FE Field Definitions

To reuse render and formatting logic, for consistency and simplicity, meta-data about fields should only live at a central point.

Architecture

- `ValueDefinition` define a specific type of value. Examples include primitives such as text or number, but also more business domain specifics such as monetaryAmount or condition
- `FieldDefinitions` define which field is of which value. Examples include the `project.title` (a `textValue`) or the `building.description.insuranceValue` (a `monetaryAmountValue`)
- For each type used in the fronend, these field definitions are created. To ease reuse of field definitions, changing which field they belong to is made easy.

Each functionality which uses these fields will then adapt the representation to their own use. For example, the grid converts a `FieldDefinition` to an appropriate `GridColDef`.

Design decisions

- The meta-data is independent to concrete functionality. Infer configuration out of the generic data and close to the actual functionality (e.g. infer `minColumnWidth` required by the grid out of the `fieldDefinition.renderWidthEm`, but do not directly include this in the field definitions as only useful for the grid).
- Each `FieldDefinition` corresponds to a single column/row. Create additional `FieldDefinition` for additional rows/columns (e.g. create an `compononentIdFieldDefinition` if the export needs a column with ids instead of labels).

The target of both decisions is to keep the meta-data simple, traded for complexity closer to the actual functionality (first decision) and more code duplication (second decision).

Implementation

The value definitions capture how a specific kind of value (e.g. `year`, `monetaryAmount`) is to be treated. This includes:

- `compare` to sort (e.g. in the `grid` or `sortControl`)
- `render` to display (e.g. in the `grid` or the `filterSummary`)
- `format` to display without EM_DASH fallback (e.g. in the `excelExport`)
- approximate width (e.g. `grid` or `excelExport` columns)
- `InputField` to filter (e.g. in the `filter`)

The field definitions then assign each field in an entity such a `valueDefinition`. To support entity structures (nested fields), there are utilities to locate the `fieldDefinition` of the substructure in the overall structure (e.g. to reuse the `buildingFieldDefinitions` in the superstructure `{building: BuildingDto }` call `locateFields('building', buildingFieldDefinitions)`)

The `fieldDefinitions` / `valueDefinitions` then power generic functionality:

- `useDataGridColumns` which converts the field definition to something consumable by the mui `DataGrid`. Note how the field definitions are reused for the initial width & the sorting of the columns, as well as for the quick search.
- `sort` which uses the field definitions to display the sort control & execute the sorting.
- `filter` which uses the field definitions to gather the applicable comparison operators, the input field to display, and to execute the filtering.
- `ExcelExport` which uses the field definitions to set the cell value.

Another potential use-case could be for the forms; but this has not been explored so far.

Example

Lets say we have the following object:

```
type Person = {
  givenName: string
  familyName: string
  age: number
  spouse: Person
}
```

We would create the following value definitions:

```
// note that we could shorten this definition by using the stringValueTemplate
export const nameValue = {
  key: 'name',
  valueType: ValueType.string,
  compare: (value: string) => value,
  render: (value: string | undefined) => renderValueOrDash(value),
  format: (value: string | undefined) => value,
  InputField: ControlledTextField,
  renderInputFieldValue: (value: string | undefined) => renderValueOrDash(value),
  widthEm: 20,
}

// note that we could shorten this definition by using the numberValueTemplate
export const ageValue = {
  key: 'age',
  valueType: ValueType.number,
  compare: (value: number) => value,
  render: (value: number | undefined) => renderValueOrDash(value),
  format: (value: number | undefined) => value,
  InputField: ControlledNumberField,
  InputFieldProps: {},
  renderInputFieldValue: (value: number | undefined) => renderValueOrDash(value),
  widthEm: 2,
}
```

Then, we would create the corresponding field definitions:

```
const givenName: FieldDefinition<Person, 'givenName'> = {
  label: () => i18n.t('entities.person.givenName'),
  path: 'givenName',
  valueDefinition: nameValue,
}

const familyName: FieldDefinition<Person, 'familyName'> = {
  label: () => i18n.t('entities.person.familyName'),
  path: 'familyName',
  valueDefinition: nameValue,
}

const age: FieldDefinition<Person, 'age'> = {
  label: () => i18n.t('entities.person.age'),
  path: 'age',
  valueDefinition: ageValue,
}

// we can reuse these field definitions to create field definitions for the spouse
```

```
const spouseFields = locateFields(
  [givenName, familyName],
  'spouse'
)

export personFields = [
  givenName,
  familyName,
  age,
  ...spouseFields
]
```