SDK Documentation

Core Types

Application

The `Application` class manages application-specific settings and objects. Moreover, it provides an interface for creating and loading documents. Usually, there is only one `Application` instance for the entire editing session.

The interface:

Creates a new `Application` instance. Receives the path to the application resources. If path is not specified, resources are looked up in the 'Resources' subdirectory of the current binary module's directory.
Throws: `ApplicationCreateError`.

```
Application(boost::optional<std::string> resourcePath = boost::none);
```

Provide access to the `Messenger` instance.

```
std::shared_ptr<const Messenger> getMessenger() const;
std::shared_ptr<Messenger> getMessenger();
```

Create an empty document of the specified type with default settings or with the specified custom settings.
Throws: `DocumentCreateError`.

```
Document::Document createDocument(Document::DocumentType documentType);
Document::Document createDocument(const Document::CreateDocumentSettings& settings);
```

Load an existing document from a file with the specified path. The document format and type can be determined automatically by the file extension, if not unspecified.
Throws: `DocumentLoadError`.

```
Document::Document loadDocument(const std::string& filePath);
Document::Document loadDocument(const std::string& filePath, const Document::LoadDocumentSettings& settings);
```

DocumentSettings

`CreateDocumentSettings` provides settings, that are necessary for creating new documents.

```
struct CreateDocumentSettings
{
    boost::optional<DocumentFormat> documentFormat;
    boost::optional<DocumentType> documentType;
    UserInfo userInfo;
    LocaleInfo localeInfo;
    TimeZone timeZone;
    FormulaType formulaType = FormulaType::A1;
};
```

`LoadDocumentSettings` provides extra settings, that are necessary for loading documents from file.

```
struct LoadDocumentSettings : public CreateDocumentSettings
{
    Encoding encoding = Encoding::UTF8;
    DSVSettings dsvSettings;

    // `documentPassword` is not stored in `DocumentSettings` for security reasons after document was created or loaded.
    // Thus it is necessary to pass password for saving of encrypted document, even if the password was specified
    // earlier during load.
    boost::optional<std::string> documentPassword;
};
```

`SaveDocumentSettings` provides extra settings, that are necessary for saving documents to file.

```
struct SaveDocumentSettings
{
    boost::optional<DocumentFormat> documentFormat;
    boost::optional<DocumentType> documentType;
    boost::optional<std::string> documentPassword;
};
```

Here are the simple structures and types used in document settings:

The `UserInfo` struct provides user information.

```
struct UserInfo
{
    std::string name;
    std::string email;
};
```

The `CurrencySignPlacement` enum contains currency sign placement variants.

```
enum class CurrencySignPlacement : std::uint8_t
{
```

```
    Prefix,
    Suffix
};
```

The `LocaleInfo` struct provides information about the locale.

```
struct LocaleInfo
{
    // The locale name is represented in the <language>_<REGION> format, where the language code is specified by the
    // ISO-639 standard and the region code by the ISO-3166 standard.
    std::string localeName = "en_US";

    // Decimal separator of numeric values (separates integer and fractional parts).
    boost::optional<std::string> decimalSeparator;

    // Symbol that separates groups of digits in numeric values.
    boost::optional<std::string> thousandSeparator;

    // Symbol that separates items in a list.
    boost::optional<std::string> listSeparator;

    // Currency symbol used in current country/region.
    boost::optional<std::string> currencySymbol;

    // Currency sign placement used in current region
    boost::optional<CurrencySignPlacement> currencyFormat;

    // Custom format string for a short date value in current region (e.g. -'m/d/yy' for en_US).
    boost::optional<std::string> shortDatePattern;

    // Custom format string for a long date value in current region (e.g. -'dddd, mmmm d, yyyy' for en_US).
    boost::optional<std::string> longDatePattern;

    // Custom format string for a time value in current region (e.g. -'h:mm AM/PM' for en_US)
    boost::optional<std::string> timePattern;
};
```

The `TimeZone` struct provides timezone information.

```
struct TimeZone
{
    int32_t offsetInSecondsToUTC = 0;
};
```

`DSVSettings` provides extra settings, that are necessary for DSV (delimiter-separated value) file conversion.

```
struct DSVSettings
{
    std::u32string separators = U",";
    char32_t escapeChar = '"';
};
```

DocumentFormat

The `DocumentFormat` enum contains all supported document formats.

```
enum class DocumentFormat : std::uint8_t
{
    PlainText,
    DSV,
    OXML,
    ODF,
    HTML,
    PDF,
    PDFA
};
```

DocumentType

The `DocumentType` enum contains all supported document types.

```
enum class DocumentType : std::uint8_t
{
    Text,
    Workbook,
    Presentation
};
```

SaveUnsupportedFeature

The `SaveUnsupportedFeature` enum contains features, unsupported while saving in certain formats.

```
enum class SaveUnsupportedFeature : std::uint8_t
{
    CommentsInHeaderFooter,
    CommentsInFootnote,
    MixedNoteAlignment,
```

```
};

using SaveUnsupportedFeatures = std::set<SaveUnsupportedFeature>;
```

Encoding

The `Encoding` enum contains all supported document encodings.

```
enum class Encoding : std::uint8_t
{
    Unknown,
    UTF8,
    UTF16BE,
    UTF16LE,
    UTF32BE,
    UTF32LE,
    Windows1250,
    Windows1251,
    Windows1252,
    ISO8859Part5,
    KOI8R,
    KOI8U,
    CP866
};
```

ExportFormat

The `ExportFormat` enum contains all supported document formats for export.

```
enum class ExportFormat : std::uint8_t
{
    PDFA1
};
```

FormulaType

The `FormulaType` enum contains all supported formula types.

```
enum class FormulaType : std::uint8_t
{
    A1,
    R1C1
};
```

Enumerator

`Enumerator` is a base class for all enumerators.
T - a type of elements returned by this enumerator.

```
template <typename T>
class Enumerator
```

The interface:

Returns `true` if the iteration has more elements.

```
virtual bool isValid() const = 0;
```

Returns the current element.
Throws: `NoSuchElementError`.

```
virtual T getCurrent() const = 0;
```

Switches enumerator to the next element.
Throws: `NoSuchElementError`.

```
virtual void goToNext() = 0;
```

Messenger

The `Messenger` class represents a simple logging utility. It can be used by a client to get an additional information about internal framework activity as well as to notify all subscribers with client-specific messages.

