SessionItem

1. Introduction

`SessionItem` class is a base element to build a hierarchical structure representing all the data of the application running. `SessionItem` can contain an arbitrary amount of basic data types, and can be a parent for other `SessionItems`.

The tree of `SessionItem` objects can be built programmatically via `SessionItem` API, or be reconstructed from persistent content (XML and JSON files, for example).

While being an end leaf in some ramified hierarchy the `SessionItem` often plays a role of a single editable/displayable entity. For example, a `SessionItem` can be seen as an aggregate of information necessary to display/edit a single integer number `42` in the context of some view. Then, it will carry:

- An integer number with the value set to `42`.
- A collection of appearance flags, stating if the value is visible, is read-only, should be shown as disabled (grayed out), and so on.
- Other auxiliary information (tooltips to be shown, allowed limits to change, and similar).

2. The data of `SessionItem`

The data carried by `SessionItem` is always associated with the role - a unique integer number defining the context in which the data has to be used. They both came in pairs, and the item can have multiple `data/roles` defined.

```
// currently supported elementary data types
using variant_t = std::variant<std::monostate, bool, int, double, std::string,
std::vector<double>>;

// convenience type
using datarole_t = std::pair<variant_t, int>;

// collection of predefined roles
namespace DataRole
{
const int kIdentifier = 0;  //!< item unique identifier
const int kData = 1;        //!< main data role
```

```
const int kDisplay = 2;     //!< display name
const int kAppearance = 3;  //!< appearance flag
}
```

In the snipped below, the data is set and then accessed for two roles, the display role holding a label and the data role, holding the value.

```
SessionItem item;
item.SetData(42, kData);
item.SetData("Width [nm]", kDisplay)

auto number = item.Data<int>(kData);
auto label = item.Data<std::string>(kDisplay);
```

2.1 Related files

- `variant.h` contains definitions of `variant_t` and `datarole_t` data types. Check it for all suported elementary data types.
- `mvvm_types.h` defines constants and enums. Check it to see current roles, or appearance flags.
- `sessionitemdata.h` contains the definition of `SessionItemData` class. It is a member of `SessionItem` and carries all the logic related to item's data. Most of methods of `SessionItemData` are replicated by `SessionItem`.
- `sessionitemdata.test.cpp` contains unit tests of `SessionItemData` and can be used as an API usage example.

3. Inheriting from `SessionItem`

The `SessionItem` class type name is stored in a string variable and can be accessed via the `GetType()` method:

```
SessionItem item;
std::cout << item.GetType() << std::endl;
>>> "SessionItem"
```

This name is used during item serialization/deserialization and during undo/redo operations to create objects of the correct type in item factories (explained in `sessionmodel.md`).

To inherit from `SessionItem` the new unique name has to be provided in the constructor of the derived class. It is convenient to make this name identical to the class name itself:

```
class SegmentItem : publis SessionItem
{
public:
  const static std::string Type = "SegmentItem";
```

```
  SegmentItem() : SessionItem(Type) {}
}
```

3.1 Related files

- `itemfactory.h` contains `ItemFactory` class definition. It is used in the context of `SessionModel` for user class registration.

4. Children of SessionItem

`SessionItem` can have an arbitrary amount of children stored in named containers. In pseudo code, it can be expressed

```
class SessionItem
{
  using named_container_t = std::pair<std::string, std::vector<SessionItem*>>;
  std::vector<named_container_t> m_tagged_items;
}
```

Named containers are a convenient way to have items tied to a certain context. The name of the container, `tag` and the position in it, `index` can be used to access and manipulate items through their parent `SessionItem`. Before adding any child to `SessionItem`, the container has to be created and its properties defined.

4.1 The `TagInfo` class

The `TagInfo` specifies information about children that can be added to a `SessionItem`. A `TagInfo` has a name, min, max allowed number of children, and vector of all types that children can have.

In the snippet below we register a tag with the name `ITEMS` intended for storing unlimited amount of other `SessionItems`.

```
SessionItem item;
item.RegisterTag(TagInfo("ITEMS", 0, -1));
```

An equivalent way of creating the same is to use convenience factory methods of the `TagInfo` class:

```
SessionItem item;
item.RegisterTag(TagInfo::CreateUniversalTag("ITEMS"));
```

Internally, it leads to the creation of a corresponding named container ready for items to be inserted. In another example, we define a tag with the name `Position` intended for storing the only item of type `VectorItem`.

```
item.RegisterTag(TagInfo("Position", 1, 1, {"VectorItem"});

// or
// item.RegisterTag(TagInfo::CreatePropertyTag("Position", "VectorItem"));
```

4.2 The `TagIndex` class

The `TagIndex` class is a simple aggregate carrying a string with container name, and an index indicating the position in the container.

```
struct TagIndex
{
  std::string tag = {};
  int index = -1;
}
```

The `TagIndex` class uniquely defines the position of a child and it is used in the `SessionItem` interface to access and manipulate items in containers.

4.3 Adding children

There are multiple ways to add children to a parent. In snipped below we register a tag with the name "`ITEMS`" intended for storing an unlimited amount of items of any type. In the next step, we insert a child into the corresponding container and modify its display name. Later, we access the child using the known `TagIndex` to print the child's display name.

```
const std::string tag("ITEMS");
SessionItem item;
item.RegisterTag(TagInfo::CreateUniversalTag(tag));

auto child0 = item.InserItem({tag, 0});
child0->SetDisplayName("Child");

std::cout << item.GetItem(tag)->GetDisplayName() << "\n";
>>> "Child"
```

There are other alternative ways to add children:

```
// appends new SessionItem
auto child0 = item.InserItem({tag, -1});

//! appends new PropertyItem
auto child1 = item.InserItem<PropertyItem>({tag, -1});

// inserts child between child0 and child1 using move semantic
```

```
auto another = std::make_unique<VectorItem>
auto child2 = item.InserItem(std::move(another), {tag, 1});
```

4.5 Related files

- `taginfo.h` defines `TagInfo` class. It holds an information about single tag of `SessionItem`.
- `sessionitemcontainer.h` defines `SessionItemContainer` class. It holds the collection of `SessionItem` objects and `TagInfo` describing container properties.
- `taggeditems.h` defines `TaggedItems` class. It is a member of `SessionItem` and it holds a collection of `SessionItemContainers`.

Links

- Martin Fowler, Presentation Model
- Martin Folwer, GUI architectures