Innovating Sales and Planning of Complex Industrial Products
Exploiting Artificial Intelligence

Deliverable 4.2
User Dialogue Component: Implementation and Test

Deliverable type:	Document
Deliverable reference number:	ITEA 20054 | D4.2
Related Work Package:	WP 4
Due date:	2024-10-31
Actual submission date:	2024-12-27
Responsible organisation:	IOTIQ
Editor:	Metin Tekkalmaz
Dissemination level:	Public
Revision:	Final | Version 1.01

Abstract:	This deliverable provides the documentation of reusable subcomponents of the User Dialogue Component (UDC). Subcomponent documentation includes information such as purpose, function, input and output data, and UI.
Keywords:	User Dialog Component, User Interface, UI/UX, Reusable UI Components

Table_head	Name 1 (partner)	Name 2 (partner)	Approval date (1 / 2)
Approval at WP level	Tomi Kanniainen (KONECRANES)	Bilge Özdemir (DAKIK)	20.11.2024 / 18.11.2024
Veto Review by All Partners			

Editor
Metin Tekkalmaz (IOTIQ)

Contributors
Ahmet Unal (ADESSO)
Kai Huittinen (WAPICE)
Klaus Hanisch (tarakos)
Mario Thron (IFAK)
Yazmin Andrea Pabon (PANEL)

Executive Summary

In sales, it is important to comprehend the customer's needs as precisely as possible and react to them as quickly as possible with an adequate price tag to gain a competitive advantage in the market. How the users interact with the tools, which are part of the sales process, plays an important role in the precision and speed of response to the customer. User interfaces, which are simple and intuitive yet provide all the information that is required by the user, are key to successful user interaction.

In the InnoSale solution, the User Dialog Component (UDC) constitutes the user interface mainly for the sales engineers and the customer. Its primary goal is to meet the high-quality needs of the sales domain. Besides the usual challenges of user interface design and implementation, the UDC should also satisfy the diverse user interaction needs arising from different use cases across various domains.

This deliverable provides the documentation of the reusable subcomponents of the UDC. The audience of this deliverable is mainly the frontend developers who will be developing the UDC of an InnoSale solutions tailored for a specific domain.

Table of Content

1 Introduction 1

2 Subcomponent Documentation 3

2.1 Reusable Component Library 3

2.1.1 Table Component 3

2.1.2 Text Mark & Bind Component 7

2.1.3 Email Drop Component 12

2.1.4 Login Component 14

2.1.5 Form Component 15

2.2 3D Web GUI for User Requirements and Product Presentation 18

2.2.1 Specification of User Requirements for Cranes 19

2.2.2 High-Quality Visualization of Cranes in the Operational Environment 19

2.2.3 Technological Background 20

2.3 Customer Feedback Component 21

2.4 Related Sales Cases Component 21

2.4.1 Overview 21

2.4.2 API 21

2.4.3 Examples 24

2.5 Price Estimation 29

2.5.1 Price Estimation User Interface 29

2.5.2 Component for Declaration of Pricing Parameters 30

2.6 Pricing Factors List Component 32

2.6.1 Overview 32

2.6.2 API 32

2.6.3 Examples 34

2.7 Guided Selling Component 35

2.8 AI Assisted Product Proposal Component 37

2.8.1 Angular Implementation 37

2.8.2 Vaadin Implementation 40

2.9 Data Visualization Component 42

2.9.1 Overview 42

2.9.2 API 43

2.9.3 Examples 45

2.10 Digital Product Description Component 52

2.10.3 Examples 53

3 Conclusion 57

4 Abbreviations 58

Figures

Figure 1 User Dialog Component and InnoSale architecture 2
Figure 2. InnoSale Table Example Usage 7
Figure 3. InnoSale Text Mark and Bind Example Usage 12
Figure 4. InnoSale Email Drop Component Example Usage 14
Figure 5. InnoSale Email Drop Component Example Usage 18
Figure 6. Requirements, Working Areas, Pick & Droppoints 19
Figure 7. GUI 3D presentation tool 20
Figure 8. Related Sales Cases Example Usage (Table) 28
Figure 9. Related Sales Cases Example Usage (Comparison Page) 28
Figure 10: Price Estimation User Interface 30
Figure 11: Component for Declaration of Pricing Parameters 31
Figure 12: Editing a Pricing Factor 32
Figure 13. InnoSale Pricing Factors Component Example Usage 35
Figure 14. User Interface of Guided Selling Component in 3D view. 36
Figure 15. User Interface of Guided Selling Component in 2D view. 37
Figure 16. InnoSale AI Assisted Product Proposal Component Example Usage 40
Figure 17. Illustration of AI Assisted Product Proposal Component. Please note that only some of the position rows are visible in the figure. 42
Figure 18. The generated line chart with the example code. 47
Figure 19. The pie chart generated with the example code. 48
Figure 20. The scatter plot generated with the given code. 50
Figure 21. The bar chart generated with the example code. 51
Figure 22. InnoSale Digital Product Description Component Example Usage 56

Tables

Table 1. InnoSale Table Use Example (.html) 6
Table 2. InnoSale Table Use Example (.ts) 6
Table 3. InnoSale Text Mark and Bind Use Example (.html) 9
Table 4. InnoSale Text Mark and Bind Use Example (.ts) 10
Table 5. InnoSale Email Drop Use Example (.html) 13
Table 6. InnoSale Email Drop Use Example (.ts) 13
Table 7. InnoSale Form Component Use Example (.html) 15
Table 8. InnoSale Form Component Use Example (.ts) 15
Table 9. Related Sales Cases Use Example (.html) 24
Table 10. Related Sales Cases Use Example (.ts) 25
Table 11. InnoSale Pricing Factors Component Use Example (.html) 34
Table 12. InnoSale Pricing Factors Component Use Example (.ts) 34
Table 13. InnoSale AI Assisted Product Proposal Component Use Example (.html) 39
Table 14. InnoSale AI Assisted Product Proposal Component Use Example (.ts) 39
Table 15. Digital Product Description Component Use Example (.html) 54
Table 16. Digital Product Description Component Use Example (.ts) 54

1 Introduction

Any complete and useful software solution needs to provide some sort of interface either to get input from the user, to present information to the user, or most of the time for both. Depending on numerous factors, such as the purpose of the solution, targeted users, the environment that the solution runs in, the design of the solution, the user interfaces may come in various forms and shapes. Command lines, text files, physical buttons and basic LED screens may be used as user interfaces of software solutions.

In the context of InnoSale, we talk about Graphical User Interfaces (GUI) due to their advantages, such as ease of use, faster learning curve, increased productivity and better user experience. Furthermore, InnoSale solution is designed to be a web-based solution due to their well-known advantages, such as cross-platform compatibility (accessible by browsers independent of the host system of the users), ease of maintenance and deployment (updates are on the server side and pushed to the users easily without (re-)installation on the client side), scalability, and accessibility. Therefore, more specifically, we focus on web-based GUIs.

User Dialog Component (UDC) in the InnoSale solution provides the GUI for the sales engineers, who are usually the first contact point for the customers, and for the customers themselves in some cases. UDC should provide the following high-level functionalities for the user:

- getting the inquiry details from the user for further processing
- creating inquiry records
- listing, searching, filtering existing inquiries
- presenting inquiry details and getting additional input about the inquiry
    - customer details
    - details of the interaction with the customer (e.g., relevant emails)
    - missing information to propose a solution
    - proposed solution for the inquiry
    - reasoning behind the proposed solution (e.g., highlights from the inquiry, laws and regulations, customer domain, etc.)
    - pricing information and reasoning behind it
    - relevant other solutions (e.g., previous projects)
- preparing a quotation

As in common web-based GUI implementations, UDC is responsible for displaying the user interface and handling user input. It relies on a supporting backend to process the data, perform operations on it, and to deliver the processed data back to the UDC to be presented to the user. In InnoSale, the Inference Engine and Knowledge Base components take the role of backend to provide the required functionality.

Figure 1 depicts the high-level architecture of the InnoSale solution. As shown in the figure, UDC interacts with Inference Engine and Knowledge Base to provide its functionality. The aim is to minimize the business logic and orchestration efforts at UDC level and let them be implemented in subcomponents of Inference Engine and Knowledge Base.

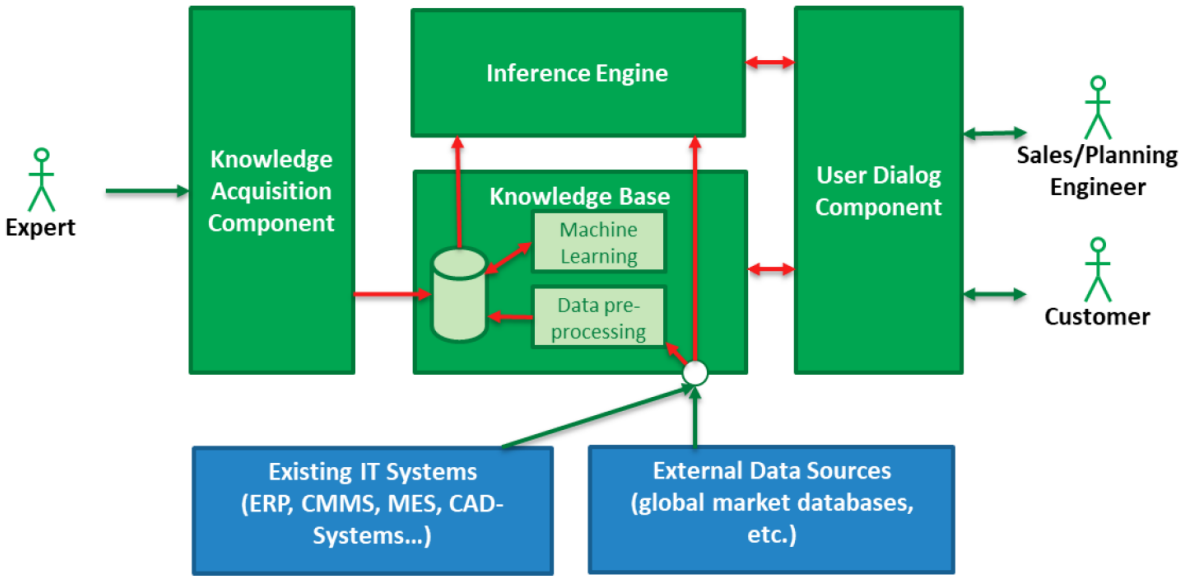

Figure 1 User Dialog Component and InnoSale architecture

As explained in InnoSale D4.1[1], it was decided to implement reusable GUI components that can be used as part of different use case frontend implementations in InnoSale as well as similar other projects in the future. Based on this, it is expected that the frontend of different use cases will be developed by using the GUI components developed as part of T4.1 as well as additional glue code and use case specific code, since the reusable components cannot cover every aspect of use case implementations.

The purpose of this deliverable is to provide documentation for the UDC reusable components specified in D4.1. Regarding each reusable component, required details, such as purpose, API and examples, have been provided.

[1] User Dialogue Component: Specification

2 Subcomponent Documentation

This section gives the documentation of the reusable GUI components for the User Dialog Component that are developed as part of T4.1. Wherever possible, the documentation approach followed by Angular Material Library[2] is followed. Hence, wherever suitable, component has the following subsections

- Overview –brief information about the component and describes different use cases
- API – formal documentation of the component with input and output parameters
- Examples – example usage of the component with the possible outcomes

In some cases, the component is not developed but an existing third-party component is included in the library (e.g. Login Component). In such cases, the original documentation of the component is referred, but still some example usage is provided in the context of InnoSale.

2.1 Reusable Component Library

The components documented in the following sections are designed to be smaller and more generic in the sense that they have a high chance of reusability in domains other than sales. But since they are an important part of the InnoSale use cases, they are developed as part of a component library.

2.1.1 Table Component

2.1.1.1 Overview

The InnoSale solution shall present data in tabular format, in many cases. One such case is the list of Sales Cases (Customer Inquiries). This component is expected to provide an easy to configure solution during the frontend development of different use cases with similar need for listing data as well as filtering and searching it.

This component is expected to solve the following problems:

- It is expected that UDC has some sort of list/table to present the current and past sales cases.
- This component implements a list showing tabular data. In InnoSale, it may be sales case with properties such as ID, company, assigned person, status, etc. The content of a cell can be text or icons.
- The search and filtering user interface will also be part of the list component; hence the user will be able to search the list using keywords or select filtering criteria and in turn the list will be updated with the responses coming from backend which is based on the search terms.
- The properties to be presented might change depending on the Use Case and the component will be configurable (number, title and width of the columns; page size, filtering options etc.) to adapt according to the use case.

2.1.1.2 API

```
import { InnosaleModule } from 'reusable-components';
```

[2] https://material.angular.io/components/categories

Directives

TableComponent<TData>	
Selector: innosale-table	
Properties	

`@Input()`
`columns:`
`TableColumnDefinition[]`
	Array of column definitions. Defines column properties such as column title, width, etc. See `TableColumnDefinition` description

`@Input()`
`tablePageData:`
`TablePageData<TData>`
	Table data to be rendered. Besides the rows to be presented includes additional data such as current page index. See `TablePageData` description

`@Input()`
`pageSizeOptions:`
`number[]`
	Page size options to be presented to the user. For example, if it is `[10, 25, 50]`, then user can change one of 10, 25 or 50 as number of rows to be displayed on a single page.

`@Input()`
`pageSize?:`
`number`
	Currently selected page size.

`@Input()`
`filterOptions?:`
`TableFilterOptions`
	Filter options to be provided to the user. See `FilterOptions` description

`@Output()`
`tableUpdate:`
`EventEmitter<TableUpdateEvent>`
	Emits when any parameter (e.g. sort column/direction, page size, etc.) that may affect the data to be displayed on the table changes. See `TableUpdateEvent` description

`@Output()`
`tableRowClick:`
`EventEmitter<TableRowClickEvent<TData>>`
	Emits when a row is clicked. See `TableRowClickEvent` description

Classes/Interfaces

FilterOptions	
Contains filter options to be presented to the user.	
Properties	
`title: string`	Title of the filter popup.

`options: {`
`  title: string;`
`  checkboxes: string[];`
`}[]`
	An array composed of
	- Title of the option groups, and
	- An array of filter options

TableColumnDefinition	
Contains column definitions.	
Properties	

`title: string`	Title of the column
`fieldName: string`	Name of the field in `TData` to be used to fill the column.
`isSortable: boolean`	Whether the data can be sorted by the column.
`width?: string`	Preferred width of the column.
`isIcon?: boolean`	Whether the column should display an icon instead of text.
TablePageData<TData>	
Data to be rendered on the table.	
Properties	
`dataSlice: TData[]`	Array of `TData` to be presented as the rows of the table. Should contain "page size" number of rows. The mapping between the `TData` field and column is defined by `columns` input field of the table.
`sliceIndex: number`	The slice index (i.e. page index) of the provided data. For example, if the data is the second slice/page of the whole data, then it should be 1.
`totalLength: number`	Total number of data rows.
TableUpdateEvent	
Represents a table update event, which is emitted whenever something affecting displayed data is changed. Once the table user receives the event, it should update the `tablePageData` accordingly.	
Properties	
`pageIndex: number`	Index of page to be displayed.
`pageSize: number`	Size (row count) of the page to be displayed.
`sortColumn?: string`	Column according to which the data is to be sorted.
`sortDirection?: "asc" | "desc"`	Sorting direction. Either asc(ening) or desc(ending).
`search?: string`	Search terms entered by the user.
`filter?: any`	Filter options selected by the user.
TableRowClickEvent<TData>	
Represents a table row click event.	
Properties	
`index: number`	Index of the row that has been clicked. Should be between 0 and (`pageSize` - 1).
`data: TData`	Data that is displayed on the clicked row.

2.1.1.3 Examples

Use of InnoSale table component as shown in Table 1 and Table 2 results a view similar to one given in Figure 2.

