nested resource of a customer.

#### #27 User Story: Home/Landing Page

As a user, I want to see a home page that provides direct access to the questionnaire page (#19) and the questionnaire management page (#26). The home page should further provide information about what the application does, similar to a landing page.

### #28 User Story: Create the Customer API

As a user, I want to be able to associate a filled out questionnaire with a customer. For this sake, I need the system to manage customer data for me.

This requires the creation of a new OData entity called Customer with the following attributes:

- Id: string - a GUID
- Name: string - the customer's display name
- CustomerCode: string - an internal code number assigned to the customer
- CreatedAt: string - the creation date
- UpdatedAt: string - the last modified date

The OData service should offer the following endpoints:

- GET /CustomerSet
- GET /CustomerSet('id')
- POST /CustomerSet
- PATCH /CustomerSet('id')
- DELETE /CustomerSet('id')

### Customer Meeting

#29 Task: Customer Meeting (Sprint 1)

#30 Task: Customer Meeting (Sprint 2)

#31 Task: Customer Meeting (Sprint 3)

#32 Task: Customer Meeting (Sprint 4)

We have to regularly meet with the customer/project owner to notify them of our progress and to see whether we are on track.

### Internal Alignment Meeting(s)

#33 Task: Internal Alignment Meeting(s) (Sprint 1)

#34 Task: Internal Alignment Meeting(s) (Sprint 2)

#35 Task: Internal Alignment Meeting(s) (Sprint 3)

#36 Task: Internal Alignment Meeting(s) (Sprint 4)

In order for the project to succeed, the team members should regularly meet to sync and align. This should happen at least once per sprint.

### #37 Task: Sprint 4 Refinement

When the sprint starts, we should refine the goals based on our current progress. We might, for example, add new stories for bugs, challenges or other tasks.

### #38 User Story: Enhance Initial Set of Form Engine Controls

See #18 for a description of why a user would need form engine controls.

As part of #18, an initial set of form engine controls has been created. If the need arises, this sprint shall expand the controls to accommodate the requirements of the questionnaire that we want to build in this sprint.

### #39 User Story: Find/Define/Outline CSRD Sections for the Final App

As a user, I want the CSRD reporting app to provide a questionnaire that draws real-world information from the CSRD. As part of this story, we should find, define and outline (in written form) sections from the CSRD that can be transformed into a questionnaire. We should include sections only from the higher-level categories which were asked by the customer (most likely, "General Disclosures" and/or "Climate Change").

### #40 User Story: Determine Exact Questionnaire Structure

As a user, I want to be able to enter data about specific CSRD sections in the questionnaire. In #39, we already obtained a shared understanding of the sections that we want to include in the questionnaire.

As part of this story, we shall refine this understanding by:

- clarifying any unknowns in the sections
- applying feedback received by the customer about those sections
- maybe investigating (an)other section(s)
- Once that is done, we should have a clear idea of the questionnaire that we want to build for the final app.

### #51 User Story: Form Engine: Conditional Pages / Submitting

As a user, I want to be able to answer a questionnaire that, depending on which questions I answer, conditionally shows/hide certain pages. At the end, I want to be able to submit the answers of the questionnaire.

#### #53 User Story: Allow saving draft from FormSchemaResult and direct navigation to ResultOverview page.

As a user, I want to be able to save my questionnaire while not entirely finished. After finishing questionnaires I want to be able to navigate directly to the result page of a questionnaire from the management page. In General the questionnaires in the overview/management page should have some kind of tags/status which indicates the status (finished, unfinished,..)

#### #58 User Story: Visual Questionnaire Improvements

As a user, I want the questionnaire to be visually clear. A problem that exists at the moment is that pages with a lot of content are hard to grok due to visual clutter and the lack of coherence. This could be solved via small improvements like separators between sections or margins/paddings.

#### #59 User Story: Form Builder - Next Generation

As a developer of form schemas, I want to be able to use the existing form builder to edit and create new CSRD questionnaires. This is currently only supported on the frontend-level - saving the questionnaires is not supported.

Integrating the form builder with the backend requires, at least, the following changes:

- Edit/Save the current questionnaire
- See a list of all editable form schemas
- A UI/UX flow for creating new questionnaires
- Draft support

#### #60 User Story: FormSchema Management Page

As a developer of form schemas, I want to be able to navigate to a form schema management page where all existing formschemas and der status (draft, undrafted) are displayed. From this side I want to be able to go the existing from builder and create a new form schema from scratch or duplicate an existing formschema, edit and save it as a new form schema.

#### #71 User Story: CSRD Questionnaire - Result Checklist

As a user who filled out a CSRD questionnaire, I am interested in what I need to report under the CSRD. For this purpose, the questionnaire's last page should display a checklist that displays the data points I need to report on.

#### #72 User Story: Form Schema Result Version Migration

As a user who filled out a questionnaire, I want to be able to manually migrate the filled result to a newer version of the questionnaire, if one exists.

#### #73 User Story: Testing/Bug-Fixing of the Final Application

As a user, I want the CSRD application to work. In order to verify that this is the case, we, the developers, must properly test the application before the submission.

#### #74 User Story: Testing the CSRD Questionnaire

As a user, I want the CSRD questionnaire to work properly in order to be presented with the correct results. To verify that this is the case, we, the developers, should properly test the questionnaire.