The interface:

An alias for the function which processes messages.

```
typedef std::function<void(const Message&)> MessageHandlerFunction;
```

Subscribes to messages using the `MessageHandler` interface.

```
virtual std::shared_ptr<Connection> subscribe(const std::shared_ptr<MessageHandler> handler) = 0;
```

Subscribes to messages using a functor.

```
virtual std::shared_ptr<Connection> subscribe(const MessageHandlerFunction& handler) = 0;
```

Notifies all subscribers with the specified message.

```
virtual void notify(const Message& message) const = 0;
```

Here are the simple structures and types used in Messenger:

The `Message` class represents an object that contains a message and its severity level.

```
class CO_API_SYMBOL Message
{
public:
    // The `Severity` enum contains all supported severity levels.
    enum class Severity : std::uint8_t
    {
        Info,
        Warning,
        Error
    };

    bool operator==(const Message& other) const;

    Severity getSeverity() const;
    const std::string& getText() const;

    // Creates `Message` with the corresponding severity level.
    static Message makeInfo(const std::string& text);
    static Message makeWarning(const std::string& text);
    static Message makeError(const std::string& text);
};
```

The `Connection` class represents a connection between `Messenger` and a client.

```
class CO_API_SYMBOL Connection
{
public:
    // Client automatically unsibscribes in destructor.
    virtual ~Connection() = default;

    // Unsibscribes client from `Messenger`.
    virtual void unsubscribe() = 0;
};
```

The `MessageHandler` is an interface which can be implemented to process messages.

```
class CO_API_SYMBOL MessageHandler
{
public:
    virtual ~MessageHandler() = default;

    virtual void handle(const Message& message) const = 0;
};
```

ID

The `ID` class represents a framework ID type.

The interface:

Comparison operators:

```
bool operator==(const ID& other) const;
bool operator!=(const ID& other) const;
```

Geometry

The `GenericPoint2D` struct represents a 2-dimensional point.

```
template <typename T>
struct GenericPoint2D
{
    GenericPoint2D();
    GenericPoint2D(const T& xArg, const T& yArg);

    std::string toString() const;

    T x;
    T y;
};
```

The `GenericSize2D` struct represents a 2-dimensional size.

```
template <typename T>
struct GenericSize2D
{
    GenericSize2D();
    GenericSize2D(const T& widthArg, const T& heightArg);

    std::string toString() const;

    T width;
    T height;
};
```

The `GenericRect` struct represents a rectangle.

```
template <typename T>
struct GenericRect
{
    GenericRect();
    GenericRect(const GenericPoint2D<T>& topLeftArg, const GenericPoint2D<T>& bottomRightArg);
    GenericRect(const T& topLeftX, const T& topLeftY, const T& bottomRightX, const T& bottomRightY);

    std::string toString() const;

    GenericPoint2D<T> topLeft;
    GenericPoint2D<T> bottomRight;
};
```

Graphics

The `ColorRGBA` struct represents color components in [0..255] range.

```
struct CO_API_SYMBOL ColorRGBA
{
    ColorRGBA();
    ColorRGBA(const std::uint8_t rArg, const std::uint8_t gArg, const std::uint8_t bArg, const std::uint8_t aArg);

    bool operator==(const ColorRGBA& other) const;

    std::uint8_t r;
    std::uint8_t g;
    std::uint8_t b;
    std::uint8_t a;
};
```

Exceptions

`BaseError` is a base class for all exceptions which SDK can throw. For further details, call `std::runtime_error::what()`.

```
class CO_API_SYMBOL BaseError : public std::runtime_error
{
public:
    explicit BaseError(const std::string& error);
};
```

`ApplicationCreateError` is thrown when an `Application` instance cannot be created.

```
class CO_API_SYMBOL ApplicationCreateError : public BaseError
{
public:
    explicit ApplicationCreateError(const std::string& error);
};
```

`IncorrectArgumentError` is thrown when one of the arguments of method or function has invalid value.

```
class CO_API_SYMBOL IncorrectArgumentError : public BaseError
{
public:
    IncorrectArgumentError(const std::string& argumentName,
                           const std::string& argumentValue,
                           const std::string& reason);
};
```

`InvalidObjectError` is thrown when the object can no longer be used.

```
class CO_API_SYMBOL InvalidObjectError : public BaseError
{
public:
    InvalidObjectError();
    InvalidObjectError(const std::string& objectName);
};
```

`DocumentCreateError` is thrown when a document cannot be created.

```
class CO_API_SYMBOL DocumentCreateError : public BaseError
```

```
{
public:
    explicit DocumentCreateError(const std::string& error);
};
```

`DocumentLoadError` is thrown when a document cannot be loaded.

```
class CO_API_SYMBOL DocumentLoadError : public BaseError
{
public:
    explicit DocumentLoadError(const std::string& error);
};
```

`DocumentSaveError` is thrown when a document cannot be saved.

```
class CO_API_SYMBOL DocumentSaveError : public BaseError
{
public:
    explicit DocumentSaveError(const std::string& error);
};
```

`DocumentExportError` is thrown when a document cannot be exported.

```
class CO_API_SYMBOL DocumentExportError : public BaseError
{
public:
    explicit DocumentExportError(const std::string& error);
};
```

`NoSuchElementError` is thrown when the element does not exist.

```
class CO_API_SYMBOL NoSuchElementError : public BaseError
{
public:
    NoSuchElementError();
};
```

`NotImplementedError` is thrown to report a not implemented functionality.

```
class CO_API_SYMBOL NotImplementedError : public BaseError
{
public:
    explicit NotImplementedError(const std::string& error);
};
```