Table 1. InnoSale Table Use Example (.html)

```
<div class="mat-elevation-z4 bordered">
  <innosale-table
    [columns]="columns"
    [filterOptions]="filterOptions"
    [pageSizeOptions]="[10, 25, 100]"
    [pageSize]="10"
    [tablePageData]="tablePageData"
    (tableRowClick)="tableRowClick($event)"
    (tableUpdate)="tableUpdate($event)"
    />
</div>
```

Table 2. InnoSale Table Use Example (.ts)

```
class InquiryListComponent implements OnInit {

  protected readonly columns: TableColumnDefinition[] = [
    { title: 'Case Id', fieldName: 'id', isSortable: true },
    { title: 'Customer', fieldName: 'customer', isSortable: true },
    { title: 'Customer Id', fieldName: 'customerId', isSortable: true },
    { title: 'Description', fieldName: 'description', isSortable: false },
    { title: 'Date', fieldName: 'date', isSortable: true },
    { title: 'Region number', fieldName: 'region', isSortable: true },
    { title: 'Product Category', fieldName: 'category', isSortable: true },
    { title: 'Status', fieldName: 'status', isSortable: true, isIcon: true },
  ];

  protected readonly filterOptions: FilterOptions = {
    title: 'Filters',
    options: [
      {
        title: 'Regions',
        checkboxes: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
      },
      {
        title: 'Status',
        checkboxes: ['Ready', 'Missing Information', 'Declined',
          'Unprocessed', 'Order', 'Production', 'Shipped']
      },
      {
        title: 'Product Category',
        checkboxes: ['A', 'B', 'C']
      },
    ]
  };

  tablePageData: TablePageData<SalesCase> = new TablePageData();
```

```
  ngOnInit() {
    this.updateInquiryList();
  }

  tableUpdate(event: TableUpdateEvent) {
    this.updateInquiryList(event);
  }

  tableRowClick(clickEvent: TableRowClickEvent<SalesCase>) {
    // go to Sales Case details page
  }

  updateInquiryList(event?: TableUpdateEvent) {
    // based on the event, fetch data from the backend
    // and set this.tablePageData using the response
  }
}
```

Figure 2. InnoSale Table Example Usage

2.1.2 Text Mark & Bind Component

2.1.2.1 Overview

The InnoSale solution should be able to analyse customer e-mails and extract requirements needed to recommend solutions and quotations. As part of this process, product/project forms are automatically filled. For a better user experience, sections of the email that were used to extract and the relation between the relevant email sections and automatically filled form fields should be highlighted.

By taking a (multiple) labelled text and a list of UI components that are in relation with the labels, this component is able to:

- Present each labelled text in its own tab.

- Highlight the labelled text sections with colours.
- Draw connection lines between labelled text sections and the corresponding UI elements.
- Use a proper colour palette to make marks as distinct as possible.
- Hide/show binding lines depending of the visibility of the ends (text section or UI element) due to scrolling or currently open tab.
- Switch tabs (if necessary) and auto-scroll to the origin of an extraction (i.e. labelled text) in case it is outside of the current viewport.
- Show marks and binding lines faded, unless the user hovers them with mouse, for improved visibility.
- Provide a flexible way to present a context menu to interact with the binding relations.

2.1.2.2 API

import { InnosaleModule } from 'reusable-components';

Directives

TextMarkBindComponent	
Selector: innosale-text-mark-bind	
Properties	
`@Input() enclosingContainer: HTMLElement | undefined`	The parent/ancestor HTML element that contains both the Text Mark & Bind component and all the target HTML elements.
`@Input() targets: BindTarget[]`	An array of target descriptions defining which label should be bind with which HTML element.
`@Input() markedTextSpecs: MarkedTextSpec[]`	An array of source descriptions containing text with appropriate labels.
`@Input() contextMenuSpec: ContextMenuItemSpec[]`	An array of context menu specifications defining the menu items to be displayed whenever a label is right clicked.
`@Input() markTagName: string`	The name of the tag used to label the text parts used for extraction. The default is `mark`
`@Input() fieldIdAttributeName: string`	The name of the attribute used to specify the id of the field extracted from the text part. The default is `mark`
`@Input() sourceLocation: "left" | "right"`	Location of the marked text (i.e. source) relative to the target elements. Can be either left or right. The default is `"right"`
`@Input() targetOffset: number`	Horizontal displacement of binding points on targets in pixels. The default is 10

`@Input()`
`tabPosition:`
`"top" | "bottom"`
	Tab position relative to the text. Can be either top or bottom. The default is `"top"`

Classes/Interfaces

BindTarget	
Defines the target to be bound	
Properties	
`fieldId: string`	The field id used to mark the corresponding text.
`elementId: string`	The of the HTML element to be bound.
ContextMenuItemSpec	
Specifies a menu item to be displayed in the context menu.	
Properties	
`label?: string`	Menu item text.
`color?: string`	Menu item color.
`action?: (fieldId: string) => void`	Callback function when the menu item is clicked. The field id of the right clicked marked text is passed as parameter.

2.1.2.3 Examples

Use of InnoSale table component as shown in Table 1 and Table 2 results a view similar to one given in Figure 2.

Table 3. InnoSale Text Mark and Bind Use Example (.html)

```
<div class="cols" #wrapper>
  <form class="col scrollable-bind-container"
      (scroll)="onScroll()"
      [formGroup]="form">
    <div *ngIf="!showList; else parameterList">
      <formly-form
          [form]="form"
          [fields]="fields"
          [model]="model"
      ></formly-form>
    </div>
  </form>

  <innosale-text-mark-bind
    class="col scrollable-bind-container"
    (scroll)="onScroll()"
    [enclosingContainer]="wrapper"
    [targets]="bindTargets"
    [markedTextSpecs]="markedTextSpecs"
    [contextMenuSpec]="contextMenuItems"
```

```
    sourceLocation="right"
    [targetOffset]="10"
    tabPosition="top"
  >
  </innosale-text-mark-bind>
</div>
```

Table 4. InnoSale Text Mark and Bind Use Example (.ts)

```
export class InquiryDetailTechnicalSpecificationsComponent {
  showList = false;
  bindTargets: BindTarget[] = [];
  form = new FormGroup({});

  @ViewChild(TextMarkBindComponent) private markBindComponent:
    | TextMarkBindComponent
    | undefined;

  protected readonly fields: FormlyFieldConfig[] = [
    {
      key: 'load',
      type: 'input',
      props: {
        label: 'Load capacity (kg)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'cranes',
      type: 'input',
      props: {
        label: 'Number of cranes on runway',
        required: true,
        type: 'number',
      },
    },
    // ...
    {
      key: 'attachment',
      type: 'select',
      props: {
        label: 'Attachment / Roof structure',
        required: true,
        options: [
          { value: 1, label: 'Steel structure' },
          { value: 2, label: 'Concrete ceiling' },
          { value: 3, label: 'I-beam' },
        ],
      },
    },
  ];
```

```
  protected model: any = { };
  protected markedTextSpecs: MarkedTextSpec[] = [];

  constructor(
    private fb: FormBuilder,
    private renderer: Renderer2,
    private cd: ChangeDetectorRef,
    private _inquiryChangeService: InquiryChangeService,
  ) {
    this.form = this.fb.group({});

    _inquiryChangeService.inquiry$.subscribe((salesCase) => {
      this.model = salesCase?.configuration?.technical_spec;

      let markedTextSpecs: MarkedTextSpec[] = [];

      // here the MarkedTextSpec filled using the response from backend
      salesCase?.documents?.forEach((document) => {
        markedTextSpecs.push({
          title: document.title,
          content: document.content?.join("<br>")
        });
      });

      // this is just a sample to show how MarkedTextSpec may look like
      markedTextSpecs.push({
        title: "document.pdf",
        content: "No <mark innosale-field-id='suspensions'>content</mark>"
      });

      this.markedTextSpecs = markedTextSpecs;
    });
  }

  ngAfterViewInit() {
    this.bindTargets = this.fields.map(
      (e) => <BindTarget>{ fieldId: e.key, elementId: e.id }
    );
    this.cd.detectChanges();
  }

  onScroll() {
    this.markBindComponent?.repaint();
  }

  // do whatever is needed once the context menu items are clicked
  contextMenuItems: ContextMenuItemSpec[] = [
    {
      label: 'Confirm',
      color: '#50e423',
      action: (fieldId: string) => {
        console.log(`Confirm triggered for key: ${fieldId}`);
      },
    },
```

```
    {
      label: 'Delete',
      color: '#FF5555',
      action: (fieldId: string) => {
        console.log(`Delete triggered for key: ${fieldId}`);
      },
    },
  ];
}
```

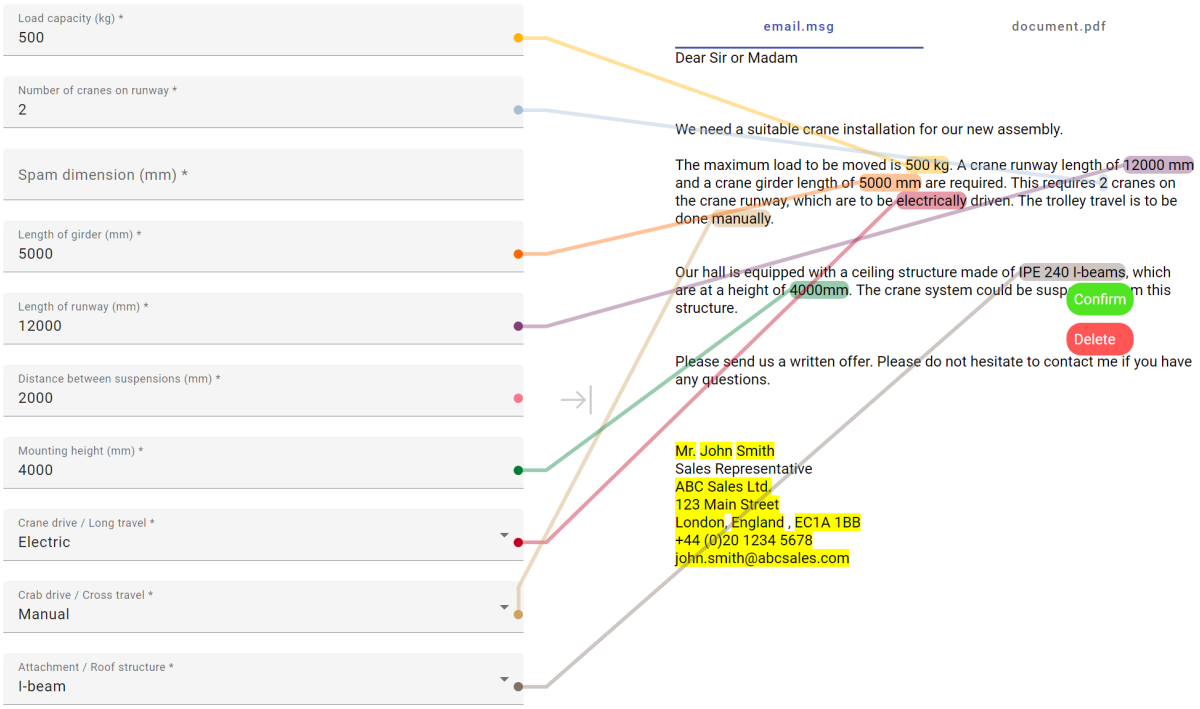

Figure 3. InnoSale Text Mark and Bind Example Usage

2.1.3 Email Drop Component

2.1.3.1 Overview

The InnoSale solution should accept drag & drop events for e-mails. Users of the InnoSale solution would like to pass some of the customer emails for analysis. To do so, the user should be able to drag an email from the email client and drop on the InnoSale solution. Email Drop Component provides the required functionality.

Email Drop Component provides the following functionality:

- Adds e-mail drop support to the desired parts of the InnoSale UDC
- Makes any UI element droppable, since it is developed to be a directive
- Calls a callback function with the email contents, whenever an e-mail is dropped

2.1.3.2 API

import { InnosaleModule } from 'reusable-components';

Directives

TextMarkBindComponent	
Directive: innosaleEmailDrop	
Properties	
`@Input() emailExtensions: string[]`	List of file extensions that the drop area should accept. Default is `['msg']`
`@Output() innosaleEmailDrop: EventEmitter<File>`	Emits the File object representing the dropped file

2.1.3.3 Examples

Table 5. InnoSale Email Drop Use Example (.html)

```
<div (innosaleEmailDrop)="onEmailDrop($event)">
  <div class="inner-box">
    <div class="image-container">
      <img src="assets\upload.png" alt="">
    </div>
    <p>Drag and Drop your E-mails here</p>
    <div class="line-container">
      <hr>
      <span>OR</span>
      <hr>
    </div>
    <button class="browse" (click)="openFileBrowser()">Browse
Files</button>
  </div>
</div>
```

Table 6. InnoSale Email Drop Use Example (.ts)

```
@Component({
  selector: 'app-inquiry-list',
  templateUrl: './inquiry-list.component.html',
  styleUrls: ['./inquiry-list.component.css'],
})
export class InquiryListComponent {

  // ...

  protected onEmailDrop(file: File) {
    file.arrayBuffer().then((buffer) => {
      const email: this.arrayBufferToBase64(buffer);
      // use email content to make a request to the backend
    });
  }

  protected openFileBrowser() {
    const input = document.createElement('input');
    input.type = 'file';
```

```
    input.accept = '.msg';

    // Listen for file selection
    input.addEventListener('change', this.handleFileSelection);

    // Trigger the file browser dialog
    input.click();
  }

  private handleFileSelection(event: Event) {
    const input = event.target as HTMLInputElement;
    const file = input.files?.[0];

    if (file && file.name.endsWith('.msg')) {
      // Perform further processing or upload the file
      console.log('File selected:', file.name);
    } else {
      console.log('Invalid file selected. Only .msg files are allowed.');
    }
  }

  private arrayBufferToBase64(buffer: ArrayBuffer): string {
    let binary = '';
    const bytes = new Uint8Array(buffer);
    const len = bytes.byteLength;
    for (let i = 0; i < len; i++) {
      binary += String.fromCharCode(bytes[i]);
    }
    return window.btoa(binary);
  }
}
```

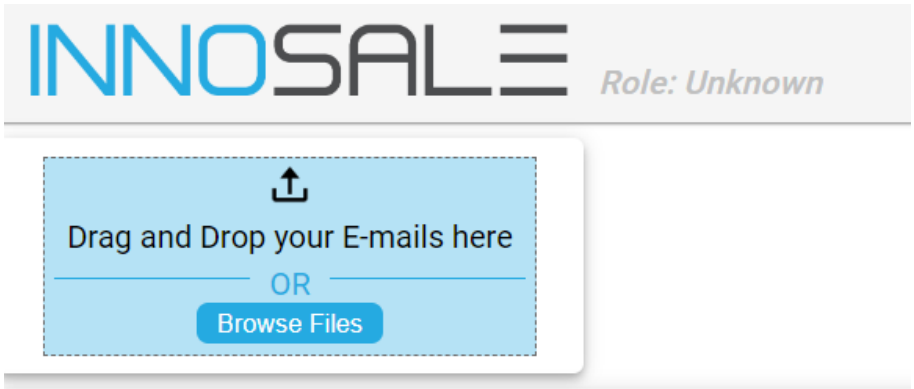

Figure 4. InnoSale Email Drop Component Example Usage

2.1.4 Login Component

2.1.4.1 Overview

As any non-public (web) application, authentication and authorization are important aspects of InnoSale solution. Since security is a critical area requiring specialized knowledge, in InnoSale, it has been decided to use an external solution for identity and access management and Keycloak has been chosen for this purpose. In UDC, for similar reasons, instead of

developing the authentication client a suitable third party, namely keycloak-angular, is chosen.

2.1.4.2 API and Examples

For API reference and usage examples, please refer to following keycloak-angular library page:
https://www.npmjs.com/package/keycloak-angular

2.1.5 Form Component

2.1.5.1 Overview

The InnoSale solution should be able to present forms, which are basically a group of label and input field pairs. One such use case is the technical specification of the product requested as part of an inquiry. Another one is customer details. Form Component should provide an easy way to specify the resulting form. For this purpose a third party solution, named formly, has been chosen.

