New Json Utility API

Raw value conversion (GetValue)

`GetValue` is a convenience helper that will either read a value into existing storage (type-deduced) or return a JSON value coerced into the specified type.

When reading into existing storage, it returns a boolean indicating whether that storage was modified.

If the JSON value cannot be converted to the specified type, an exception will be generated. For non-nullable type conversions (most POD types), `null` is considered to be an invalid type.

```
std::string one;
std::optional<std::string> two;

JsonUtils::GetValue(json, one);
// one is populated or an exception is thrown.

JsonUtils::GetValue(json, two);
// two is populated, nullopt or an exception is thrown

auto three = JsonUtils::GetValue<std::string>(json);
// three is populated or an exception is thrown

auto four = JsonUtils::GetValue<std::optional<std::string>>(json);
// four is populated or nullopt
```

Key lookup (GetValueForKey)

`GetValueForKey` follows the same rules as `GetValue`, but takes an additional key. It is assumed that the JSON value passed to GetValueForKey is of `object` type.

```
std::string one;
std::optional<std::string> two;

JsonUtils::GetValueForKey(json, "firstKey", one);
// one is populated or unchanged.

JsonUtils::GetValueForKey(json, "secondKey", two);
// two is populated, nullopt or unchanged

auto three = JsonUtils::GetValueForKey<std::string>(json, "thirdKey");
// three is populated or zero-initialized

auto four = JsonUtils::GetValueForKey<std::optional<std::string>>(json,
"fourthKey");
// four is populated or nullopt
```

Rationale: Value-Returning Getters

JsonUtils provides two types of `GetValue...` : value-returning and reference-filling.

The reference-filling fixtures use type deduction so that a developer does not need to specify template parameters on every `GetValue` call. It excels at populating class members during deserialization.

The value-returning fixtures, on the other hand, are very useful for partial deserialization and key detection when you do not need to deserialize an entire instance of a class or you need to reason about the presence of members.

To provide a concrete example of the latter, consider:

```
if (const auto guid{ GetValueForKey<std::optional<GUID>>(json, "guid") })
    // This condition is only true if there was a "guid" member in the provided JSON
object.
    // It can be accessed through *guid.
}
```

If you are...	Use
Deserializing	`GetValue(..., storage)`
Interrogating	`storage = GetValue<T>(...)`

Converting User-Defined Types

All conversions are done using specializations of `JsonUtils::ConversionTrait<T>` . To implement a converter for a user-defined type, you must implement a specialization of `JsonUtils::ConversionTrait<T>` .

Every specialization over `T` must implement `static T FromJson(const Json::Value&)` and `static bool CanConvert(const Json::Value&)` .

```
struct MyCustomType { int val; };

template<>
struct ConversionTrait<MyCustomType>
{
    // This trait converts a string of the format "[0-9]" to a value of type
MyCustomType.

    static MyCustomType FromJson(const Json::Value& json)
    {
        return MyCustomType{ json.asString()[0] - '0' };
    }

    static bool CanConvert(const Json::Value& json)
    {
        return json.isString();
    }
};
```

Converting User-Defined Enumerations

Enumeration types represent a single choice out of multiple options.

In a JSON data model, they are typically represented as strings.

For parsing enumerations, JsonUtils provides the `JSON_ENUM_MAPPER` macro. It can be used to establish a converter that will take a set of known strings and convert them to values.

```
JSON_ENUM_MAPPER(CursorStyle)
{
    // pair_type is provided by ENUM_MAPPER.
    JSON_MAPPINGS(5) = {
        pair_type{ "bar", CursorStyle::Bar },
        pair_type{ "vintage", CursorStyle::Vintage },
        pair_type{ "underscore", CursorStyle::Underscore },
        pair_type{ "filledBox", CursorStyle::FilledBox },
        pair_type{ "emptyBox", CursorStyle::EmptyBox }
    };
};
```

If the enum mapper fails to convert the provided string, it will throw an exception.

Converting User-Defined Flag Sets

Flags represent a multiple-choice selection. They are typically implemented as enums with bitfield values intended to be ORed together.

In JSON, a set of flags may be represented by a single string (`"flagName"`) or an array of strings (`["flagOne", "flagTwo"]`).

JsonUtils provides a `JSON_FLAG_MAPPER` macro that can be used to produce a specialization for a set of flags.

Given the following flag enum,

```
enum class JsonTestFlags : int
{
    FlagOne = 1 << 0,
    FlagTwo = 1 << 1
};
```

You can register a flag mapper with the `JSON_FLAG_MAPPER` macro as follows:

```
JSON_FLAG_MAPPER(JsonTestFlags)
{
    JSON_MAPPINGS(2) = {
        pair_type{ "flagOne", JsonTestFlags::FlagOne },
        pair_type{ "flagTwo", JsonTestFlags::FlagTwo },
    };
};
```

The `FLAG_MAPPER` also provides two convenience definitions, `AllSet` and `AllClear`, that can be used to represent "all choices" and "no choices" respectively.

```
JSON_FLAG_MAPPER(JsonTestFlags)
{
    JSON_MAPPINGS(4) = {
        pair_type{ "never", AllClear },
        pair_type{ "flagOne", JsonTestFlags::FlagOne },
        pair_type{ "flagTwo", JsonTestFlags::FlagTwo },
        pair_type{ "always", AllSet },
    };
};
```

Because flag values are additive, `["always", "flagOne"]` will result in the same behavior as `"always"`.

If the flag mapper encounters an unknown flag, it will throw an exception.

If the flag mapper encounters a logical discontinuity such as `["never", "flagOne"]` (as in the above example), it will throw an exception.

Advanced Use

`GetValue` and `GetValueForKey` can be passed, as their final arguments, any value whose type implements the same interface as `ConversionTrait<T>` --that is, `FromJson(const Json::Value&)` and `CanConvert(const Json::Value&)`.

This allows for one-off conversions without a specialization of `ConversionTrait` or even stateful converters.

Stateful Converter Sample

```
struct MultiplyingConverter {
    int BaseValue;

    bool CanConvert(const Json::Value&) { return true; }

    int FromJson(const Json::Value& value)
    {
        return value.asInt() * BaseValue;
    }
};

...

Json::Value json{ 66 }; // A JSON value containing the number 66
MultiplyingConverter conv{ 10 };

auto v = JsonUtils::GetValue<int>(json, conv);
// v is equal to 660.
```

Behavior Chart

GetValue(T&) (type-deducing)

-	json type invalid	json null	valid

`T`	❌ exception	❌ exception	✓ converted
`std::optional<T>`	❌ exception	🟨 `nullopt`	✓ converted

GetValue<T>() (returning)

-	json type invalid	json null	valid
`T`	❌ exception	❌ exception	✓ converted
`std::optional<T>`	❌ exception	🟨 `nullopt`	✓ converted

GetValueForKey(T&) (type-deducing)

GetValueForKey builds on the behavior set from GetValue by adding a "key not found" state. The remaining three cases are the same.

val type	key not found	*json type invalid*	*json null*	*valid*
`T`	🔵 unchanged	❌ *exception*	❌ *exception*	✓ *converted*
`std::optional<T>`	🔵 unchanged	❌ *exception*	🟨 *`nullopt`*	✓ *converted*

GetValueForKey<T>() (return value)

val type	key not found	*json type invalid*	*json null*	*valid*
`T`	🟨 `T{}` (zero value)	❌ *exception*	❌ *exception*	✓ *converted*
`std::optional<T>`	🟨 `nullopt`	❌ *exception*	🟨 *`nullopt`*	✓ *converted*

Future Direction

These converters lend themselves very well to automatic *serialization*.