`OutOfRangeError` is thrown to report an out-of-range error.

```
class CO_API_SYMBOL OutOfRangeError : public BaseError
{
public:
    explicit OutOfRangeError(const std::string& rangeAsStr);
};
```

`ParseError` is thrown when a text parameter cannot be parsed.

```
class CO_API_SYMBOL ParseError : public BaseError
{
public:
    explicit ParseError(const std::string& error);
};
```

`UnknownError` is thrown when an unknown critical exception has occurred in the framework. In this case, the application is to be terminated because the framework is in undefined state.

```
class CO_API_SYMBOL UnknownError : public BaseError
{
public:
    UnknownError();
    explicit UnknownError(const std::string& error);
};
```

C++ Examples

Basic usage:

```
auto application = API::Application();
auto document = application.createDocument(API::Document::DocumentType::Workbook);

// Work with the document...

document.saveAs(filePath);
```

Python Examples

Basic usage:

```python
from CollabioPythonSDK import CoreAPI

if __name__ == '__main__':
    application = CoreAPI.Application()
    document = application.createDocument(CoreAPI.DocumentType_Workbook)

    # Work with the document...

    document.saveAs(file_path)
```

Document Model

Document

The `Document` class provides an interface for an access to the document object model which can be used to read and modify the document using high-level entities.
All objects from the hierarchy can throw `InvalidObjectError`.

The interface:

Save the document to a file with the specified path. The document format and type will be determined by file extension, if not unspecified.
Throws: `DocumentSaveError`.

```
void saveAs(const std::string& filePath) const;
void saveAs(const std::string& filePath, const SaveDocumentSettings& settings) const;
```

Validates the document and checks what will be lost while saving specified type to the specified format.

```
SaveUnsupportedFeatures validate(DocumentFormat format, DocumentType type) const;
```

Exports the document to a file with the specified path and format.
Throws: `DocumentExportError`.

```
void exportTo(const std::string& filePath, ExportFormat format) const;
```

Provides an access to the blocks which are stored in the document.

```
Blocks getBlocks();
```

Provides an access to the bookmarks which are stored in the document.

```
Bookmarks getBookmarks();
```

Provides an access to the scripts which are stored in the document.

```
std::shared_ptr<Scripts> getScripts();
```

Creates a range containing the whole document.

```
Range getRange();
```

Scripts

The `Scripts` class provides an interface for an access to the scripts collection.

The interface:

Returns a script by name.
Returns `nullptr` if a script with the specified name does not exist.

```
virtual std::shared_ptr<Script> getScript(const std::string& name) const = 0;
```

Adds a new script or changes an existing value.

```
virtual void setScript(const std::string& name, const std::string& script) = 0;
```

Removes a script by name.

```
virtual void removeScript(const std::string& name) = 0;
```

Returns `Enumerator` which iterates all scripts.

```
virtual std::shared_ptr<Enumerator<std::shared_ptr<Script>>> getEnumerator() = 0;
```

Script

The `Script` class provides an interface for an access to a script.

The interface:

The script name property.

```
virtual std::string getName() const = 0;
virtual void setName(const std::string& name) = 0;
```

The script body property.

```
virtual std::string getBody() const = 0;
```

```
virtual void setBody(const std::string& body) = 0;
```

Blocks

The `Blocks` class provides an interface for an access to the blocks collection.

The interface:

Returns a block with corresponding type by index or boost::none if index is out of range.

```
boost::optional<Block> getBlock(std::size_t blockIndex);
boost::optional<Paragraph> getParagraph(std::size_t paragraphIndex);
boost::optional<Shape> getShape(std::size_t shapeIndex);
boost::optional<Table> getTable(std::size_t tableIndex);
boost::optional<Field> getField(std::size_t fieldIndex);
```

Returns a block with corresponding type by ID or boost::none if there is no table with the ID specified.

```
boost::optional<Table> getTable(const ID& tableID);
```

Returns `Enumerator` which iterates all corresponding blocks. The enumerator becomes invalid when the `Blocks` object has been modified.

```
std::shared_ptr<Enumerator<Block>> getEnumerator();
std::shared_ptr<Enumerator<Paragraph>> getParagraphsEnumerator();
std::shared_ptr<Enumerator<Shape>> getShapesEnumerator();
std::shared_ptr<Enumerator<Table>> getTablesEnumerator();
std::shared_ptr<Enumerator<Field>> getFieldsEnumerator();
```

Block

`Block` is a base class for all document blocks.

The interface:

Casts the `Block` object to the corresponding type. If it is impossible, return boost::none.

```
boost::optional<Paragraph> toParagraph() const;
boost::optional<Table> toTable() const;
boost::optional<Shape> toShape() const;
boost::optional<Field> toField() const;
```

Returns a range containing the block.

```
Range getRange();
```

Removes this block from document. Current instance of `Block` become invalid.

```
void remove();
```

Bookmarks

The `Bookmarks` class provides an interface for an access to document's bookmarks.

The Bookmarks interface in C++ SDK:

To insert a bookmark it is neccessary to have a position object. The position in the end of the document is used in this example. The name of the bookmark should be unique.

```
auto position = document.getRange().getEnd();
position.insertBookmark("EndOfDoc");
```

It is possible to remove the bookmark markers from the document. The contents between the bookmark markers will be retained.

```
document.getBookmarks().removeBookmark("EndOfDoc");
```

The Bookmarks interface in Python SDK:

To insert a bookmark it is neccessary to have a position object. The position in the end of the document is used in this example. The name of the bookmark should be unique.

```
position = document.getRange().getEnd();
position.insertBookmark("EndOfDoc");
```

It is possible to remove the bookmark markers from the document. The contents between the bookmark markers will be retained.

```
document.getBookmarks().removeBookmark("EndOfDoc");
```

Borders and RangeBorders

The `Borders` class represents the borders of a cell, shape etc.

The `RangeBorders` class inherits `Borders` and represents the borders of a cell range. It extends `Borders` with an ability to set inner borders to another style than the outer ones (left, right, top and bottom).