2.1.5.2 API

The library home page is https://formly.dev/
The API documentation is accessible from https://formly.dev/docs/api/core/
Some usage examples can be seen here https://formly.dev/docs/examples

2.1.5.3 Examples

Table 7. InnoSale Form Component Use Example (.html)

```
<form class="col scrollable-bind-container"
  (scroll)="onScroll()"
  [formGroup]="form">
  <formly-form
    [form]="form"
    [fields]="fields"
    [model]="model"
  ></formly-form>
</form>
```

Table 8. InnoSale Form Component Use Example (.ts)

```
@Component({
  selector: 'app-inquiry-detail-technical-specifications',
  templateUrl: './inquiry-detail-technical-specifications.component.html',
  styleUrls: ['./inquiry-detail-technical-specifications.component.css'],
})
export class InquiryDetailTechnicalSpecificationsComponent {
  form = new FormGroup({});

  protected readonly fields: FormlyFieldConfig[] = [
    {
      key: 'load',
      type: 'input',
```

```
      props: {
        label: 'Load capacity (kg)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'cranes',
      type: 'input',
      props: {
        label: 'Number of cranes on runway',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'spam',
      type: 'input',
      props: {
        label: 'Spam dimension (mm)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'girder',
      type: 'input',
      props: {
        label: 'Length of girder (mm)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'runway',
      type: 'input',
      props: {
        label: 'Length of runway (mm)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'suspensions',
      type: 'input',
      props: {
        label: 'Distance between suspensions (mm)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'mounting',
      type: 'input',
```

```
      props: {
        label: 'Mounting height (mm)',
        required: true,
        type: 'number',
      },
    },
    {
      key: 'craneDrive',
      type: 'select',
      props: {
        label: 'Crane drive / Long travel',
        required: true,
        options: [
          { value: 1, label: 'Manual' },
          { value: 2, label: 'Electric' },
        ],
      },
    },
    {
      key: 'crabDrive',
      type: 'select',
      props: {
        label: 'Crab drive / Cross travel',
        required: true,
        options: [
          { value: 1, label: 'Manual' },
          { value: 2, label: 'Electric' },
        ],
      },
    },
    {
      key: 'attachment',
      type: 'select',
      props: {
        label: 'Attachment / Roof structure',
        required: true,
        options: [
          { value: 1, label: 'Steel structure' },
          { value: 2, label: 'Concrete ceiling' },
          { value: 3, label: 'I-beam' },
        ],
      },
    },
  ];

  protected model: any = { };

  constructor(
    private fb: FormBuilder,
    private _inquiryChangeService: InquiryChangeService,
  ) {
    this.form = this.fb.group({});

    _inquiryChangeService.inquiry$.subscribe((inquiry) => {
```

```
        // use inquiry to set model, here technical_spec contains a json
        // with the right attribute names matching the field keys
        // (e.g. load, cranes, spam, etc.)
        this.model = inquiry?.technical_spec;
    });
  }
}
```

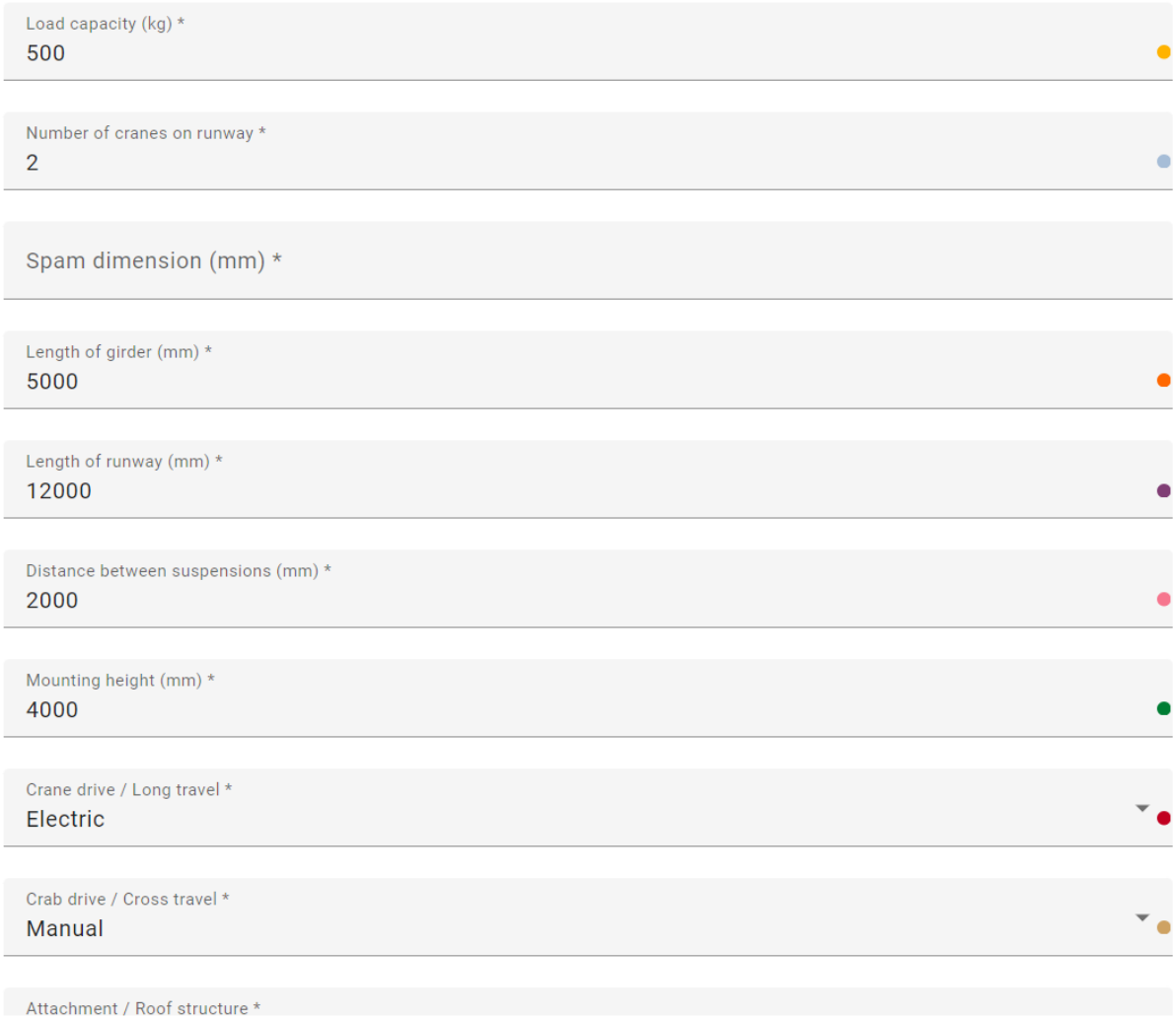

Figure 5. InnoSale Email Drop Component Example Usage

2.2 3D Web GUI for User Requirements and Product Presentation

The developed 3D software solution serves two main functions: the specification of user requirements for cranes and the high-resolution visualization of cranes in their actual operational environment, such as a factory hall. It is designed to guide users through the planning and visualization of crane systems, helping them find the right products for their specific needs.

The application offers an interactive, web-based graphical user interface (GUI) that allows users to manage both technical requirements and the visual representation of cranes within

a 3D space. The following sections provide detailed descriptions of the two primary tasks of the GUI.

2.2.1 Specification of User Requirements for Cranes

The first major function of the software focuses on user input and specification of requirements for crane systems. In this phase, users interact with a 3D environment to define operational parameters for crane operations.

User Interactions:

- **Drawing Workspaces:** Users can define work areas as transparent boxes by specifying length, width, and height in the 3D environment. This defines the areas where the crane will operate.
- **Definition of Pick and Drop Points:** Users can specify points where materials will be picked up and dropped off, which are visually represented within the 3D view.
- **Input of Material Flow Routes:** Material flows between the defined pick and drop points can be specified. These routes are displayed within the 3D environment to visualize material flow and crane movement.

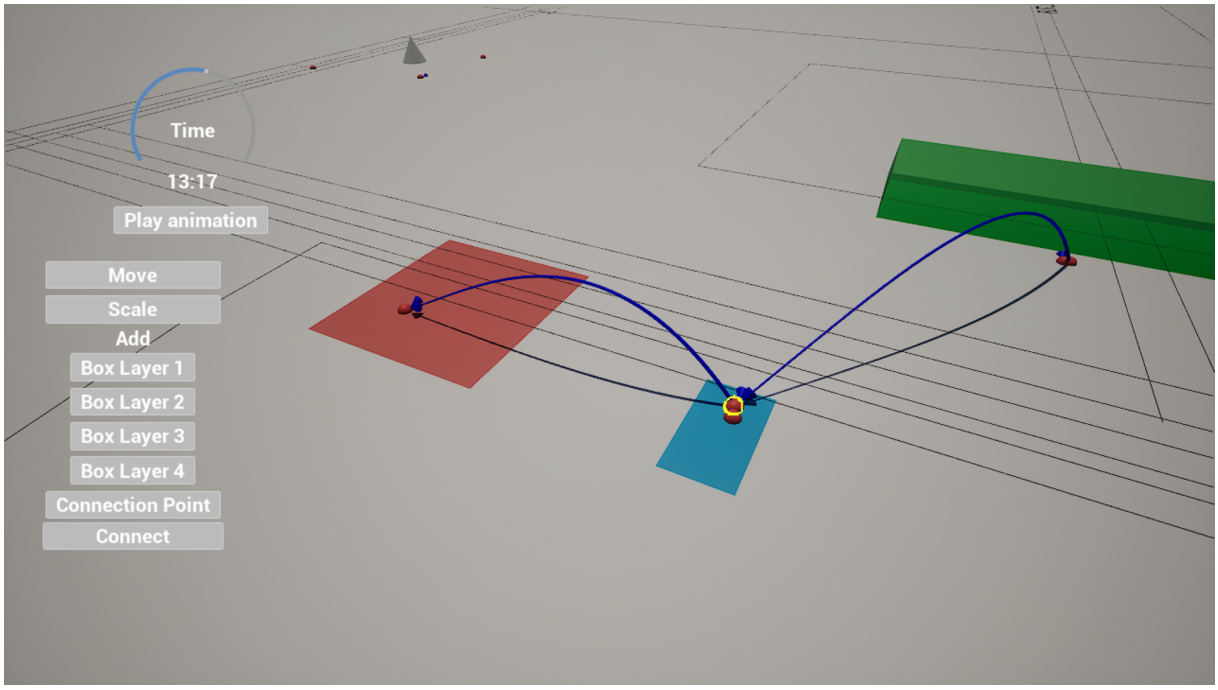

Figure 6. Requirements, Working Areas, Pick & Droppoints

A screenshot of the GUI could show a scene where the user is drawing a workspace as a transparent box, with pick and drop points and material flow routes highlighted in the 3D environment. This would illustrate how the user defines specifications directly in the 3D interface.

2.2.2 High-Quality Visualization of Cranes in the Operational Environment

After specifying all requirements, the software's second function comes into play: the detailed visualization of cranes in their intended operational environment. Based on the user's inputs end results of the guided selling tool, the crane models are automatically generated and placed within the 3D environment.

Process:

- **Guided Selling Tool:** Based on the defined requirements, a guided selling tool automatically suggests appropriate crane products.
- **Automatic Visualization:** Once a product configuration is selected, the visualization tool loads the crane models and places them in the virtual factory hall.
- **Interactive Features:**
    - Users can add, position, and delete interior elements like machines or vehicles to create a more realistic operational environment.
    - Previously defined elements like pick and drop points, workspaces, and material flow routes are also displayed and visualized.
    - Users can navigate through the 3D factory model from different perspectives.
    - Animations of crane movements can be started and stopped to visualize operations within the factory.

Figure 7. GUI 3D presentation tool

2.2.3 Technological Background

Due to the high complexity and size of the 3D models, it is necessary to render them server-side. This reduces loading times and minimizes the data volume that needs to be transferred to the client. The generated images are streamed to the user's web browser, ensuring a smooth user experience.

Communication between the GUI components and the server is handled via Websockets, enabling real-time interaction within the 3D environment.

The developed 3D web GUI provides a highly interactive and user-friendly solution for specifying crane requirements and realistically visualizing crane models in their future operational environments. By combining guided product selection, real-time visualization,

and animations, the software offers an intuitive platform for planning and presenting complex crane systems.

2.3 Customer Feedback Component

One of the purposes of InnoSale solution is to extract the requirements of the customer based on customer input (e.g., email). But in most of the cases, the initial customer input may not contain all the required information to prepare an offer. Customer Feedback Component was initially planned to collect the remaining/missing information from the customer as easy as possible and the approach would be to use the Form Component described in Section 2.1.5. But after further consideration, it was decided to cancel the implementation of this component, mainly because the added value of the component would be marginal on top of the Form Component. From the UDC point of view, it would be a special configuration if the Form Component, which can actually be implemented in the demonstrators if needed. The main challenge of such a use case lies in (a) deciding which fields to request from the user and (b) ensuring proper authorization/authentication so that only the necessary users have access, both of which are beyond the scope of UDC.

2.4 Related Sales Cases Component

2.4.1 Overview

The InnoSale solution should be able to list similar Sales Cases (Customer Inquiries) that might be related to the new Inquiry. This related sales cases list can help the Sales Engineer to identify similarities and verify the adequacy of the final offer generated in comparison with previous or similar previous offers. The component can indicate how similar/relevant those previous cases are using a score system that will define similarity between the current Inquiry and the previous case. Furthermore, if information regarding field level similarity is available, it allows selection of a previous Sales Case for more in detail presentation of similarity.

This component provides the following functionality:

- Provides a user-friendly overview list of similar cases with a clear indication of similarity score.
- If configured so, allow for a more detailed presentation of similarity between the current inquiry and a previous sales case.
- Detailed similarity info is presented at a field level.
- Provide search and filtering capabilities in a way that similar cases with specific parameters ranges can be listed for comparison.

2.4.2 API

This component is based on the InnoSale Table Component presented in section 2.1.1. Therefore, although it extends (and in some sense, restricts) the Table Component, the API and usage is very similar. Main differences are highlighted below with **bold**.

import { RelatedSalesCasesModule } from 'library';

Directives

RelatedSalesCasesComponent <TData>	
Selector: innosale-related-sales-cases	
Properties	

`@Input()`
`columns:`
`RelatedSalesCasesTableColumnDefinition`
`[]`
	Array of column definitions. Defines column properties such as column title, width, etc. See `RelatedSalesCasesTableColumnDefinition` description

`@Input()`
`tablePageData:`
`TablePageData<TData>`
	Table data to be rendered. Besides the rows to be presented includes additional data such as current page index. See `TablePageData` description

`@Input()`
`pageSizeOptions:`
`number[]`
	Page size options to be presented to the user. For example, if it is `[10, 25, 50]`, then user can change one of 10, 25 or 50 as number of rows to be displayed on a single page.

`@Input()`
`pageSize?:`
`number`
	Currently selected page size.

`@Input()`
`filterOptions?:`
`TableFilterOptions`
	Filter options to be provided to the user. See `FilterOptions` description

`@Input()` **`similarityMatrix?:`** **`RelatedSalesCasesSimilarityMatrix`**	Contains the data to be used in comparison page. It should be set to `undefined` if the table needs to be shown. See `RelatedSalesCasesSimilarityMatrix` description

`@Output()`
`tableUpdate:`
`EventEmitter<TableUpdateEvent>`
	Emits when any parameter (e.g. sort column/direction, page size, etc.) that may affect the data to be displayed on the table changes. See `TableUpdateEvent` description

`@Output()`
`tableRowClick:`
`EventEmitter<TableRowClickEvent<TData>
	undefined>`	Emits when a row is clicked. The user of the component is expected to set `similarityMatrix` based on the clicked row. If the “back” button is clicked on similarity page, an `undefined` value is emitted. See `TableRowClickEvent` description

Classes/Interfaces

FilterOptions
Contains filter options to be presented to the user.

