Documentation

Release 3.3.2.2

XSQUARE - REPORTS

Jun 19, 2024

CONTENT

CHAPTER

ONE

GENERAL INFORMATION

1.1 Content of documentation

See the chapters of the report server documentation in the table of Contents.

1.2 Purpose of the report server

The main functions of the report server are:

- generation of reports in DOCX, XLSX, PDF formats;
- merging PDF documents;
- generation of a printed PDF form based on the previously generated PDF document with a stamp.

1.3 Report server modules

The report server includes the following modules:

- *Service for generating reports from a template document*
- *PDF report generator*
- *PDF document merging service*
- *Print form generation service*

1.4 Operating systems

Program was tested on the following x86-64 operating systems:

- Ubuntu 18+
- Red Hat 8+
- Debian 10+
- CentOS 7+

1.5 Report server installation

The process of the Report Server installation is described in the section *Installation*

1.6 Usage examples

Examples of requests and templates are given in the section *Examples*.

1.7 Getting the first report

The process of creating a simple report using the report server is described in the section *My first report*.

CHAPTER TWO

ARCHITECTURE AND SYSTEM REQUIREMENTS

2.1 Architecture

The architecture of the application is a web server processing client HTTP requests by generating reports based on template documents and returning the result to the client.

2.2 Runtime environment

Supported architectures:

- x86-64
- ARM

Supported OS:

- Ubuntu 18+
- Red Hat 8+
- Debian 10+
- CentOS 7+

2.3 System requirements

Minimal system requirements:

- CPU - 1 Core
- RAM - 100 MB
- HDD - 100 MB + Logs

Installation of virtualization/containerization system and operating system is carried out at the discretion of the Administrator based on the needs of the Organization.

CHAPTER

THREE

INSTALLATION

The program was tested on the following operating systems:

- Ubuntu 18+
- Red Hat 8+
- Debian 10+
- CentOS 7+

3.1 Installation

```
mkdir -p /usr/local/xsquare.xreports
```

3.1.1 Content transfer

```
scp -r  xreports assets/ templates/  root@[host]/usr/local/xsquare.xreports
```

3.1.2 Installing Libre Office

We recommend installing Libre Office distributions version 7.4.7.2 from the vendor's website (not from the OS repository):

```
tar -xvzf LibreOffice_7.4.7.2_Linux_x86-64_rpm.tar.gz
cd LibreOffice_7.4.7.2_Linux_x86-64_rpm/
cd RPMS/
yum install *.rpm
find  / -name "soffice"
vi /usr/local/xsquare.xreports/config.json
```

3.1.3 Configuring Systemctl

```
vi /etc/systemd/system/xsquare.xreports.service

[Unit]
Description=XSQUARE-Reports
After=syslog.target
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/xsquare.xreports/xreports
WorkingDirectory=/usr/local/xsquare.xreports
User=root

[Install]
WantedBy=multi-user.target
```

Starting services

```
systemctl daemon-reload
systemctl enable xsquare.xreports
systemctl start xsquare.xreports
systemctl status xsquare.xreports
```

3.1.4 Installing Fonts for RPM based OS

```
dnf install msttcore-fonts-installer
fc-cache -fv
```

3.1.5 Configuration file config.json

The config.json configuration file must be present in the directory with the service before starting the Program. The configuration file contains 3 sections:

1. The App descriptor where one can define the basic service settings:

```
    {
"app": {
    "port": "8087",
    "debug": false
},
```

- “port” - string defines the number of the network port on which the service will be started (by default - 8087)
- “debug” - лBoolean value that enables debugging mode in which a detailed requests processing log is available, as well as all the requests and finished documents are saved in the local service directory **reports_debug** ((similar to using the flag *enable-debug-report-save* in the request properties). The default value is - **false**.

2. The FormatConversion descriptor which defines the settings for the Libre Office soffice application:

```
"formatConversion": {
        "format-conversion-dir": "",
        "soffice-max-process-count": 0,
        "soffice-path": ""
},
```

- "format-conversion-dir" - string defines a directory for storing temporary files (by default -"/tmp")
- "soffice-max-process-count" - number which defines the quantity of processor cores that soffice can use for conversion (default is "0" to use all available cores)
- "soffice-path" - string which specifies the path to the soffice executable file (by default - the path added to environment variables when installing Libre Office)

FUNCTIONAL CHARACTERISTICS

XSQUARE-REPORTS – is an independent web service designed to generate reports in different formats from template documents by HTTP request in XML or JSON formats based on requests with data and document templates in different formats.