The Borders interface:

Get the corresponding border properties.

```
const LineProperties& getLeft() const;
const LineProperties& getRight() const;
const LineProperties& getTop() const;
const LineProperties& getBottom() const;
const LineProperties& getDiagonalDown() const;
const LineProperties& getDiagonalUp() const;
```

Get the common properties of left, right, top and bottom borders. All the different properties will be unset (boost::none for C++, None for Python).

```
LineProperties getOuter() const;
```

Get the common properties of down and up diagonal borders. All the different properties will be unset (boost::none for C++, None for Python).

```
LineProperties getDiagonals() const;
```

Set the corresponding border properties. The unset fields of borders (which have boost::none for C++, None for Python) will not be modified.

```
Borders& setLeft(const LineProperties& lineProperties);
Borders& setRight(const LineProperties& lineProperties);
Borders& setTop(const LineProperties& lineProperties);
Borders& setBottom(const LineProperties& lineProperties);
Borders& setDiagonalDown(const LineProperties& lineProperties);
Borders& setDiagonalUp(const LineProperties& lineProperties);
```

Sets the left, right, top and bottom borders. The unset fields of borders (which have boost::none for C++, None for Python) will not be modified.

```
Borders& setOuter(const LineProperties& lineProperties);
Borders& setDiagonals(const LineProperties& lineProperties);
```

The RangeBorders interface:

Get the corresponding inner border properties.

```
const LineProperties& getInnerHorizontal() const;
const LineProperties& getInnerVertical() const;
```

Get the common properties of the inner borders. All the different properties will be unset (boost::none for C++, None for Python).

```
LineProperties getInner() const;
```

Get the common properties of the inner and outer (but not diagonal) borders. All the different properties will be unset (boost::none for C++, None for Python).

```
LineProperties getAll() const;
```

These methods do exactly the same as in the `Border` class, but return `RangeBorders`, that is convenient for chaining methods in a single expression.

```
RangeBorders& setLeft(const LineProperties& lineProperties);
RangeBorders& setRight(const LineProperties& lineProperties);
RangeBorders& setTop(const LineProperties& lineProperties);
RangeBorders& setBottom(const LineProperties& lineProperties);
RangeBorders& setDiagonalDown(const LineProperties& lineProperties);
RangeBorders& setDiagonalUp(const LineProperties& lineProperties);

RangeBorders& setOuter(const LineProperties& lineProperties);
RangeBorders& setDiagonals(const LineProperties& lineProperties);
```

Set the corresponding inner border properties. The unset fields of borders (which have boost::none for C++, None for Python) will not be modified.

```
RangeBorders& setInnerHorizontal(const LineProperties& lineProperties);
RangeBorders& setInnerVertical(const LineProperties& lineProperties);
```

Sets the inner borders. The unset fields of borders (which have boost::none for C++, None for Python) will not be modified.

```
RangeBorders& setInner(const LineProperties& lineProperties);
```

Sets the inner and outer (but not diagonal) borders. The unset fields of borders (which have boost::none for C++, None for Python) will not be modified.

```
RangeBorders& setAll(const LineProperties& lineProperties);
```

Table

The `Table` class represents a table.
Inherits: `Block`

The Table interface in C++ SDK:

To insert a table it is neccessary to have a position object. The position in the end of the document is used in this example. A unique table ID should be returned after a successful operation.

```
auto position = document.getRange().getEnd();
const auto tableID = position.insertTable(3, 3, "My Table");
```

To remove a table it is neccessary to have a table object. The table object can be retrieved from the document by the table ID. Note that the table object becomes invalid after removing table from the document.

```
auto table = document.getBlocks().getTable(tableID);
table->remove();
```

To remove a table it is neccessary to have a table object. The table object can be retrieved from document by the table index in the document. Note that the table object becomes invalid after removing table from document.

```
auto table = document.getBlocks().getTable(0);
table->remove();
```

There is no difference between insertion tables in a text document and worksheets in a workbook document. To insert a worksheet it is neccessary to have a position object. The position in the end of the document is used in this example. A unique table ID should be returned after a successful operation.

```
auto position = document.getRange().getEnd();
const auto tableID = position.insertTable(10, 10, "Sheet");
```

To merge cells it is needed to have a `CellRange` object. The `CellRange` object has methods for merging and unmerging the cells.

```
auto cellRange = table.getCellRange(CellRangePosition(1, 1, 2, 2));
derefPtr(cellRange).merge();
```

To unmerge cells it is needed to have a `Cell` object from merged range. The `Cell` object has method for unmerging all cells from the merged range which this cell is a part of.

```
auto cell = table.getCell(CellPosition(2, 2));
cell.unmerge();
```

[Cpp_SetColumnWidth_example]

The Table interface in Python SDK:

To insert a table it is neccessary to have a position object. The position in the end of the document is used in this example. A unique table ID should be returned after a successful operation.

```
position = document.getRange().getEnd();
position.insertTable(3, 3, "Table");
```

To remove a table it is neccessary to have a table object. The table object can be retrieved from the document by the table ID. Note that the table object becomes invalid after removing table from the document.

```
table = document.getBlocks().getTable(table_id);
table.remove();
```

To remove a table it is neccessary to have a table object. The table object can be retrieved from document by the table index in the document. Note that the table object becomes invalid after removing table from document.

```
table = document.getBlocks().getTable(0);
table.remove();
```

There is no difference between insertion tables in a text document and worksheets in a workbook document. To insert a worksheet it is neccessary to have a position object. The position in the end of the document is used in this example. A unique table ID should be returned after a successful operation.

```
position = document.getRange().getEnd();
id = position.insertTable(10, 10, "Sheet");
```

To merge cells it is needed to have a `CellRange` object. The `CellRange` object has methods for merging and unmerging the cells.

```
cellRange = table.getCellRange(CoreAPI.CellRangePosition(1, 1, 3, 3));
cellRange.merge();
#
```

To unmerge cells it is needed to have a `Cell` object from merged range. The `Cell` object has method for unmerging all cells from the merged range which this cell is a part of.

```
cell = table.getCell(CellPosition(2, 2));
cell.unmerge();
```