Properties	
`title: string`	Title of the filter popup.
`options: { title: string; checkboxes: string[]; }[]`	An array composed of - Title of the option groups, and - An array of filter options
RelatedSalesCasesSimilarityMatrix	
Contains comparison data.	
Properties	
`fieldDefinitions: { label: string, fieldName: string, }[]`	Defines the fields to be presented in comparison page with the labels to be used for the fields as well as the field/attribute names.
`currentInquiry: { title: string, fields: Map<string, string | number> }[]`	Title to be used with the current inquiry section as well as the values for the current inquiry.
`comparedInquiries: { title: string, fields: Map< string, { value: string | number, similarity?: number } > }[]`	Data for the inquiries to be compared with. It includes the title for each compared inquiry (or sales case), the value for each field as well as the similarities.
RelatedSalesCasesTableColumnDefinition extends TableColumnDefinition	
Contains column definitions.	
Properties	
`title: string`	Title of the column
`fieldName: string`	Name of the field in `TData` to be used to fill the column.
`isSortable: boolean`	Whether the data can be sorted by the column.
`width?: string`	Preferred width of the column.
`isIcon?: boolean`	Whether the column should display an icon instead of text.
`isSimilarity?: boolean`	Whether the column is used for similarity.
TablePageData<TData>	
Data to be rendered on the table.	
Properties	

`dataSlice: TData[]`	Array of `TData` to be presented as the rows of the table. Should contain "page size" number of rows. The mapping between the `TData` field and column is defined by `columns` input field of the table.
`sliceIndex: number`	The slice index (i.e. page index) of the provided data. For example, if the data is the second slice/page of the whole data, then it should be 1.
`totalLength: number`	Total number of data rows.
TableUpdateEvent	
Represents a table update event, which is emitted whenever something affecting displayed data is changed. Once the table user receives the event, it should update the `tablePageData` accordingly.	
Properties	
`pageIndex: number`	Index of page to be displayed.
`pageSize: number`	Size (row count) of the page to be displayed.
`sortColumn?: string`	Column according to which the data is to be sorted.
`sortDirection?: "asc" | "desc"`	Sorting direction. Either asc(ening) or desc(ending).
`search?: string`	Search terms entered by the user.
`filter?: any`	Filter options selected by the user.
TableRowClickEvent<TData>	
Represents a table row click event.	
Properties	
`index: number`	Index of the row that has been clicked. Should be between 0 and (`pageSize` - 1).
`data: TData`	Data that is displayed on the clicked row.

2.4.3 Examples

Use of Related Sales Cases component as shown in Table 9 and Table 10 results a view similar to one given in Figure 8 and Figure 9.

Table 9. Related Sales Cases Use Example (.html)

```
<innosale-related-sales-cases
  [columns]="columns"
  [pageSizeOptions]="[10, 25, 100]"
  [pageSize]="10"
  [filterOptions]="filterConfig"
  [tablePageData]="tablePageData"
  (tableUpdate)="tableUpdate($event)"
  (tableRowClick)="tableRowClick($event)"
```

```
   [similarityMatrix]="similarityMatrix"
></innosale-related-sales-cases>
```

Table 10. Related Sales Cases Use Example (.ts)

```
interface InnoSaleRelatedSalesCase extends SalesCaseListItem {
  id?: string | undefined;
  customer?: string | undefined;
  customerId?: string | undefined;
  description?: string | undefined;
  date?: string | undefined;
  region?: string | undefined;
  category?: string | undefined;
  status?: string | undefined;
  highlighted?: boolean | undefined;
  alreadyCustomer?: boolean | undefined;
  similarity?: number;
}

@Component({
  selector: 'app-inquiry-detail-related-sales-cases',
  templateUrl: './inquiry-detail-related-sales-cases.component.html',
  styleUrls: ['./inquiry-detail-related-sales-cases.component.css']
})
export class InquiryDetailRelatedSalesCasesComponent {
  columns: RelatedSalesCasesTableColumnDefinition[] = [];
  tablePageData: TablePageData<InnoSaleRelatedSalesCase> = new TablePageData();

  similarityMatrix?: RelatedSalesCasesSimilarityMatrix;

  constructor(
    private router: Router,
    public dialog: MatDialog,
    private _inquiriesService: InquiriesService
  ) {
    this.columns = [
      { title: 'Case Id', fieldName: 'id', isSortable: true },
      { title: 'Customer', fieldName: 'customer', isSortable: true },
      { title: 'Customer Id', fieldName: 'customerId', isSortable: true },
      { title: 'Description', fieldName: 'description', isSortable: false },
      { title: 'Date', fieldName: 'date', isSortable: true },
      { title: 'Region number', fieldName: 'region', isSortable: true },
      { title: 'Product Category', fieldName: 'category', isSortable: true },
      { title: 'Price', fieldName: 'price', isSortable: true },
      { title: 'Price related to present time', fieldName: 'pricePresentTime',
isSortable: true },
      { title: 'Similarity', fieldName: 'similarity', isSortable: false,
isSimilarity: true },
    ];
  }

  tableRowClick(clickEvent: TableRowClickEvent<SalesCaseListItem> | undefined,) {
    if (clickEvent === undefined) {
      this.similarityMatrix = undefined;
```

```
    } else {
      this.similarityMatrix = {
        fieldDefinitions: [
          { label: "Field 1", fieldName: "field1" },
          { label: "Field 2", fieldName: "field2" },
          { label: "Field 3", fieldName: "field3" },
          { label: "Field 4", fieldName: "field4" },
          { label: "Field 5", fieldName: "field5" },
          { label: "Field 6", fieldName: "field6" },
          { label: "Field 7", fieldName: "field7" },
          { label: "Field 8", fieldName: "field8" },
        ],
        currentInquiry: {
          title: "Current Inquiry",
          fields: new Map([
            ["field1", "Current field1"],
            ["field2", "Current field2"],
            ["field3", "Current field3"],
            ["field5", "Current field5"],
            ["field6", "Current field6"],
            ["field7", "Current field7"],
            ["field8", "Current field8"],
          ])
        },
        comparedInquiries: [
          {
            title: "Inquiry " + clickEvent.data.id,
            fields: new Map([
              ["field1", { similarity: 0.7, value: "Inquiry " +
clickEvent.data.id + " field1" }],
              ["field2", { similarity: 0.7, value: "Inquiry " +
clickEvent.data.id + " field2" }],
              ["field3", { similarity: 0.3, value: "Inquiry " +
clickEvent.data.id + " field3" }],
              ["field4", { value: "Inquiry " + clickEvent.data.id + " field4" }],
              ["field5", { similarity: 0.5, value: "Inquiry " +
clickEvent.data.id + " field5" }],
              ["field7", { similarity: 0.1, value: "Inquiry " +
clickEvent.data.id + " field7" }],
              ["field8", { similarity: 0.9, value: "Inquiry " +
clickEvent.data.id + " field8" }],
            ])
          }
        ]
      }
    }
  }

  tableUpdate(event: TableUpdateEvent) {
    this.updateInquiryList(event);
  }

  updateInquiryList(event?: TableUpdateEvent) {
    this._inquiriesService.getInquiries(
      event?.search,
```

```
      event?.sortColumn,
      event?.sortDirection,
      event?.pageIndex,
      event?.pageSize,

      // TODO: Enhance the API to set/accept filter parameters on a proper manner
      // event.filter,
      event?.filter["Regions"],
      event?.filter["Status"],
      event?.filter["Product Category"]

      ).subscribe((d) => {
        this.tablePageData = {
          dataSlice: d.data,
          sliceIndex: d.pageIndex,
          totalLength: d.totalLength,
        };

        // randomly generate similarity data
        for (let item of this.tablePageData.dataSlice) {
          item.similarity = Math.random();
        }
      }
    );
  }

   //------- Filter -------
   filterConfig = {
    title: 'Filters',
    options: [
      {
        title: 'Regions',
        checkboxes: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
      },
      {
        title: 'Product Category',
        checkboxes: ['A', 'B', 'C']
      },
    ]
  };
}
```

Search... Filter

Case Id	Customer	Customer Id	Description	Date	Region number	Product Category	Price	Price related to present time	Similarity
098292	Customer 75	113	KBK	2020-02-14T00:00:00	1	A			
014212	Customer 74	111	Jib	2020-02-14T00:00:00	0	B			
077555	**Customer 02**	**3**	**Hoist for KBK**	**2020-02-14T00:00:00**	**3**	**A**			
042367	Customer 11	17	KBK	2020-02-15T00:00:00	4	A			
038724	**Customer 45**	**68**	**Jib**	**2020-02-16T00:00:00**	**6**	**A**			
072754	Customer 05	8	Hoist for KBK	2020-02-13T00:00:00	1	C			
057889	Customer 00	0	KBK	2020-02-21T00:00:00	4	C			
046122	**Customer 59**	**89**	**Jib**	**2020-02-22T00:00:00**	**2**	**B**			
036758	Customer 58	87	Hoist for KBK	2020-02-22T00:00:00	1	A			
018504	Customer 87	131	KBK	2020-02-21T00:00:00	4	A			

Items per page: 10 1 – 10 of 84

Figure 8. Related Sales Cases Example Usage (Table)

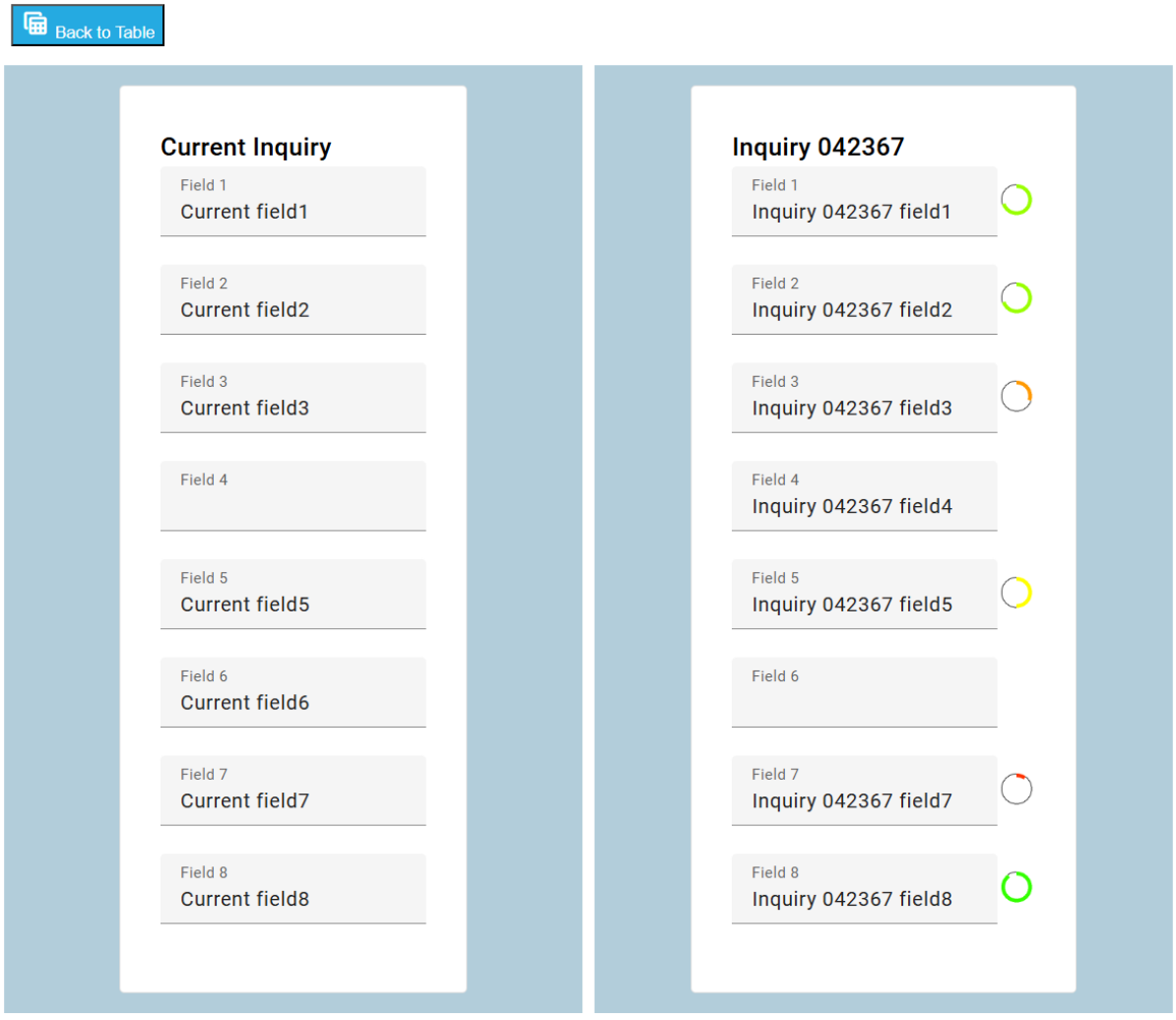

Figure 9. Related Sales Cases Example Usage (Comparison Page)

2.5 Price Estimation

2.5.1 Price Estimation User Interface

The Pricing Calculation page is an element of a my multi-page Web-application, which utilizes fuzzy logic for dynamic pricing. For that purpose it is interfaced to the Fuzzy Control Language Engine (FCLE), which has been developed in scope of T3.5. The UI component includes several parts. The page displays input fields for various pricing factors and a calculation of a final pricing factor can be performed based on the entered data. While the system can calculate the pricing factor, you need to input the current baseline price for the total product. Then, the following is calculated:

- The Fuzzy Pricing Factor: This value is calculated using fuzzy logic rules applied to the provided inputs.
- Final Price Recommendation: A suggested final price is generated by multiplying the current baseline price with the fuzzy pricing factor, providing a dynamic and data-driven pricing recommendation.

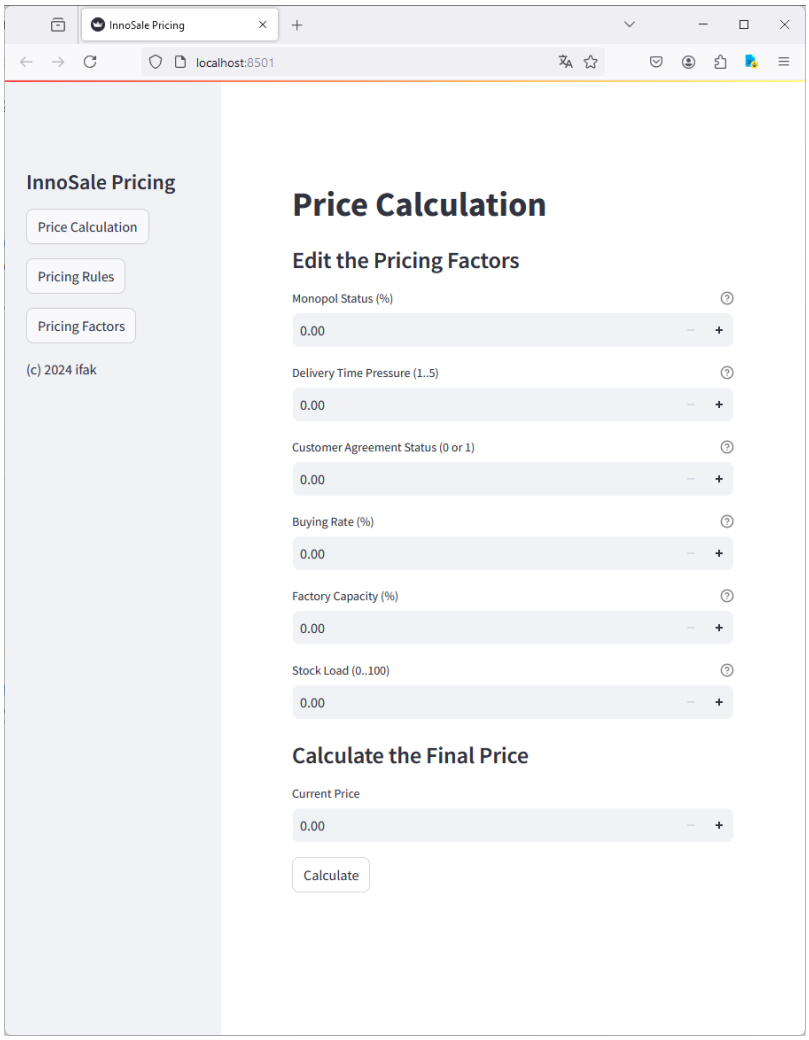

Figure 10: Price Estimation User Interface

2.5.2 Component for Declaration of Pricing Parameters

The component for declaration of pricing factors is a further element of the my multi-page Web-application, which utilizes fuzzy logic for dynamic pricing. It allows users to view and manage various pricing parameters that impact product delivery and customer agreements (see Figure 11). Initially, it displays a set of default factors, each with a name, variable name, unit, and description. Users can edit or delete existing parameters, with changes triggering an automatic rerender of the displayed list (see Figure 12). Additionally, users can add new factors or modify existing ones through a form, where they can specify details like the name, variable name, unit, and description. The page efficiently handles session state, ensuring that edits, deletions, and new entries are consistently updated and reflected on the screen and in other pages of the application.