The main functional characteristics and features of the XSQUARE-REPORTS report server:

- report generation in OOXML format (*.docx and *.xlsx documents) from a template document by HTTP request in XML or JSON format
- report generation in PDF format by HTTP request in JSON format
- merging documents in PDF format into one document by HTTP request in JSON format
- generation of the printed form of the document in PDF format by HTTP-request in JSON format

CHAPTER

FIVE

LIFE CYCLE

5.1 General information

The chapter describes processes that maintain the XSQUARE- REPORTS software lifecycle, including troubleshooting of issues identified during software operation, software enhancements, and information on staffing needs for Program maintenance.

Program	XSQUARE-REPORTS Report Server
Developer	LLC XSQUARE
User	A legal entity using the Program under an agreement with the Developer
User	https://xsquare.ru

5.2 Maintaining Program life cycle

The Program life cycle includes the following stages:

1. Design and development of the Program by the Developer.
2. Testing and detection of malfunctions in the Program operation by the Developer.
3. Installation, operation and installing Program updates by the User in accordance with the license agreement with the Developer.
4. Upgrading the Program by the Developer according to the plan of improvements and enhancements, as well as pursuant to the requests of the User.
5. The Developer provides technical support to the User regarding the installation, integration, and operation of the Program.
6. The Developer releases the updated versions of the Program.
7. The Developer manages all stages of the Program life cycle except for installation, integration, and operation of the Program by the User.

5.3 Troubleshooting while using the Program

Malfunctions detected during the Program operation can be troubleshooted in the following ways:

1. the Developer making corrections to the Program code according to the Program development roadmap;
2. the Developer making corrections to the Program code based on the User's request.

The User can generate the following requests:

- An incident report with provided information on the conditions of the bug occurrence with supporting graphical information, log files, information on the software environment and the software version numbers, including the version and release number of the Program. The request shall also contain information about the expected and actual behavior of the Program and any other information that will help the Developer to diagnose and fix the Program malfunction.
- Request to improve the Program to change its behavior to achieve the desired results by the User.
- Request for additional information on the operation and capabilities of the Program.

Requests can only be sent by the User using the Developer's tracking system - https://tracker.xsquare.ru. Access to the tracking system is provided upon purchase of the Program.

The Developer accepts and records all requests of the User. Each request is assigned a unique number allowing to trace the communication history between the User and the Developer.

The Developer informs the User about the new functionality of the Program or about adding the task to the Program development roadmap.

The Developer has the right to request from the User any additional information that may be useful for troubleshooting the Program.

If the User fails to provide or provides insufficient information requested by the Developer, the latter has the right to suspend making the required changes to the Program code.

5.4 Program improvement

The Program is continuously improved and upgraded, regular updates are released, information materials are published on the Program Website, Users are informed about changes in the Program.

The User may initiate a request to change or improve the operation of the Program by submitting a request to the Developer.

Requests can be sent by the User using the tracking system https://tracker.xsquare.ru

The Developer accepts and records all requests of the User. Each request is assigned a unique number allowing to trace the communication history between the User and the Developer.

The Developer informs the User about the new functionality of the Program or about adding the task to the Program development roadmap.

The Developer has the right to request from the User any additional information that may be useful for troubleshooting the Program.

If the User fails to provide or provides insufficient information requested by the Developer, the latter has the right to suspend making the required changes to the Program code.

5.5 Technical support of the Program

Technical support of the Program is carried out by sending requests to the Developer. Requests can be sent by the User using the tracking system https://tracker.xsquare.ru.

The Developer accepts and records all requests of the User. Each request is assigned a unique number allowing to trace the communication history between the User and the Developer.

The User technical support includes:

- Assistance in Program installation.
- Assistance in basic system-wide components installation (operating system, HTTP Server, database server, etc.).
- Assistance in integrating the Program into the User's existing solutions.
- Assistance in troubleshooting malfunctions detected in the Program operation.
- Advice and guidance on the operation of the Program.
- Collecting information on incorrect Program operation for subsequent Program upgrades according to the improvement plan.
- Informing the User about Program updates.