The `Table` object has a method for changing the specified column width in points. The following example demonstates how to set the width of the second column to 300 points.

```
table.setColumnWidth(1, 300)
```

CellPosition

The `CellPosition` struct represents a cell position in a table.

```
struct CO_API_SYMBOL CellPosition
{
    CellPosition();
    CellPosition(const std::size_t rowArg, const std::size_t columnArg);

    std::string toString() const;

    std::size_t row;
    std::size_t column;
};
```

CellRangePosition

The `CellRangePosition` struct represents a position of cells range in a table .

```
struct CO_API_SYMBOL CellRangePosition
{
    CellRangePosition();
    CellRangePosition(const CellPosition& topLeftArg, const CellPosition& bottomRightArg);
    CellRangePosition(const std::size_t topLeftRow,
                      const std::size_t topLeftColumn,
                      const std::size_t bottomRightRow,
                      const std::size_t bottomRightColumn);

    std::string toString() const;

    CellPosition topLeft;
    CellPosition bottomRight;
};
```

CellRange

The `CellRange` class represents a set of cells.

The interface:

Returns the top row index of the range.

```
virtual size_t getBeginRow() const = 0;
```

Returns the first column index of the range.

```
virtual size_t getBeginColumn() const = 0;
```

Returns the last row index of the range (not the index after the last one).

```
virtual size_t getLastRow() const = 0;
```

Returns the first column index of the range (not the index after the last one).

```
virtual size_t getLastColumn() const = 0;
```

Returns `Enumerator` which iterates all cells.

```
virtual std::shared_ptr<Enumerator<Cell>> getEnumerator() = 0;
```

Sets the borders of the cells in the range.

```
virtual void setBorders(const RangeBorders& borders) = 0;
```

Function to set cell properties for range.

```
virtual void setCellProperties(const CellProperties& cellProperties) = 0;
```

To get cell properties for range. Returns structure with only the same properties for all cell in range.

```
virtual CellProperties getCellProperties() const = 0;
```

Merge cells in range.

```
virtual void merge() = 0;
```

Cell

The `Cell` class represents a table cell with its content, formatting, formulas etc.

The interface:

Cell format property. Returns a range containing the cell.

```
Range getRange();
```

The cell format property (numeric, text, date etc.).

```
CellFormat getFormat() const;
void setFormat(const CellFormat format);
```

Returns format string from the cell.

```
std::string getCustomFormat() const;
```

Sets format to the cell. Throws: `ParseError`.

```
void setCustomFormat(const std::string& formatString);
```

Sets corresponding value to the cell.

```
void setBool(const bool value);
void setNumber(const double value);
void setText(const std::string& value);
```

Enters a formula into the cell. Throws: `ParseError`.

```
void setFormula(const std::string& value);
```

Gets formula text.

```
std::string getFormulaAsString() const;
```

Returns corresponding formatted cell value.

```
std::string getFormattedValue() const;
```

Determines and sets corresponding value and format to the cell. Sets text if autodetect failed.

```
void setFormattedValue(const std::string& value);
```

Cell decoration properties accessors (like background, vertical alignment etc.).The unset properties (which have boost::none for C++, None for Python) will not be modified.

```
CellProperties getCellProperties() const;
void setCellProperties(const CellProperties& cellProperties);
```

Sets the borders of the cell.

```
void setBorders(const Borders& borders);
```

Unmerge cells in range which this cell is part of.

```
void unmerge();
```

CellProperties

`CellProperties` contains decoration properties, related to a cell.

```
struct CellProperties
{
    boost::optional<VerticalAlignment> verticalAlignment;
    boost::optional<ColorRGBA> backgroundColor;
    boost::optional<TextLayout> textLayout;
};
```

Setting cell properties in C++ SDK:

To set cell properties, it is necessary to create an instance of `CellProperties` and initialize the necessary fields. E.g.:

```
CellProperties cellProperties;
cellProperties.verticalAlignment = VerticalAlignment::Center;
cellProperties.backgroundColor = ColorRGBA(255, 0, 0, 0);
cell.setCellProperties(cellProperties);
```

However, in C++ it can be done in a shorter way with `CellPropertiesBuilder`:

```
cell.setCellProperties(CellPropertiesBuilder()
                           .setVerticalAlignment(VerticalAlignment::Center)
                           .setBackgroundColor(ColorRGBA(255, 0, 0, 0)));
```

`CellPropertiesBuilder` allows to construct `CellProperties` with a single expression, that is convenient, when it is necessary to set a few cell properties at once.

```
class CO_API_SYMBOL CellPropertiesBuilder
{
public:
    CellPropertiesBuilder& setVerticalAlignment(const boost::optional<VerticalAlignment>& verticalAlignment);
```

```
    CellPropertiesBuilder& setBackgroundColor(const boost::optional<ColorRGBA>& backgroundColor);
    CellPropertiesBuilder& setTextLayout(const boost::optional<TextLayout>& textLayout);

    operator const CellProperties&() const;
};
```

Note, that this class is available only in C++ SDK and not available through the bindings to the other languages.

Setting cell properties in Python SDK:

To set cell properties, it is necessary to create an instance of `CellProperties` and initialize the necessary fields. E.g.:

```
cell_properties = CoreAPI.CellProperties()
cell_properties.verticalAlignment = CoreAPI.VerticalAlignment_Center
cell_properties.backgroundColor = CoreAPI.ColorRGBA(255, 0, 0, 0)
cell.setCellProperties(cell_properties)
```

However, in Python it can be done in a shorter way with a special constructor, accepting kwargs. E.g.:

```
cell.setCellProperties(CoreAPI.CellProperties(
    verticalAlignment = CoreAPI.VerticalAlignment_Center,
    backgroundColor = CoreAPI.ColorRGBA(255, 0, 0, 0)
))
```

Note, that this constructor is available only in Python SDK.