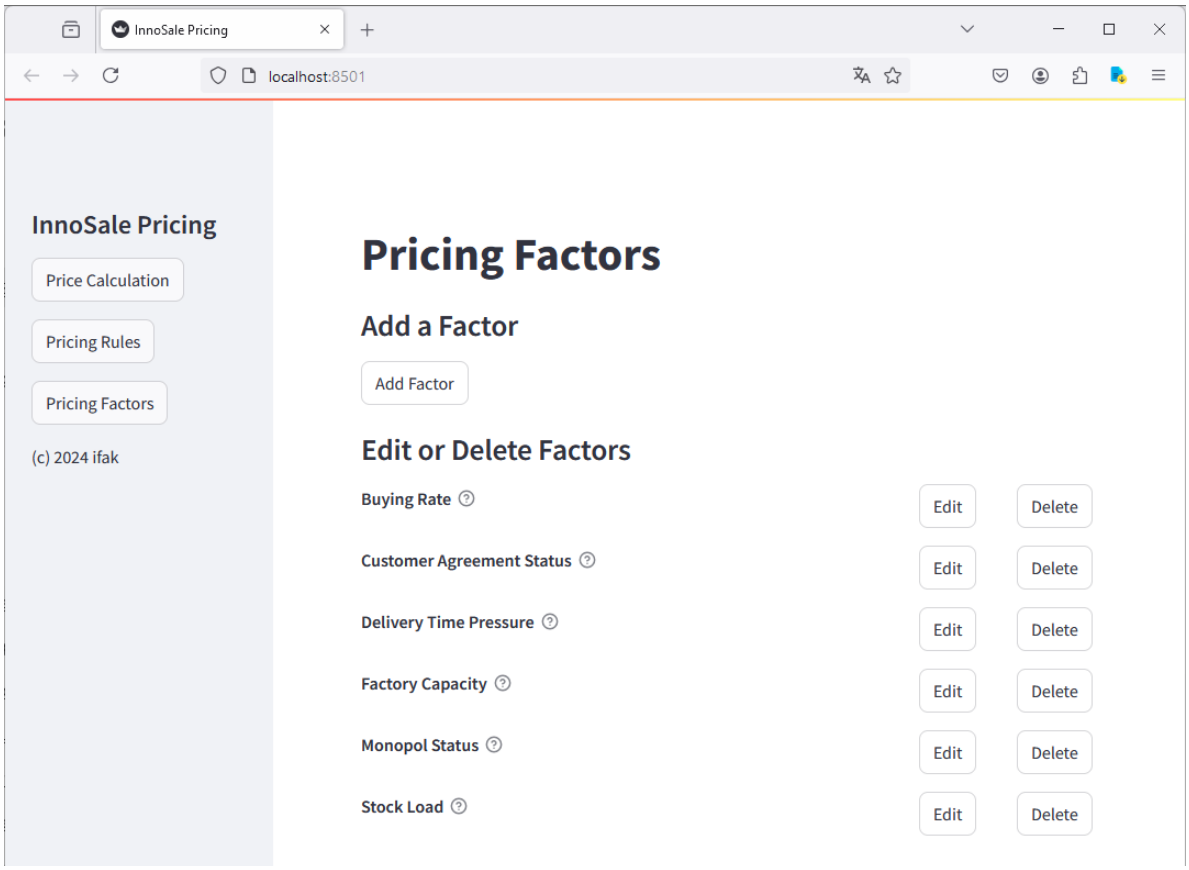

Figure 11: Component for Declaration of Pricing Parameters

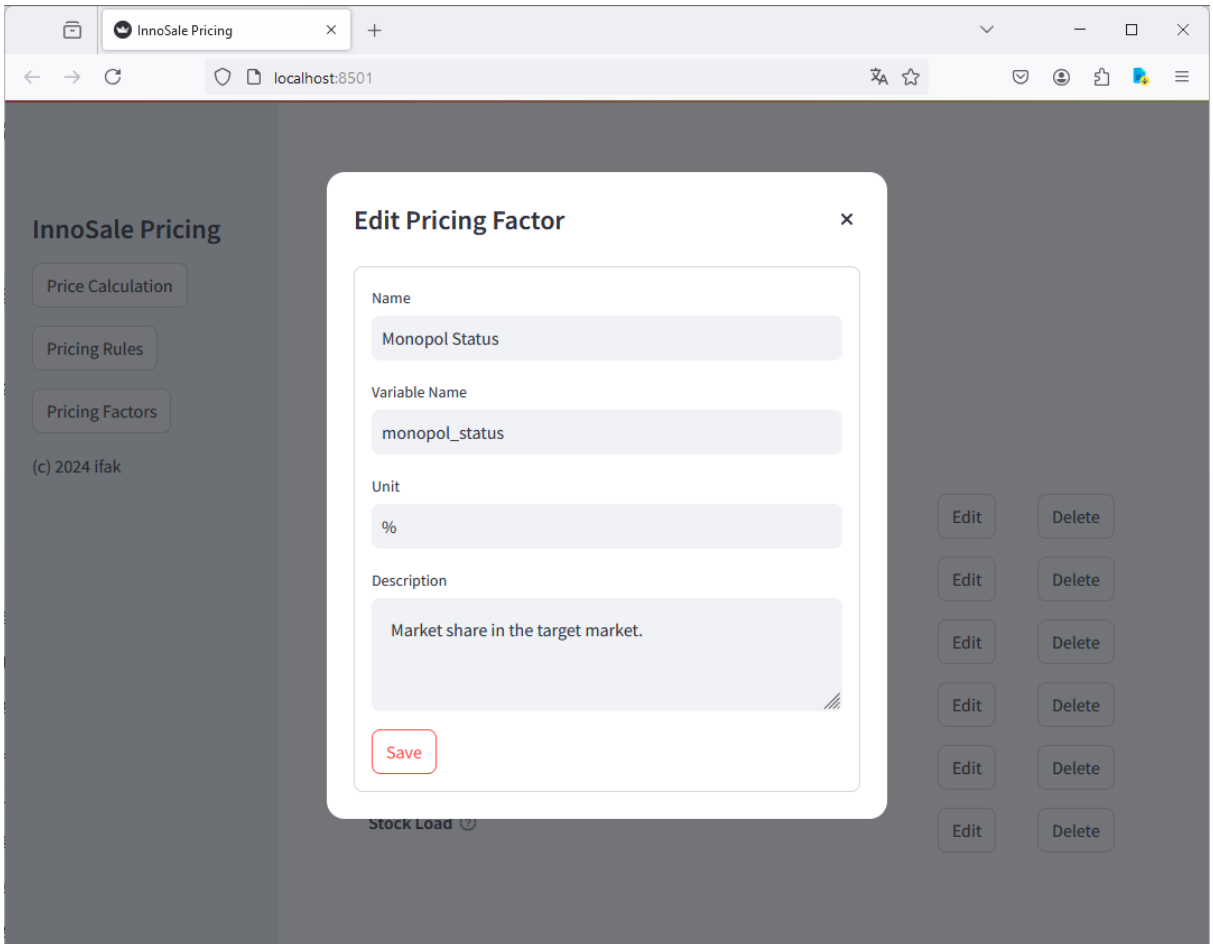

Figure 12: Editing a Pricing Factor

2.6 Pricing Factors List Component

2.6.1 Overview

The Pricing Factors Component is an Angular standalone component designed to display pricing factors in a tabular format with the ability to trigger a recalculation based on user input. The component allows localization and customization of labels as well as button texts. It can displays multiple sets of pricing factors at once.

2.6.2 API

```
import {
  PricingFactorsComponent,
  PricingFactor,
  PricingFactorsLocalization
} from 'pricing-factors';
```

Directives

PricingFactorsComponent	
Directive: pricing-factors	
Properties	

`@Input() pricingFactorsList: PricingFactor[][]`	Array of array of PricingFactor interface which contains sub tables row types
`@Input() localization: PricingFactorsLocalization`	Localization interface of the PricingFactors component (If not set, it will default to English)
`@Output() recalculateTriggered: EventEmitter`	Emits when recalculate button triggered

Classes/Interfaces

PricingFactor	
This class defines the data to be displayed for each row in a pricing table, representing different pricing factors.	
Properties	
`factor_set: string`	Represents the specific factor set or category being evaluated (e.g., region, product type).
`price: number`	Displays the base price associated with the factor set.
`adjust: number`	Shows the adjustment value applied to the base price, which could be positive or negative.
`percent: number`	Indicates the percentage adjustment applied to the price, reflecting the factor's influence.
PricingFactorsLocalization	
This class provides the localized labels and text used across the pricing component, allowing for translation and customization based on different locales.	
Properties	
`tableColumnTitleFactorSet: string`	The localized title for the "Factor Set" column in the table.
`tableColumnTitlePrice: string`	The localized title for the "Price" column in the table.
`tableColumnTitleAdjust: string`	The localized title for the "Adjustment" column in the table.
`tableColumnTitlePercent: string`	The localized title for the "Percentage" column in the table.
`buttonTitleRecalculate: string`	The label for the "Recalculate" button, which triggers recalculation of prices based on new factors.
`headerTitle: string`	The localized title for the header of the pricing table or component.

2.6.3 Examples

Table 11. InnoSale Pricing Factors Component Use Example (.html)

```
<div style="height: 100%; width: 100%;">
    <pricing-factors
        [pricingFactorsList]="pricingFactorsList"
        [localization]="pricingFactorsLocalization"
        (recalculateTriggered)="recalculateTriggered()"
    ></pricing-factors>
</div>
```

Table 12. InnoSale Pricing Factors Component Use Example (.ts)

```
@Component({
  selector: 'app-inquiry-detail-pricing',
  standalone: true,
  imports: [PricingFactorsComponent],
  templateUrl: './inquiry-detail-pricing.component.html',
  styleUrls: ['./inquiry-detail-pricing.component.css'],
})
export class InquiryDetailPricingComponent extends InquiryDetailChild {
  pricingFactorsList: PricingFactor[][] = [
    [
      { factor_set: "Factor #1", price: 0, adjust: -1.3, percent: 0.3 },
      { factor_set: "Factor #2", price: 0, adjust: -1.3, percent: 0.3 },
      { factor_set: "Factor #3", price: 0, adjust: -1.3, percent: 0.3 },
      { factor_set: "Factor #4", price: 0, adjust: -1.3, percent: 0.3 },
    ],
    [
      { factor_set: "Factor #1", price: 3, adjust: -8.9, percent: 0.8 },
      { factor_set: "Factor #2", price: 3, adjust: -8.9, percent: 0.8 },
      { factor_set: "Factor #3", price: 3, adjust: -8.9, percent: 0.8 },
    ]
  ]

  pricingFactorsLocalization: PricingFactorsLocalization = {
    tableColumnTitleFactorSet: "Faktorensatz",
    tableColumnTitlePrice: "Preis",
    tableColumnTitleAdjust: "Anpassen",
    tableColumnTitlePercent: "Prozent",
    buttonTitleRecalculate: "Neu Berechnen",
    headerTitle: "Berechnungsdetails"
  }

  recalculateTriggered() {
    alert("Triggered: 'buttonRecalculate' function!")
  }
}
```

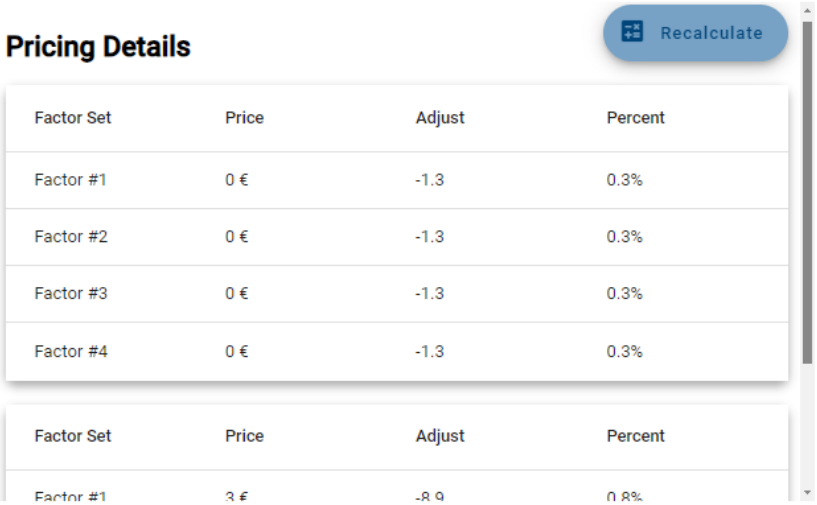

Figure 13. InnoSale Pricing Factors Component Example Usage

2.7 Guided Selling Component

Guided Selling Component contains the planning of a factory layout in 3D and a set of questions and rules based on which the component proposes products for the user that best match the customer's need. The questions are formulated in such a way that a purchasing customer having no detailed technical knowhow of the cranes can understand and answer them. In connection with the guided selling questions, the user can create a visual representation of the factory layout in 3D environment.

Guided selling component includes the following functionalities and information:

- Defining the size of the factory into which the cranes are needed.
- Creating desired number of working stations by drag & dropping them to desired locations in a 3D factory layout. Working station is a specific area within the factory where products are being manufactured and where cranes are needed to lift/lower the load.
- Defining the size of working stations where the cranes are being operated.
- Defining the locations of pick and drop points inside the working station from where the loads are lifted by a crane.
- Automatically calculating the distances between different working stations and pick & drop points.
- Defining the volitional route and intensity of the material flow according to which the loads are moved between working stations inside the factory environment. Alternatively, the component can optimize the route and material flow according to shortest possible route between pick & drop points from one working station to another in the order they were created.
- Defining the size and weight of the loads being lifted.
- Defining customer requirements for load handling, e.g. if sway control and turning the load up to 180 degrees are needed.
- Proposing and adding cranes into the factory layout in connection with working stations.

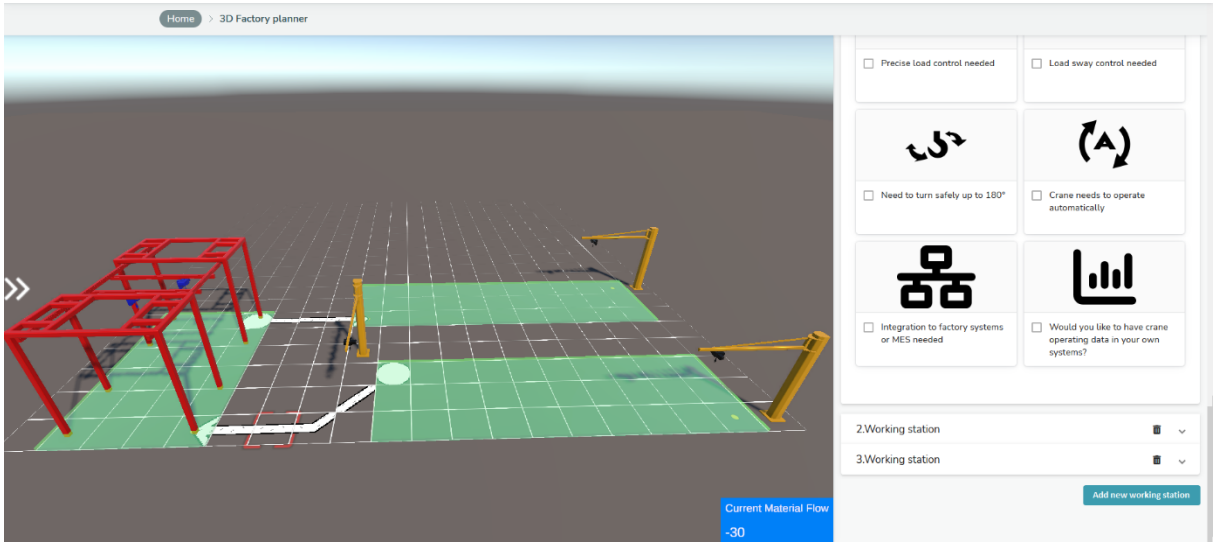

Figure 14. User Interface of Guided Selling Component in 3D view.

Because the dimensions, weight and other requirements of the load being manufactured may vary in different locations within the factory, the guided selling questions are working station specific. Thus, based on rules reflecting the input from the user, the component may propose different cranes for different working stations. When a crane is proposed to a specific working station, user is then given a possibility to drag & drop the crane to desired location in the factory layout.

The camera view in the 3D environment can be rotated, moved, zoomed in/out, and changed between 3D/2D angle of view. The 3D view can also be opened in full screen mode. When a working station or added crane is selected in 3D environment, the 3D object is highlighted with blue color and the corresponding guided selling questions appear on the screen. The user interface of guided selling questions is implemented with Angular technology. The user interface of 3D visualization is implemented with Unity game engine. The 3D area covers approximately 70 % of the screen width on the left and guided selling questions about 30 % of screen width on the right.