5.6 Staffing needs for Program maintenance

Personnel operating the Program from the User end should possess the following qualifications:

1. Basic skills in administering Unix family operating systems;
2. Basic skills in working with office packages;
3. Proficiency in using PCs and web browsers.

In case of any questions, personnel should contact the Developer for technical support.

CHAPTER SIX

OPERATION

This section describes how to keep the application running and the sequence in which the components are loaded.

6.1 Report server

To run xsquare.xreports, the user must ensure that a properly set up configuration file is available.

```
cat /usr/local/xsquare.xreports/config.json
```

The command should display the correct configuration file described in the “Installation” section. Next, the report server should be started by executing the command:

```
systemctl start xsquare.xreports
```

Afterwards, one should check the status of the application server:

```
systemctl status xsquare.xreports
```

If errors occur, they will be logged. One can check the error messages by executing the command:

```
journalctl -u xsquare.xreports
```

CHAPTER

SEVEN

MODULES

7.1 Service for generating reports from a template document

7.1.1 General description of the service

The service generates a report in OOXML format (*.docx and *.xlsx documents) from a template document by HTTP request in XML or JSON format. Optionally, the report can be converted to PDF.

Algorithm of the service operation: tags of the template document are replaced by specific data indicated in the request.

Main features of the service

1. Generating reports in DOCX format based on a template document. It is possible to generate:
    - simple reports in DOCX format;
    - multi-reports in DOCX format (reports from one template based on multiple input data).
2. Generating reports in XLSX format based on a template document.
3. Converting reports to PDF format.

7.1.2 Working with template documents

To generate a report, one needs to create a template in DOCX or XLSX format.

Location of template files

Template documents in DOCX and XLSX formats are stored locally in the *templates* directory located in the application directory of the service.

Template file format

To create template documents, one should use a word processor (MS Word/Excel, P7-Office, LibreOffice Writer/Calc, Yandex Documents, etc.).

Important:

- Templates for generating documents in DOCX format must be saved in DOCX format.
- Templates for generating documents in XLSX format must be saved in XLSX format.
- Templates for subsequent conversion to PDF should be saved in DOCX format (preferably using Libre Office, as the resulting PDF will be fully consistent with the visual representation of the document in Libre Office).

Using tags in the template

Document templates can contain tags that will be replaced with input data from the request. The tag is specified in square brackets. Example: [debt].

In the body of the request, the input data for the tags is contained in input-data. No brackets are specified in the request (see examples from "Request structure" in the "Generating a report in DOCX format" subsection and from "Request structure" in the "Generating a report in XLSX format" subsection).

Working with images

There is the following principle of working with images:

- an image intended for subsequent replacement is added to the template;
- When the document is generated, it is replaced by an image file that is passed in the request. The new content of the image file encoded in BASE64 is passed in the request.

Image tags in the template are set in the image property "Alt text" without square brackets.

Supported formats: JPEG, PNG.

Limitations for images

1. The image format (JPEG, PNG) in the request must match the format of the image being replaced in the template.
2. If it is necessary to replace several template images with different images from the input data, the images in the template must also be different since the image in the document body refers to its content (file), which means that replacing the content in the document will change all the images that refer to this content.

Examples of templates

Example templates can be found in the *examples archive*

Examples of templates with images:

For documents in DOCX format:

- barcode_code-128.docx
- example_from_xml.docx
- example_from_json.docx

For documents in XLSX format:

- barcode_code-128.xlsx
- base_example.xlsx

7.1.3 Generating a report in DOCX format

Service description

Service name	Service for generating reports from a template document in **DOCX** format
Path to service	Simple reports in DOCX format: • [host]:[port]/word_report_json • [host]:[port]/word_report_xml Multi-reports in DOCX format: • [host]:[port]/word_multi_report_1_N_json • [host]:[port]/word_multi_report_1_N_xml
Method	POST
Parameters	The request body must contain an object in JSON or XML format. One can read more about the structure of the request body in the "Request body structure" subsection. In response, the service returns a document file in DOCX format encoded in base64. When converting a report to PDF format, the service returns a document file in PDF format encoded in base64.
Purpose	The service is designed to generate reports from a template document in **DOCX** format. Conversion to PDF is possible.