CellFormat

The `CellFormat` enum contains all supported cell formats.

```
enum class CellFormat : uint8_t
{
    General,
    Percentage,
    Number,
    Text,
    Currency,
    Accounting,
    Date,
    Time,
    DateTime,
    Fraction,
    Scientific,
    Custom
};
```

Setting cell format in C++ SDK:

To get cell format properties as CellFormat, getFormat() function is used. The cell format is detected automatically, when formatted value is inserted. E.g.:

```
cell.setFormattedValue("1E+37");

const auto& format = cell.getFormat();
EXPECT_EQ(API::Document::CellFormat::Scientific, format);
```

It is possible to set CellFormat for the cell. Then it is applied to the cell value instead of the existed format. E.g.:

```
const auto& inputFormat = API::Document::CellFormat::Scientific;

cell.setNumber(1.0);
cell.setFormat(inputFormat);

const auto& formatString = cell.getFormat();
EXPECT_EQ(inputFormat, formatString);
```

To get cell format properties as format string (e.g. "# ?/?"), getCustomFormat() function is used.

```
cell.setFormattedValue("1E+37");

const auto& formatString = cell.getCustomFormat();
EXPECT_EQ("0.00E+00", formatString);
```

getCustomFormat() function sets cell format properties via format string or throws an exception, if format string is incorrect. Usage example:

```
const auto& inputFormatString = "0.00E+00";

cell.setNumber(1.0);
cell.setCustomFormat(inputFormatString);

const auto& formatString = cell.getCustomFormat();
EXPECT_EQ(inputFormatString, formatString);
```

Charts

The `Charts` class provides an interface for an access to the charts collection.

The interface:

Returns the number of charts.

```
std::size_t getChartsCount() const;
```

Returns a chart by chart index (not by drawing index).
Throws: `OutOfRangeError`.

```
Chart getChart(std::size_t chartIndex);
```

Returns chart index by drawing index.
Returns `boost::none` if drawing by given index is a picture or index is out of range.

```
boost::optional<size_t> getChartIndexByDrawingIndex(std::size_t drawingIndex);
```

Chart

The `Chart` class represents a chart in a document with all its elements (Title, Legend, Type, Data, Range etc.).

The interface:

The chart type property.

```
ChartType getType() const;
void setType(ChartType chartType);
```

Returns chart series count.

```
std::size_t getRangesCount() const;
```

Returns a `ChartRangeInfo` by index.

```
ChartRangeInfo getRange(std::size_t rangesIndex) const;
```

Returns chart title.

```
boost::optional<std::string> getTitle() const;
```

Sets chart range.

```
void setRange(const CellRangePosition& chartRange);
```

Sets chart rect.

```
void setRect(const Rect& rect);
```

Returns 'true' if there are no values in chart.

```
bool isEmpty() const;
```

Returns 'true' whenever chart ranges can be selected by an one rect without gaps.

```
bool isSolidRange() const;
```

Returns 'true' if chart is 3D

```
bool is3D() const;
```

Returns chart series direction type

```
SeriesDirectionType getDirectionType() const;
```

Returns ChartLabelsInfo: categories labels mode, series name labels mode and identifier for chart constructed by

```
// one row/column
ChartLabelsInfo getChartLabels() const;
```

Returns data range string for chart.

```
std::string getRangeAsString() const;
```

Apply chart settings to currently selected chart: range info(new chart range and table name for range), series direction type, chart title and labels info(series name and categories)

```
void applySettings(const boost::optional<TableRangeInfo>& rangeInfo,
                   boost::optional<SeriesDirectionType> directionType,
```

```
                const boost::optional<std::string>& title,
                boost::optional<ChartLabelsInfo> labelsInfo);
```

ChartRangeType

`ChartRangeType` describes a type of data range for a chart.

```
enum class ChartRangeType : std::uint8_t
{
    Series,
    SeriesName,
    Categories,
    DataPoint,
};
```

LabelsDetectionMode

`LabelsDetectionMode` describes the mode of automatic detection of labels for a chart.

```
enum class LabelsDetectionMode : std::uint8_t
{
    Unknown,
    FirstRow,
    FirstColumn,
    NoLabels,
};
```

ChartLabelsInfo

`ChartLabelsInfo` describes chart labels.

```
struct CO_API_SYMBOL ChartLabelsInfo
{
    ChartLabelsInfo(const LabelsDetectionMode categories, const LabelsDetectionMode seriesName, const bool oneColumnRow)
        : categoriesMode(categories)
        , seriesNameMode(seriesName)
        , isOneColumnRowChart(oneColumnRow)
    {
    }

    LabelsDetectionMode categoriesMode;
    LabelsDetectionMode seriesNameMode;
    bool isOneColumnRowChart = false;
};
```

ChartRangeInfo

`ChartRangeInfo` describes a chart serie.

```
struct CO_API_SYMBOL ChartRangeInfo
{
    ChartRangeInfo(const TableRangeInfo& rangeInfo,
                   const ColorRGBA& color,
                   const bool hidden,
                   const ChartRangeType type)
        : tableRangeInfo(rangeInfo)
        , rangeColor(color)
        , isHidden(hidden)
        , rangeType(type)
    {
    }

    TableRangeInfo tableRangeInfo;
    // Series color.
    ColorRGBA rangeColor;

    bool isHidden = false;

    ChartRangeType rangeType;
};
```

TableRangeInfo

`TableRangeInfo` describes new chart range.

```
struct CO_API_SYMBOL TableRangeInfo
{
    TableRangeInfo(const CellRangePosition& range, const ID& id)
        : tableRange(range)
        , tableID(id)
    {
    }

    CellRangePosition tableRange;
    ID tableID;
```

```
};
```

ChartType

The `ChartType` enum contains all supported chart types.

```
enum class ChartType : std::uint8_t
{
    Unknown = 0,
    Bar,
    BarStacked,
    BarPercentStacked,
    Column,
    ColumnStacked,
    ColumnPercentStacked,
    Line,
    LineStacked,
    LinePercentStacked,
    LineWithMarker,
    LineWithMarkerStacked,
    LineWithMarkerPercentStacked,
    Area,
    AreaStacked,
    AreaPercentStacked,
    Pie,
    PieExploded,
    Scatter
};
```