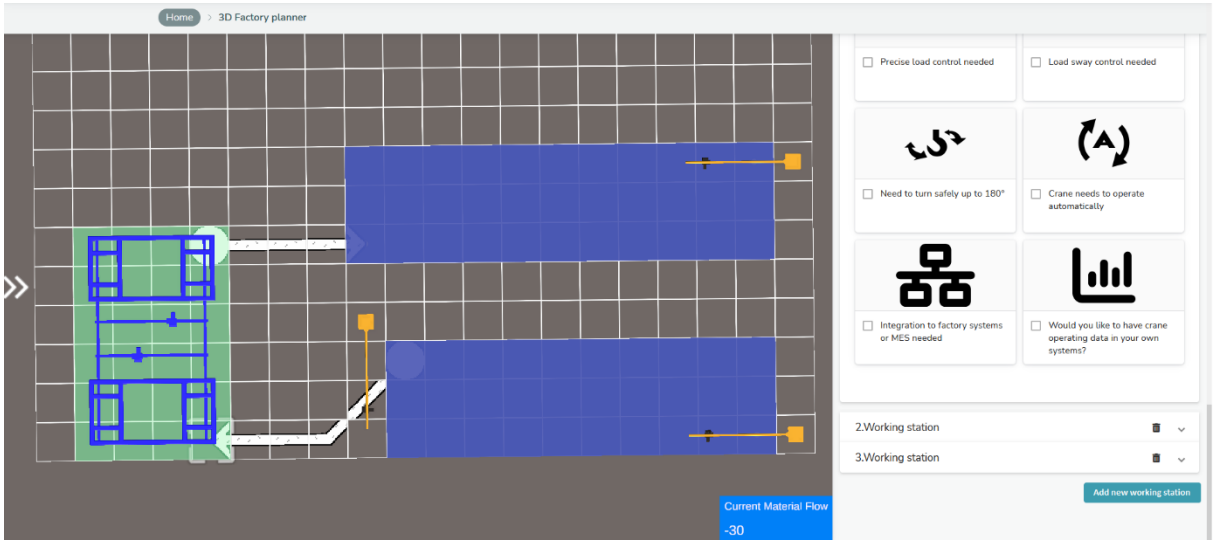

Figure 15. User Interface of Guided Selling Component in 2D view.

2.8 AI Assisted Product Proposal Component

This component was implemented using two different frameworks due to different environment that they would be used. One is in Angular[3] and the other is in Vaadin[4]. Respective descriptions are given in the subsequent sections.

2.8.1 Angular Implementation

2.8.1.1 Overview

The AI Assisted Product Proposal Component is an Angular component designed to display and manage a table of AI-assisted product proposals. It provides users with the ability to accept or decline these proposals based on AI feedback. The component allows localization, customization of labels as well as button texts.

2.8.1.2 API

```
import {
  AiAssistedProduct,
  AiAssistedProductProposalComponent,
  AiAssistedProductProposalLocalization
} from 'ai-assisted-product-proposal';
```

Directives

AiAssistedProductProposalComponent	
Directive: ai-assisted-product-proposal	
Properties	

[3] https://angular.dev/

[4] https://vaadin.com/

`@Input()`
`data:`
`AiAssistedProduct[]`
	Array of AiAssistedProduct interface which contains the tables row types

`@Input()`
`localization:`
`AiAssistedProductProposalLocalization`
	Localization interface of the AiAssistedProductProposal component (If not set, it will default to English)

`@Output()`
`aiAssistedProductAcception:`
`EventEmitter<{`
`aiFeedbackStatus: boolean[],`
`accepted: boolean`
`}>`
	Emits the row checks representing in the table

Classes/Interfaces

AiAssistedProduct	
This object defines the data to be displayed in each row of the table within the AI-assisted product proposal.	
Properties	
`parameter: string`	Represents the name or type of parameter being evaluated or displayed.
`value: string`	Displays the corresponding value or data associated with the parameter.
`accuracy_percentage: number`	Indicates the percentage of accuracy for the displayed value, reflecting the confidence level.
`ai_feedback: boolean`	Shows whether AI feedback has been provided for this row, typically as a true/false indicator.
AiAssistedProductProposalLocalization	
This object provides the localized labels and text used across the component, enabling translations or adjustments for different languages or contexts.	
Properties	
`tableColumnTitleParameter: string`	The localized title for the "Parameter" column in the table.
`tableColumnTitleValue: string`	The localized title for the "Value" column in the table.
`tableColumnTitleAccuracy: string`	The localized title for the "Accuracy Percentage" column in the table.
`tableColumnTitleAiFeedback: string`	The localized title for the "AI Feedback" column in the table.
`buttonTitleAccept: string`	The label for the "Accept" button, allowing users to confirm or approve a proposal.
`buttonTitleDecline: string`	The label for the "Decline" button, allowing users to reject a proposal.

2.8.1.3 Examples

Table 13. InnoSale AI Assisted Product Proposal Component Use Example (.html)

```
<div style="height: 100%; width: 100%;">
    <ai-assisted-product-proposal
        [data]="aiAssistedProducts"
        [localization]="aiAssistedProductProposalLocalization"
        (aiAssistedProductAcception)="handleData($event)"
    ></ai-assisted-product-proposal>
</div>
```

Table 14. InnoSale AI Assisted Product Proposal Component Use Example (.ts)

```
@Component({
  selector: 'app-inquiry-ai-assisted-product-proposal',
  standalone: true,
  imports: [AiAssistedProductProposalComponent],
  templateUrl: './inquiry-ai-assisted-product-proposal.component.html',
  styleUrl: './inquiry-ai-assisted-product-proposal.component.css'
})

export class InquiryAiAssistedProductProposalComponent {
  aiAssistedProducts: AiAssistedProduct[] = [
    { parameter: 'Parameter1', value: 'Value1', accuracy_percentage: 30,
ai_feedback: false },
    { parameter: 'Parameter2', value: 'Value2', accuracy_percentage: 30,
ai_feedback: false },
    { parameter: 'Parameter3', value: 'Value3', accuracy_percentage: 30,
ai_feedback: false },
    { parameter: 'Parameter4', value: 'Value4', accuracy_percentage: 30,
ai_feedback: false },
    { parameter: 'Parameter5', value: 'Value5', accuracy_percentage: 30,
ai_feedback: false },
    { parameter: 'Parameter6', value: 'Value6', accuracy_percentage: 30,
ai_feedback: false },
  ];

  aiAssistedProductProposalLocalization: AiAssistedProductProposalLocalization =
{
    tableColumnTitleParameter: "Parameter",
    tableColumnTitleValue: "Wert",
    tableColumnTitleAccuracy: "Genauigkeit (%)",
    tableColumnTitleAiFeedback: "KI-Feedback",
    buttonTitleAccept: "Akzeptieren",
    buttonTitleDecline: "Abfall",
  }

  handleData(result: { aiFeedbackStatus: boolean[], accepted: boolean }) {
    const { aiFeedbackStatus, accepted } = result
    console.log(aiFeedbackStatus);
    console.log(accepted);
  }
}
```

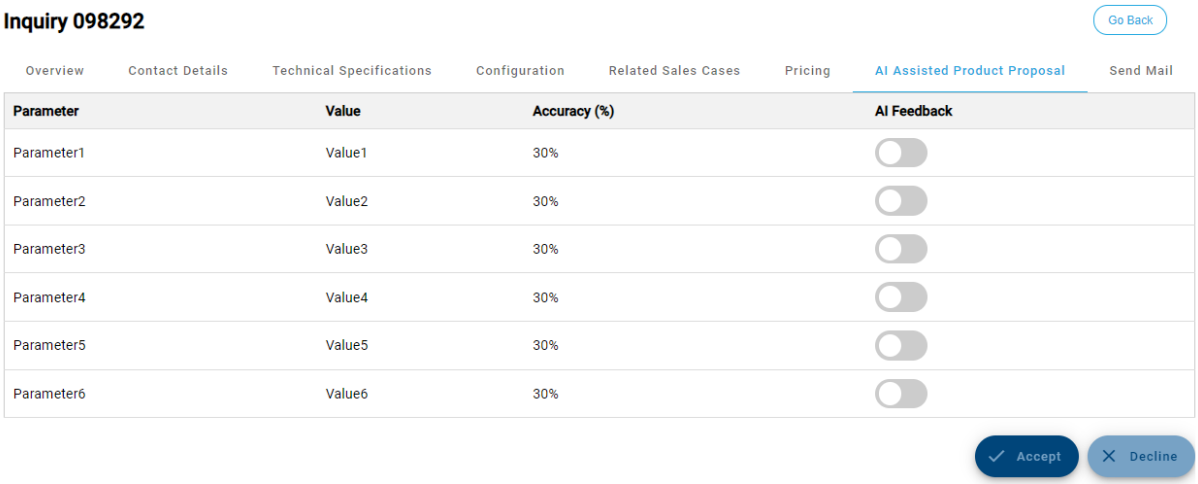

Figure 16. InnoSale AI Assisted Product Proposal Component Example Usage

2.8.2 Vaadin Implementation

AI Assisted Product Proposal Component outlines for the sales user the product configuration that is proposed by the AI algorithm. The purpose is to find best fitting product configuration that matches the customer's geographical area when making a new offer for the customer. The component contains the following information:

- **Offer ID:** Offer ID of the previous offer received from and proposed by the AI algorithm. The proposed product configuration for the new offer is taken from this previous offer that best matches the need of a new customer.
- **Pos:** Position number of the offer items.
- **Parameter:** A product configuration parameter which varies based on what products the AI algorithm is proposing.
- **Value:** Value of the product configuration parameter. The value is always a configuration specific information.
- **Feedback:** A possibility for the user to give feedback per parameter on how well each proposed configuration parameter matches the need of a new customer to whom the offer is being made. The feedback is given by ticking the checkboxes in each row of the proposed product configuration. Ticked checkbox signifies that the value matches well the customer need. By default, all checkboxes are unticked.
- **3D image of the product:** Shows a 3D model of the configured product as it were in the previous offer from which the AI algorithm has taken the product configuration. The 3D model is fetched from Summium® CPQ database.
- **Yes/Cancel buttons:** Yes/Cancel buttons from which the user can either approve or reject the proposed product configuration. If approved, the view is transferred automatically to product configurator to finalize the product configuration. If rejected, the AI Assisted Product Proposal Component is closed, and the user can start the product configuration from the beginning.

The component is opened inside Wapice's Summium® CPQ sales configurator and is developed using existing Summium® CPQ technologies. The product configuration parameters and their values are fetched from Summium® CPQ database based on the offer ID received from the AI algorithm. User interface of the component is developed using Vaadin UI framework.

AI Assisted Product Proposal Component is illustrated in a figure below. Due to reasons of confidentiality, some information has been hidden from the figure.

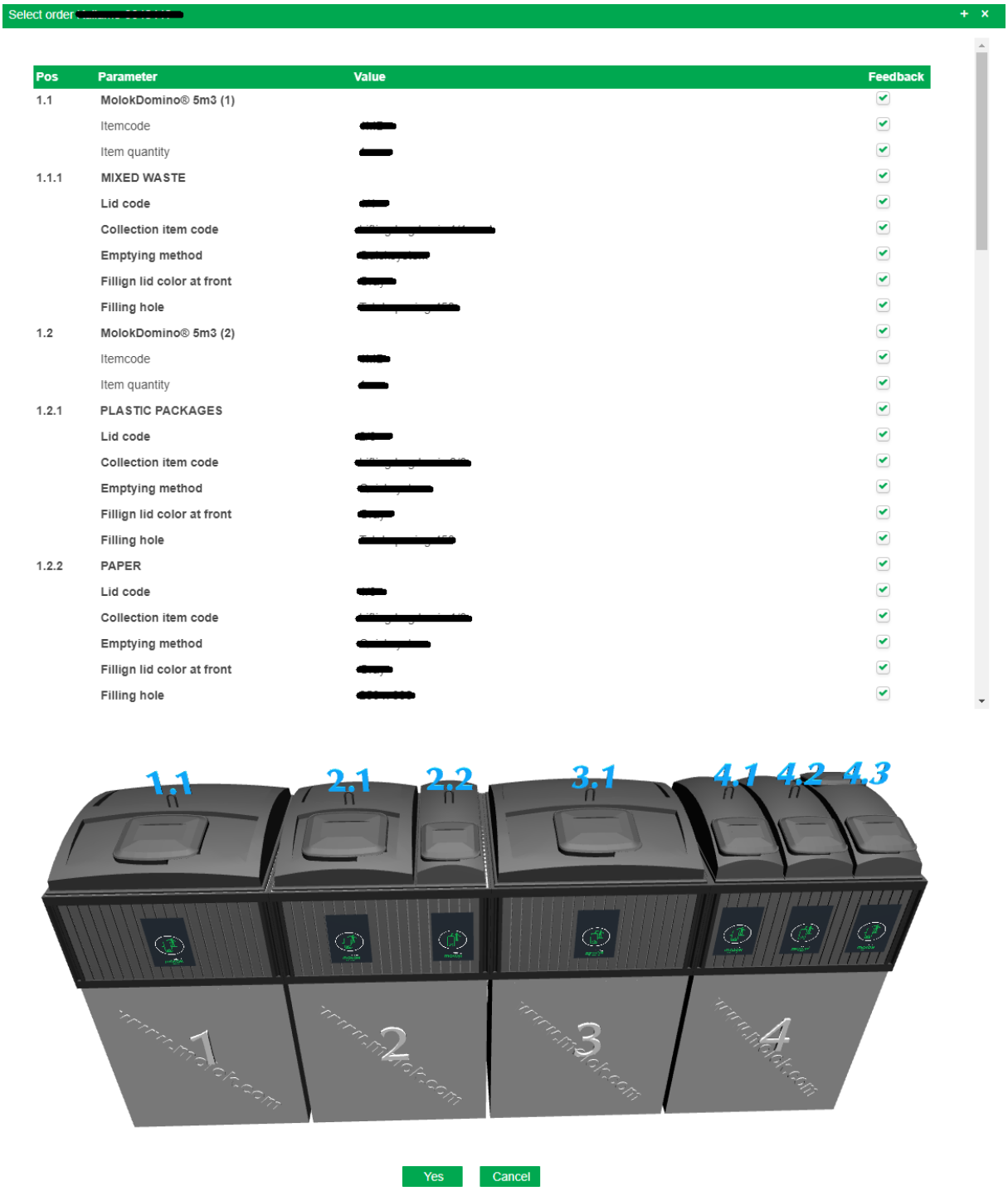

Figure 17. Illustration of AI Assisted Product Proposal Component. Please note that only some of the position rows are visible in the figure.

2.9 Data Visualization Component

2.9.1 Overview

The **Data Visualization Component** in the InnoSale solution is designed to visually present data to users in various graphical formats (charts, graphs, etc.). It supports multiple visualization types such as bar charts, line charts, pie charts, and more. This component can

be reused across different domains where data needs to be visualized and allows for customization depending on the data format, visualization type, and user preferences.

The component solves the following use cases:

- Provides visual representation of sales data, customer interactions, performance metrics, and other business intelligence data.
- Supports different types of charts such as bar charts, line charts, pie charts, and scatter plots.
- Allows customization of data labels, axis labels, tooltips, legends, and themes to match the branding.
- Supports data interaction such as zooming, filtering, and drilling down into data points, enhancing user engagement.
- Responsive design that adapts to various screen sizes and devices.
- Ensures accessibility by supporting screen readers and other assistive technologies.
- Integrates with real-time data to support live dashboards and dynamic updates.
- Provides exporting options for charts and graphs in formats such as PNG, PDF, and SVG.
- Optimized for performance, handling large datasets smoothly.

These enhancements ensure the component is versatile, interactive, and capable of meeting diverse visualization requirements in different application contexts.

2.9.2 API

```
import { InnosaleModule } from 'reusable-components';
```

Directives

DataVisualizationComponent	
Selector: innosale-data-viz	
Properties	
`@Input() chartType: string`	Defines the type of chart to be displayed (e.g., bar, line, pie, scatter).
`@Input() chartData: ChartData`	Contains the data to be visualized. The structure includes labels and datasets.
`@Input() chartOptions: ChartOptions`	Defines various options to customize the chart, such as display options for tooltips, legend, axis labels, grid lines, etc.
`@Input() height: string`	These inputs determine the size of the chart in pixels or percentages (e.g., '400px', '100%').
`@Input() width: string`	

`@Input()`
`colors:`
`Color[]`
	Used to define the colors for the chart. Optionally provide an array of color codes for the different datasets or chart sections.

`@Input()`
`exportEnabled:`
`boolean`
	Enables the option for exporting the chart (e.g., to PNG or PDF). If true, the chart can be exported to an image file.

`@Output()`
`chartClick:`
`EventEmitter<ChartClickEvent>`
	Emits when the user clicks on a chart element (e.g., data point or legend). The emitted event contains information about the clicked element.

Classes/Interfaces

ChartData	
Represents the data to be visualized in the chart.	
`labels: string[]`	Array of labels for the chart (e.g., months, categories).

`label: string`
`data: number[];`
`backgroundColor?: string`
`borderColor?: string`
	An array composed of:
	- Label for the dataset
	- Data points corresponding to each label
	- Background color for the chart elements
	- Border color for line charts

ChartOptions	
Configuration options to customize the chart appearance and behavior.	
`responsive?: boolean`	Whether the chart should adjust its size to fit its container.
`legend?: { display: boolean }`	Show or hide the legend.

`scales?: {`
`  xAxes: AxisOptions[];`
`  yAxes: AxisOptions[];`
`}`
	Configuration for x and y axes.

`tooltips?: { enabled: boolean }`	Show or hide tooltips.
AxisOptions	
Defines options for axes (x or y)	
`display: boolean`	Whether to show the axis or not.
`labelString?: string`	Axis label.
`ticks?: { beginAtZero?: boolean }`	Tick options, like forcing axis to start at zero.
ChartClickEvent	
Represents the event triggered when the user clicks on a chart element.	
`elementIndex: number`	Index of the clicked element.
`datasetIndex: number`	Index of the dataset.
`data: any`	Data associated with the clicked element.

2.9.3 Examples

2.9.3.1 Data Input Information

Chart Data (chartData): This input holds the actual data that the chart will display. It includes labels for the x-axis (like dates or categories) and datasets, which are the y-axis values. Developer also define colors here.

```
{
  labels: ['January', 'February', 'March'],
  datasets: [
    {
      label: 'Revenue',
      data: [15000, 22000, 18000],
      backgroundColor: '#2196F3'
    },
  ]
}
```

Chart Options (chartOptions): This input holds configuration for customizing the chart's appearance, including settings for the axis, tooltips, and legends.

Below is an example for a bar chart options with custom axis labels and grid lines:

```
const chartOptions: ChartOptions = {
  responsive: true,  // Chart will resize based on the container
  legend: {
    display: true,   // Show the legend
    position: 'top', // Legend will be displayed at the top
  },
  tooltips: {
    enabled: true,   // Enable tooltips to display when hovering over data
points
  },
  scales: {
    xAxes: [
      {
        display: true, // Show the x-axis
        labelString: 'Months', // Label for the x-axis
        gridLines: {
          display: false, // Hide grid lines on the x-axis
        },
        ticks: {
          fontSize: 12, // Font size for the x-axis labels
        },
      },
    ],
    yAxes: [
      {
        display: true, // Show the y-axis
        labelString: 'Revenue', // Label for the y-axis
        ticks: {
```

```
          beginAtZero: true,  // Ensure the y-axis starts from zero
          fontSize: 12,       // Font size for the y-axis labels
        },
        gridLines: {
          color: '#e0e0e0',  // Grid line color for the y-axis
        },
      },
    ],
  },
};
```

Below is an example for a line chart with custom tooltips and legend configuration:

```
const data = [
  { date: new Date(2023, 0, 1), value: 30 },
  { date: new Date(2023, 0, 15), value: 43 },
  { date: new Date(2023, 1, 1), value: 50 },
  { date: new Date(2023, 1, 15), value: 54 },
  { date: new Date(2023, 2, 1), value: 42.76 },
  { date: new Date(2023, 2, 15), value: 55 },
  { date: new Date(2023, 3, 1), value: 68 },
  { date: new Date(2023, 3, 15), value: 65 },
  { date: new Date(2023, 4, 1), value: 60 }
];
const chartData: ChartData = {
  labels: data.map(d => d.date), // Use Date objects as labels
  datasets: [
    {
      label: 'Sales Over Time',
      data: data.map(d => d.value),
      borderColor: 'steelblue',
      backgroundColor: 'rgba(70, 130, 180, 0.3)',
    },
  ],
};
const chartOptions: ChartOptions = {
  responsive: true,
  legend: {
    display: true,
    position: 'bottom',
    labels: { fontColor: '#333', fontSize: 14, },
  },
  tooltips: {
    enabled: true,
    mode: 'index',
    intersect: false,
    backgroundColor: '#333',
    titleFontColor: '#fff',
    bodyFontColor: '#fff',
  },
```

```
  scales: {
    xAxes: [
      {
        type: 'time', // Specify type as 'time' to handle Date objects
        time: {
          unit: 'month', // Set the unit to 'month' for better readability
          displayFormats: { month: 'MMM D',},
        },
        display: true,
        labelString: 'Month',
      },
    ],
    yAxes: [
      {
        display: true,
        labelString: 'Revenue (USD)',
        ticks: { beginAtZero: false, min: 30, max: 70, },
      },
    ],
  },
};
<DataVisualizationComponent
  chartType="line"
  [chartData]="chartData"
  [chartOptions]="chartOptions"
  height="400px"
  width="800px"
  [colors]="['steelblue']"
  exportEnabled="true"
></DataVisualizationComponent>
```

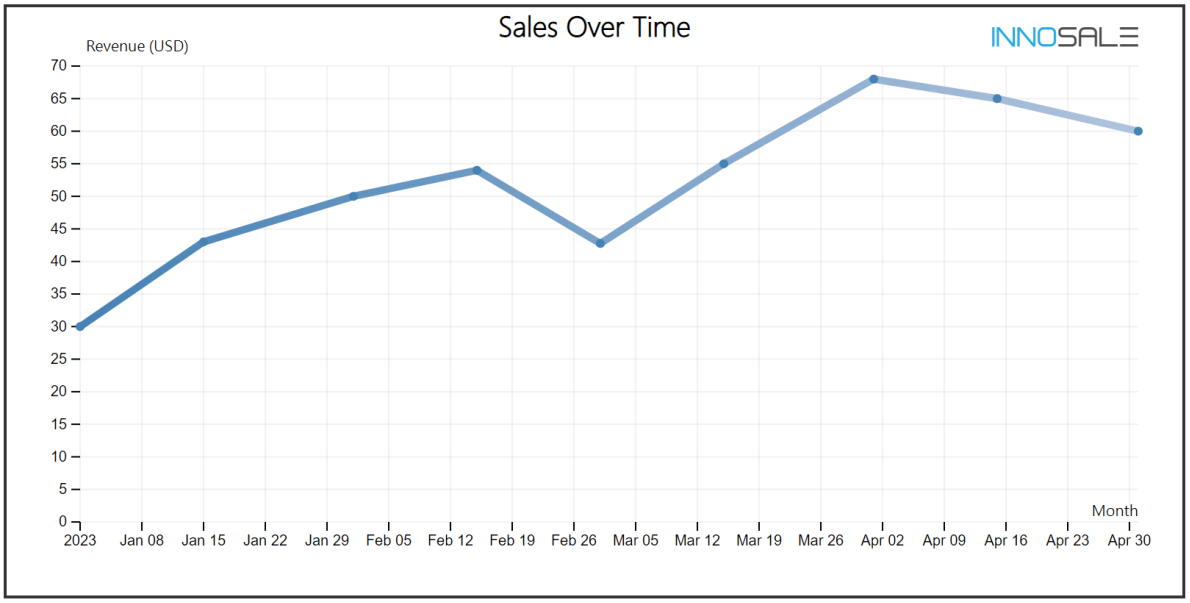

Figure 18. The generated line chart with the example code.

Below is an example for a pie chart options:

```
const chartOptions: ChartOptions = {
  responsive: true, // Adjust chart size according to container size
  legend: {
    display: true,   // Show the legend
    position: 'right', // Legend on the right side
    labels: {
      fontColor: 'black', // Legend font color
    },
  },
  tooltips: {
    enabled: true,   // Enable tooltips
    callbacks: {
      label: function(tooltipItem, data) {
        const dataset = data.datasets[tooltipItem.datasetIndex];
        const currentValue = dataset.data[tooltipItem.index];
        return 'Value: ' + currentValue;
      }
    }
  },
};
```

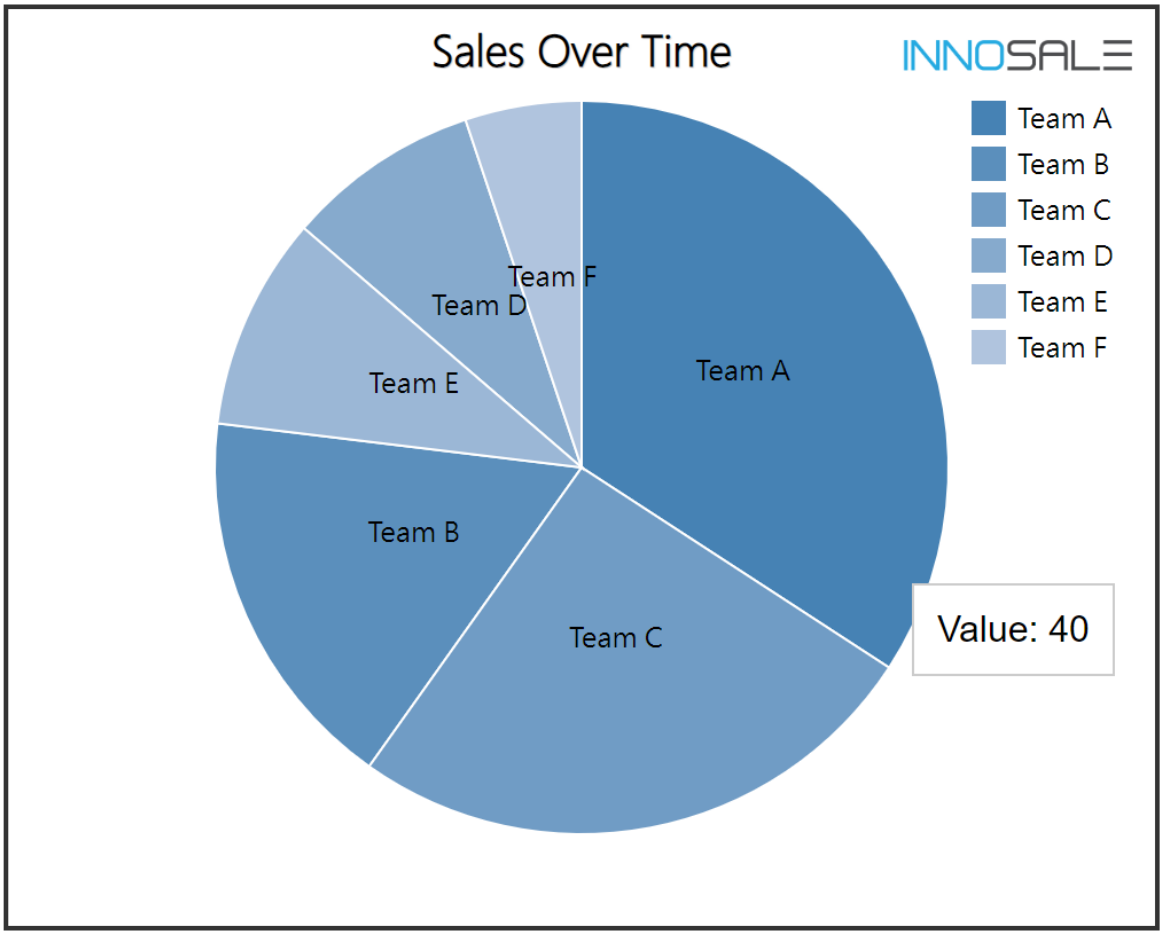

Figure 19. The pie chart generated with the example code.

Below is an example for a scatter plot options with zooming and panning options:

```
const chartOptions: ChartOptions = {
  responsive: true,
  tooltips: {
    enabled: true,
    mode: 'nearest',  // Tooltips display when hovering near data points
    intersect: true,  // Tooltips appear when directly intersecting with
data points
  },
  scales: {
    xAxes: [
      {
        type: 'linear',  // X-axis will have linear scale
        position: 'bottom',
        ticks: {
          min: 0,  // Minimum value for x-axis
          max: 100,  // Maximum value for x-axis
        },
      },
    ],
    yAxes: [
      {
        type: 'linear',  // Y-axis will have linear scale
        ticks: {
          min: 0,  // Minimum value for y-axis
          max: 100,  // Maximum value for y-axis
        },
      },
    ],
  },
  pan: {
    enabled: true,   // Enable panning
    mode: 'xy',      // Allow panning on both x and y axes
  },
  zoom: {
    enabled: true,   // Enable zooming
    mode: 'xy',      // Allow zooming on both x and y axes
  },
};
```

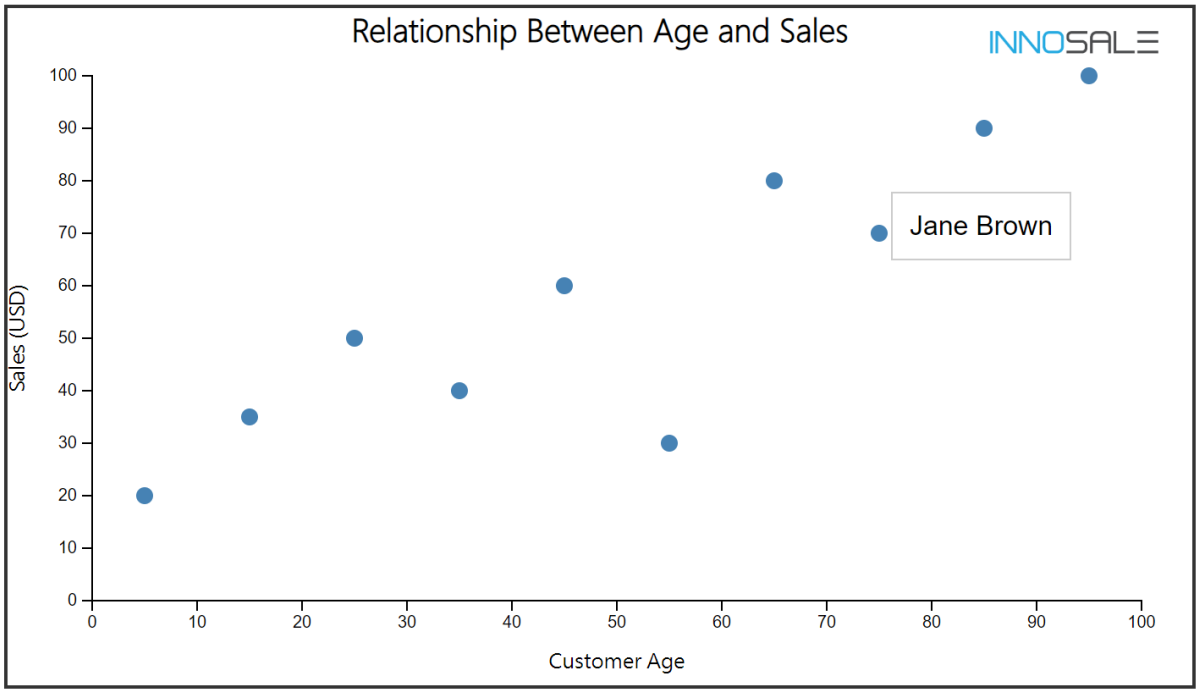

Figure 20. The scatter plot generated with the given code.