ScriptPosition

`ScriptPosition` contains all the possible scripts of the text (subscript, superscript or normal).

```
enum class ScriptPosition : std::uint8_t
{
    SuperScript,
    SubScript,
    NormalScript,
};
```

SeriesDirectionType

The `SeriesDirectionType` enum contains chart series direction type.

```
enum class SeriesDirectionType : std::uint8_t
{
    EnumFirst,
    Unknown = EnumFirst,
    ByRow,
    ByColumn,
    EnumLast = ByColumn
};
```

TextLayout

`TextLayout` contains all the possible options of layout text in a cell, shape etc.

```
enum class TextLayout : std::uint8_t
{
    // Always layout text in a single line (if text is longer than the object, containing it, the text just overlaps
    // with the neighboring objects).
    SingleLine,
    // Text is wrapped by word boundraries, the height of the object, containing it, is increased to the necessary size.
    WrapByWords,
    // Text size is chosen in the way, the text fits the width of the object, containing it.
    ShrinkSizeToFitWidth,
};
```

VerticalAlignment

`VerticalAlignment` contains all vertical alignments for a cell, shape etc.

```
enum class VerticalAlignment : std::uint8_t
{
    Bottom,
    Center,
    Top,
};
```

Paragraph

The `Paragraph` class represents a paragraph.
Inherits: `Block`

The interface:

Paragraph properties accessors (like alignment, indents, spacing). The unset properties (which have boost::none for C++, None for Python) will not be modified.

```
ParagraphProperties getParagraphProperties();
void setParagraphProperties(const ParagraphProperties& properties);
```

LineStyle

`LineStyle` contains all the available styles of any line (e.g. border, line in shape, text underlining line etc).

```
enum class LineStyle : uint8_t
{
    NoLine,
    Solid,
    Dot,
    Dash,
    LongDash,
    DashDot,
    DotDotDash,
    Double,
    Wave,
};
```

LineProperties

`LineProperties` contains decoration properties, related to a line.

```
struct LineProperties
{
    boost::optional<LineStyle> style;
    boost::optional<LineWidth> width;
    boost::optional<ColorRGBA> color;
};
```

```
using LineWidth = float;
```

A convenient constant for invisible line (i.e. a line with the `NoLine` style).

```
extern const LineProperties NoLineProperties;
```

`LinePropertiesBuilder` allows to construct `LineProperties` with a single expression, that is convenient, when it is necessary to set a few text properties at once.

```
class CO_API_SYMBOL LinePropertiesBuilder
{
public:
    LinePropertiesBuilder& setStyle(const boost::optional<LineStyle>& style);
    LinePropertiesBuilder& setWidth(const boost::optional<LineWidth>& width);
    LinePropertiesBuilder& setColor(const boost::optional<ColorRGBA>& color);

    operator const LineProperties&() const;
};
```

Note, that this class is available only in C++ SDK and not available through the bindings to the other languages.

For properties and builder usage examples see the other properties, for instance `TextProperties`.

Alignment

The `Alignment` enum contains all supported alignment types.

```
enum class Alignment : std::uint8_t
{
    Default,
    Left,
    Center,
    Right,
    Justify,
    Distributed,
    Fill
};
```

Shape

The `Shape` class represents a shape.
Inherits: `Block`

The interface:

Field

The `Field` class represents an autogenerated field block (e.g. table of contents).

Inherits: `Block`

The interface:

Position

The `Position` class represents a location within a document. It is intended to abstract away implementation details of the document. Position updates it's state after any document changes.

The interface:

Inserts text into the position.

```
void insertText(const std::string& text);
```

Inserts table into the position. Throws: `DocumentModificationError`.

```
ID insertTable(size_t rowsCount, size_t columnsCount, const std::string& name);
```

Inserts a page break into the position.

```
void insertPageBreak();
```

Inserts a line break into the position. Note, however, that it does not split the paragraph.

```
void insertLineBreak();
```

Inserts named bookmark into the position.

```
void insertBookmark(const std::string& name);
```

Range

The `Range` class manages information about a range of positions in a document and represents in terms of [begin, end).

The interface:

The range begin position read-only property.

```
Position getBegin() const;
```

The range end position read-only property.

```
Position getEnd() const;
```

Returns textual representation of the content in the range. Note that some of the content (e.g. pictures) will be skipped, some (e.g. tables) will be transformed.

```
std::string extractText() const;
```

Removes the content of the range.

```
void removeContent();
```

Replaces the content of the range with the specified text.

```
void replaceText(const std::string& text);
```

Returns the decoration properties of the text in the range. The properties that have mixed values will be unset (boost::none for C++, None for Python).

```
TextProperties getTextProperties() const;
```

Text decoration properties accessors. The unset properties (which have boost::none for C++, None for Python) will not be modified.

```
void setTextProperties(const TextProperties& textProperties);
```

Search

The `Search` class provides an interface for performing searches in a document.

The interface:

Provides a case-insensitive text search in a document.

```
virtual std::shared_ptr<Enumerator<Range>> findText(const std::string& text) = 0;
```

Provides a case-insensitive text searching in the specified range.

```
virtual std::shared_ptr<Enumerator<Range>> findText(const std::string& text, const Range& range) = 0;
```

Global functions:

Creates a new `Search` instance with the specified `Document` object.

```
CO_API_SYMBOL std::shared_ptr<Search> createSearch(const Document& document);
```

TextProperties

`TextProperties` contains decoration properties, related to a fragment of text (range). It can be either the whole paragraph, or a part of it, or even the text in a few cells in table. Text properties can be applied to any `Range`.

```
struct TextProperties
{
    boost::optional<std::string> fontName;
    boost::optional<FontSize> fontSize;
    boost::optional<bool> bold;
    boost::optional<bool> italic;
    boost::optional<bool> underline;
    boost::optional<bool> strikethrough;
    boost::optional<bool> allCapitals;
    boost::optional<ScriptPosition> scriptPosition;
    boost::optional<ColorRGBA> textColor;
    boost::optional<ColorRGBA> backgroundColor;
    boost::optional<CharacterSpacingSize> characterSpacing;
};
```

```
using FontSize = float;
using CharacterSpacingSize = float;
```

Setting text properties in C++ SDK:

To set properties for a text, it is necessary to create an instance of `TextProperties` and initialize the necessary fields. E.g.:

```
TextProperties textProperties;
textProperties.bold = true;
textProperties.textColor = ColorRGBA(255, 0, 0, 0);
document.getBlocks().getParagraph(0)->getRange().setTextProperties(textProperties);
```

However, in C++ it can be done in a shorter way with `TextPropertiesBuilder`:

```
document.getBlocks().getParagraph(0)->getRange().setTextProperties(
    TextPropertiesBuilder()
        .setBold(true)
        .setTextColor(ColorRGBA(255, 0, 0, 0)));
```

`TextPropertiesBuilder` allows to construct `TextProperties` with a single expression, that is convenient, when it is necessary to set a few text properties at once.

```
class CO_API_SYMBOL TextPropertiesBuilder
{
public:
    TextPropertiesBuilder& setFontName(const boost::optional<std::string>& fontName);
    TextPropertiesBuilder& setFontSize(const boost::optional<FontSize>& fontSize);
    TextPropertiesBuilder& setBold(const boost::optional<bool>& bold);
    TextPropertiesBuilder& setItalic(const boost::optional<bool>& italic);
    TextPropertiesBuilder& setUnderline(const boost::optional<bool>& underline);
    TextPropertiesBuilder& setStrikethrough(const boost::optional<bool>& strikethrough);
    TextPropertiesBuilder& setAllCapitals(const boost::optional<bool>& allCapitals);
    TextPropertiesBuilder& setScriptPosition(const boost::optional<ScriptPosition>& scriptPosition);
    TextPropertiesBuilder& setTextColor(const boost::optional<ColorRGBA>& textColor);
    TextPropertiesBuilder& setBackgroundColor(const boost::optional<ColorRGBA>& backgroundColor);
    TextPropertiesBuilder& setCharacterSpacing(const boost::optional<CharacterSpacingSize>& characterSpacing);

    operator const TextProperties&() const;
};
```

Note, that this class is available only in C++ SDK and not available through the bindings to the other languages.

Setting text properties in Python SDK:

To set text properties, it is necessary to create an instance of `TextProperties` and initialize the necessary fields. E.g.:

```
text_properties = CoreAPI.TextProperties()
text_properties.bold = True
text_properties.textColor = CoreAPI.ColorRGBA(255, 0, 0, 0)
document.getBlocks().getParagraph(0).getRange().setTextProperties(text_properties)
```

However, in Python it can be done in a shorter way with a special constructor, accepting kwargs. E.g.:

```
document.getBlocks().getParagraph(0).getRange().setTextProperties(CoreAPI.TextProperties(
    bold = True,
    textColor = CoreAPI.ColorRGBA(255, 0, 0, 0)
))
```

Note, that this constructor is available only in Python SDK.

Exceptions

`DocumentModificationError` is thrown when it's not possible to perform a document modification operation.

```
class CO_API_SYMBOL DocumentModificationError : public BaseError
{
public:
    DocumentModificationError(const std::string& reason);
};
```

`PositionDocumentsMismatchError` is thrown when a few positions refer to different documents and, thus, cannot be used in one operation.

```
class CO_API_SYMBOL PositionDocumentsMismatchError : public BaseError
{
public:
    PositionDocumentsMismatchError();
};
```

Bookmarks

The `Bookmarks` class provides an interface for an access to document's bookmarks.

The interface:

Returns a range for a bookmark content or boost::none if no such bookmark.

```
boost::optional<Range> getBookmarkRange(const std::string& bookmarkName);

void removeBookmark(const std::string& bookmarkName);
```

Scripting

Scripting

The `Scripting` class manages the Lua virtual machine. It provides an interface for running the Lua scripts which are stored in the `Document` instance.

The interface:

Runs a script with the specified name.
Throws: `ScriptExecutionError`.

```
virtual void runScript(const std::string& name) = 0;
```

Global functions:

Creates the `Scripting` instance with the specified `Document` object.

```
CO_API_SYMBOL std::shared_ptr<Scripting> createScripting(const Document::Document& document);
```

API Versioning

API Version mechanism

The API Version constants allow to determine if the previous and the current versions of API are compatible or not.
If the `Major` part of API version was changed, there were backward incompatible changes in API and thus client code has to be reviewed and possibly updated. See the "Incompatible API modification" section for the details, which parts of API were changed. Backward incompatible changes include renaming, removing or incompatible signature changing existing classes or methods.
If the `Minor` part of API version was changed, there were only backward compatible changes in API and there is no need to change the client code to make it work with a newer version of API. Note, that currently, only compatibility on C++ source level is guaranteed in this case and no binary compatiblity should be expected. Thus, it is necessary to recompile the client code with a newer library version. Backward incompatible changes include adding new methods, types and class members.

It is recommended to check the API Version before the initialization like this:

```
unsigned int ExpectedMajorAPIVersion = 1;
unsigned int ExpectedMinorAPIVersion = 0;
if (!Version::isAPIVersionCompatible(ExpectedMajorAPIVersion, ExpectedMinorAPIVersion))
{
    // Report a fatal error about incompatible library version and exit.
}

// Start working with the library (create Application and so on).
```

Basic usage:

```
from CollabioPythonSDK import CoreAPI

if __name__ == '__main__':
    expected_major_api_version = 1
    expected_minor_api_version = 0
    if not CoreAPI.isAPIVersionCompatible(expected_major_api_version, expected_minor_api_version):
        # Report a fatal error about incompatible library version and exit.
        pass

    # Start working with the library (create Application and so on).
```

Current API versions are:

```
const unsigned int Major = 1;
const unsigned int Minor = 0;
```

Check if the specified API version is compatible with the current one:

```
bool isAPIVersionCompatible(unsigned int major, unsigned int minor);
```

Note, that both the constants and the function are in the `Version` namespace.

Incompatible API modification

No backward incompatible changes have been made yet.