2.9.3.2 Usage

Table 5. InnoSale Basic Bar Chart Usage with sales data for different regions (.html)

```
<div class="mat-elevation-z4 bordered">
  <innosale-data-viz
    [chartType]="'bar'"
    [chartData]="salesChartData"
    [chartOptions]="chartOptions"
    [height]="'400px'"
    [width]="'100%'"
    (chartClick)="onChartClick($event)"
  ></innosale-data-viz>
</div>
```

Table 6. InnoSale Basic Chart Data (.ts)

```
class SalesDashboardComponent implements OnInit {
  salesChartData: ChartData = {
    labels: ['Region 1', 'Region 2', 'Region 3', 'Region 4', 'Region 5'],
    datasets: [
      {
        label: 'Sales in USD',
        data: [50000, 75000, 60000, 40000, 45000],
        backgroundColor: ['steelblue', 'lightsteelblue'], // If only two
is given, it will be interpolated
      },
    ],
  };

  chartOptions: ChartOptions = {
```

```
    responsive: true,
    legend: { display: true },
    scales: {
      xAxes: [{ display: true, labelString: 'Regions' }],
      yAxes: [{ display: true, ticks: { beginAtZero: true }, labelString:
'Sales' }],
    },
    tooltips: { enabled: true },
  };

  ngOnInit() {}

  onChartClick(event: ChartClickEvent) {
    console.log(`Element clicked: ${event.elementIndex}, Dataset:
${event.datasetIndex}`);
  }
}
```

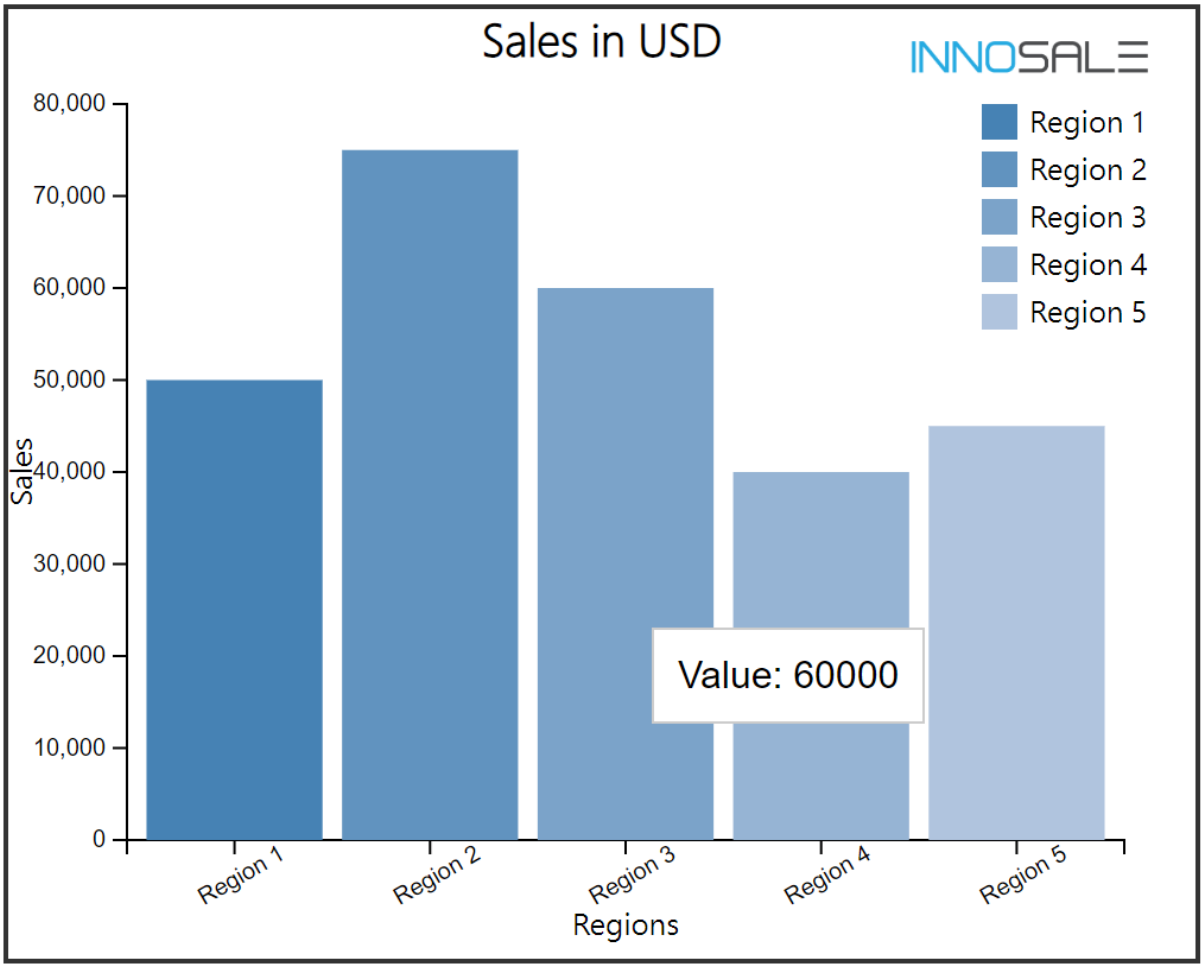

Figure 21. The bar chart generated with the example code.

2.10 Digital Product Description Component

2.10.1 Overview

The Digital Product Description Component enables a clear and structured presentation of digital products, enhancing effective communication. Digital Product Owners can set up key product details such as license, price, development technologies, and more, allowing customers to evaluate whether the product meets their requirements. By providing transparency about the product's features and development process, users are more likely to trust and invest in it.

This component is designed to allow the product owners to:

- Define the product's name and description
- Selecting and creating the digital product category (e.g., libraries, mobile apps, desktop apps)
- Set the product's price
- Add images and supported documents
- Attach relevant artifacts

2.10.2 API

```
import { DigitalProductDescriptionComponent, DigitalProduct,
DigitalProductLocalization } from 'digital-product-description';
```

Directives

digital-product-description	
Selector:	digital-product-description
Properties	
`@Input product: DigitalProduct`	The digital product data to be displayed or edited
`@Input editable: boolean`	Whether the component should be in edit mode
`@Input localization: DigitalProductLocalization`	Localization interface for the component (defaults to English)
`@Input allowedCategories: string[]`	Array of allowed product categories
`@Input maxImages: number`	Maximum number of images allowed
`@Input maxDocuments: number`	Maximum number of documents allowed
`@Input maxArtifacts: number`	Maximum number of artifacts allowed
`@Output productChange: EventEmitter<DigitalProduct>`	Emits when the product data is updated
`@Output productSave: EventEmitter<DigitalProduct>`	Emits when the save button is clicked
`@Output productCancel: EventEmitter<void>`	Emits when the cancel button is clicked
`@Output imageUpload: EventEmitter<File>`	Emits when a new image is uploaded
`@Output documentUpload: EventEmitter<File>`	Emits when a new document is uploaded

`@Output artifactUpload: EventEmitter<File>`	Emits when a new artifact is uploaded
`@Methods reset()`	Resets the component to its initial state
`@Methods validate()`	Validates the current product data and returns boolean
`@Methods exportProduct()`	Exports the current product data as JSON

Classes/Interfaces

Digital Product	
This interface defines the core data structure for a digital product entry.	
`name: string`	The name of the digital product.
`description: string`	Detailed description of the product.
`category: string`	The category of the digital product (e.g., library, mobile app).
`price: number`	The price of the product
`License: string`	The type of licence under which the product is distributed.
`technologies: string[]`	Array of technologies used in product development
`images: Image[]`	Array of product images/screenshots
`documents: Document[]`	Array of supporting documentation files
`artifacts: Artifact[]`	Array of additional product artifacts
Image	
Defines the structure for product images.	
`url: string`	The URL or path to the image
`caption: string`	Description of the image
`Type: string`	Type of image (e.g., screenshot, logo)
Document	
Defines the structure for supporting documents	
`name: string`	Name of the document
`url: string`	URL or path to the document
`type: string`	Type of document (e.g., manual, specification)
Artifact	
Defines the structure for product artifacts	
`name: string`	Name of the artifact
`url: string`	URL or path to the artifact
`type: string`	Type of artifact
`description: string`	Description of the artifact

2.10.3 Examples

Use of Digital Product Description component as shown in Table 15 results a view similar to one given in

Table 15. Digital Product Description Component Use Example (.html)

```
<!-- digital-product-example.component.html -->
<div class="digital-product-container">
  <h2>Digital Product Management</h2>

  <!-- Basic implementation -->
  <digital-product-description
    [product]="productData"
    [editable]="true"
    (productChange)="onProductChange($event)"
    (productSave)="onProductSave($event)"
    (productCancel)="onProductCancel()">
  </digital-product-description>

  <!-- Advanced implementation with all available options -->
  <digital-product-description
    #productComponent
    [product]="productData"
    [editable]="true"
    [localization]="customLocalization"
    [allowedCategories]="allowedCategories"
    [maxImages]="5"
    [maxDocuments]="3"
    [maxArtifacts]="10"
    (productChange)="onProductChange($event)"
    (productSave)="onProductSave($event)"
    (productCancel)="onProductCancel()"
    (imageUpload)="onImageUpload($event)"
    (documentUpload)="onDocumentUpload($event)"
    (artifactUpload)="onArtifactUpload($event)">
  </digital-product-description>

  <!-- Example of a button using component methods -->
  <button (click)="validateAndSave()">
    Validate and Save Product
  </button>
</div>
```

Table 16. Digital Product Description Component Use Example (.ts)

```
import { Component, OnInit } from '@angular/core';
import {
  DigitalProduct,
  DigitalProductLocalization
} from 'digital-product-description';

@Component({
  selector: 'app-digital-product-example',
  templateUrl: './digital-product-example.component.html',
  styleUrls: ['./digital-product-example.component.css']
})
export class DigitalProductExampleComponent implements OnInit {
  // Sample product data
  productData: DigitalProduct = {
    name: 'Analytics Dashboard Library',
    description: 'A comprehensive React component library for building analytics
dashboards',
    category: 'library',
    price: 299.99,
    license: 'MIT',
    technologies: ['React', 'TypeScript', 'D3.js'],
    images: [
      {
        url: '/assets/dashboard-preview.png',
```

```
        caption: 'Dashboard Preview',
        type: 'screenshot'
      }
    ],
    documents: [
      {
        name: 'Technical Documentation',
        url: '/assets/docs/technical-spec.pdf',
        type: 'specification'
      }
    ],
    artifacts: [
      {
        name: 'Demo Application',
        type: 'demo',
        url: 'https://demo.example.com',
        description: 'Live demo of the dashboard components'
      }
    ]
  };

  // Custom localization (optional)
  customLocalization: DigitalProductLocalization = {
    productNameLabel: 'Product Title',
    descriptionLabel: 'Product Overview',
    categoryLabel: 'Product Category',
    priceLabel: 'Product Price (USD)',
    licenseLabel: 'License Type',
    technologiesLabel: 'Tech Stack',
    imagesLabel: 'Screenshots & Images',
    documentsLabel: 'Documentation',
    artifactsLabel: 'Related Artifacts',
    saveButtonLabel: 'Publish Product',
    cancelButtonLabel: 'Discard Changes'
  };

  // Available categories for the product
  allowedCategories: string[] = [
    'library',
    'mobile-app',
    'desktop-app',
    'web-app',
    'plugin',
    'framework'
  ];

  constructor() {}

  ngOnInit(): void {}

  // Event handlers
  onProductChange(updatedProduct: DigitalProduct): void {
    console.log('Product updated:', updatedProduct);
    this.productData = updatedProduct;
  }

  onProductSave(product: DigitalProduct): void {
    console.log('Saving product:', product);
    // Implement save logic here
  }

  onProductCancel(): void {
    console.log('Edit cancelled');
    // Implement cancel logic here
  }

  onImageUpload(file: File): void {
    console.log('New image uploaded:', file);
    // Implement image upload logic here
```

```
  }

  onDocumentUpload(file: File): void {
    console.log('New document uploaded:', file);
    // Implement document upload logic here
  }

  onArtifactUpload(file: File): void {
    console.log('New artifact uploaded:', file);
    // Implement artifact upload logic here
  }

  // Example of using the component methods
  validateAndSave(): void {
    // Get reference to the component using ViewChild if needed
    if (this.productComponent.validate()) {
      const exportedData = this.productComponent.exportProduct();
      console.log('Valid product data:', exportedData);
      // Proceed with save
    }
  }
}
```

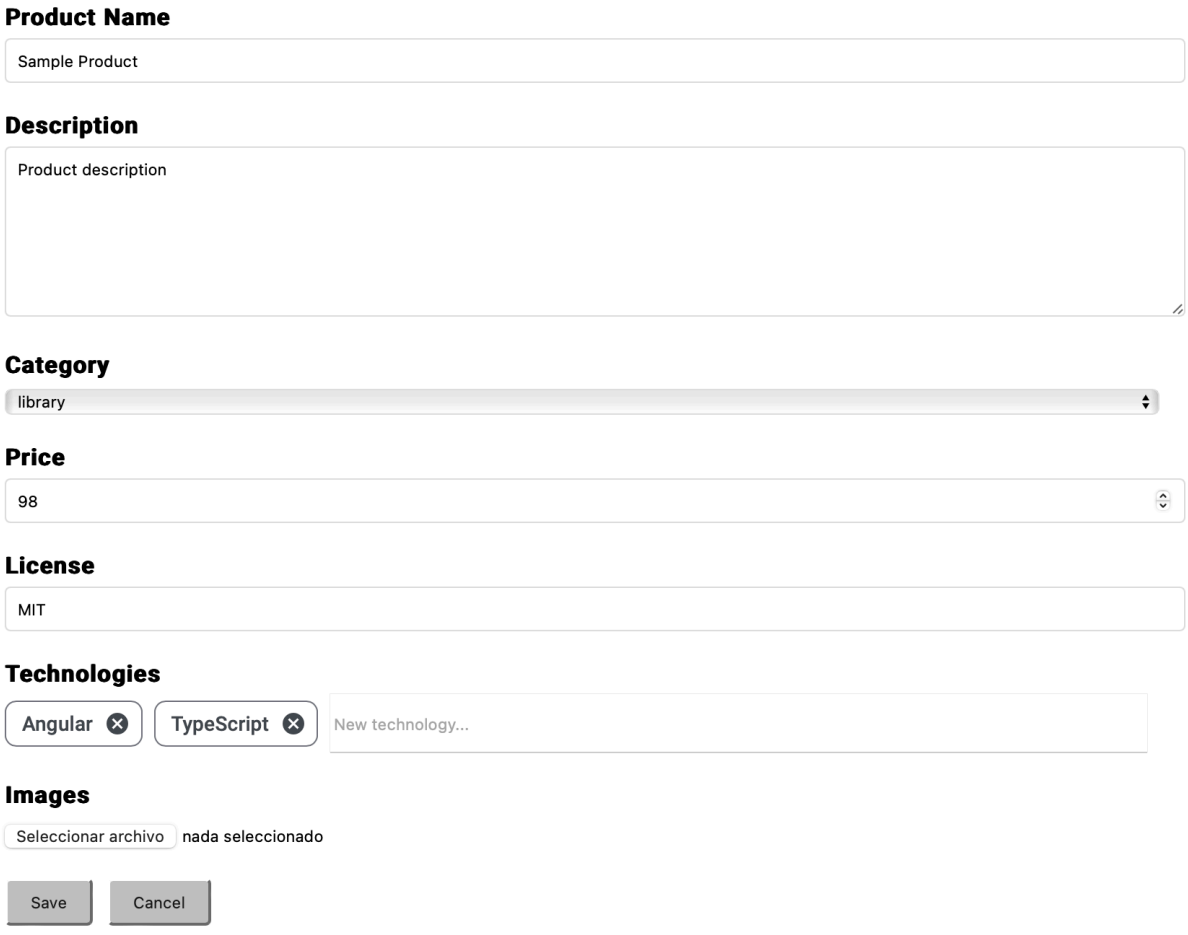

Figure 22. InnoSale Digital Product Description Component Example Usage

3 Conclusion

This deliverable provided the design details of all UDC subcomponents which were described in deliverable D4.1 User Dialogue Component: Specification. As explained earlier, the aim is not to provide a complete solution but rather building blocks of similar solutions, since each domain has its own requirements specific to that domain.

The subcomponent implementations are tested mainly as part of sample toy projects as well as while utilizing the subcomponents in the early implementations of the use cases. Furthermore, while the component designs are current as of this deliverable's publication, it is important to note that the use case implementations in WP6 are still ongoing. As a result, there may be minor changes to the subcomponents provided here, depending on any last-minute updates.

4 Abbreviations

2D	2-Dimensional
3D	3-Dimensional
AI	Artificial Intelligence
API	Application Programming Interface
CPQ	Configure, Price, Quote
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
LED	Light Emitting Diode
LLE	Light Lifting Equipment
MES	Manufacturing Execution System
PDF	Portable Document Format
SSO	Single Sign-On
UDC	User Dialog Component
UI	User Interface