Request body structure

The body of the request contains an object in JSON or XML format that includes:

1. Template descriptor.
2. Input data to substitute into the template instead of tags.
3. Request options.
4. Response format.

Template descriptor

The element (object) of the `template` descriptor contains:

- `uri` - string which specifies the location of the template document file depending on how the `id` parameter is interpreted. Supported value: *local*.
- `id` - string, Template identifier. It specifies the path to the template document file relative to the `templates` service directory. It is also used to write a report file in debug mode and to identify a request in the log.

Example for XML format:

```xml
<template uri="local" id="Newsletter"/>
```

Example for JSON format:

```
      "template":
{
    "uri": "local",
    "id": "template_example_1"
}
```

Input-data

The input data element (object) `input-data` contains:

- Simple string data to substitute into the template instead of tags.
- Values for conditional expressions.
- Descriptors.
- List Descriptors.
- Data to be substituted into image tags.
- Block Descriptors.

Note: The difference between the data structure for JSON and XML is that instead of XML elements, data is represented as objects and lists of objects. Tag names are specified by object field names. Lists, tables, images are child objects with respect to the `input-data` object.

Note: For multi-reports the `input-data-array` list (array) is used, which contains elements (objects) with the same structure as `input-data`. For more details on multi-reports, please refer to the subsection "Working with a multi-report document".

Simple string data to be substituted into the template instead of tags

For XML format

The `tags` element contains a list of child `tag` elements with the `name` attribute and the desired specific value with which the tag in the template will be replaced when generating the report. The `name` attribute matches the tag in the template.

Example:

```
<input-data>
<tags>
        <tag name="ORGANIZATION">JSC «xxxxyyyy»</tag>
</tags>
</input-data>
```

For JSON format

Tag names are given by the names of the objects within the `input-data`.

Example:

```
 "input-data":
         {
"ORGANIZATION": "JSC «xxxxyyyy»"
 }
```

Description of tables

For XML format

The tables descriptor element contains a list of child table elements with the name attribute. The name attribute matches the tag in the template. The tag in the template will be replaced by the table created according to the table description in the request.

Example:

```
<input-data>
    <tables>
        <table name="TABLE-FORMATTED">
                   <rows>
                       <row>
                           <cell tag="NO">1</cell>
                           <cell tag="agreement_number">Example number 1</cell>
                           <cell tag="district">Example district 1</cell>
                           <cell tag="enterprise">Example enterprise 1</cell>
                           <cell tag="disconnect_date">Example date 1</cell>
                           <cell tag="address">Example address 1</cell>
                           <cell tag="complex_field">Example sum 1</cell>
                       </row>
                   </rows>
               </table>
               <table name="TABLE-NO-FORMAT">
                   <header>
                       <cell>No</cell>
                       <cell>Case No. in the Arbitration Court</cell>,
                       <cell>Debt period</cell>,
                       <cell>Principal debt, USD incl.VAT</cell>,
                       <cell>Interest, USD</cell>,
                       <cell>Fee, USD</cell>,
                       <cell>Penalty by court decision</cell>,
                       <cell>Total repayment amount, USD</cell>,
                       <cell>Maturity, not later</cell>
                   </header>
                   <rows>
                       <row>
                           <cell>1</cell>,
                           <cell>A12-1234/2030</cell>,
                           <cell>October 2017</cell>,
                           <cell>12 345,58</cell>,
                           <cell/>
                           <cell>23 456,00</cell>,
                           <cell>78 912,41</cell>,
                           <cell/>
                           <cell>31.08.2019</cell>
                       </row>
                   </rows>
               </table>
```

(continues on next page)

(continued from previous page)

```
            </tables>
</input-data>
```

For JSON format

Table descriptors are child objects of the input-data object. The field names of the table objects match the tags in the template.

Example:

```
"input-data":
 {
     "TABLE-NO-FORMAT":
     {
         "header":
         [
             "No",
             "Case No. in the Arbitration Court",
             "Debt period",
             "Principal debt, USD incl.VAT",
             "Interests, USD",
             "fee, USD",
             "Penalty by court decision",
             "Total repayment amount, USD",
             "Maturity, not later"
         ],
         "rows":
         [
             [
                 "1",
                 "A12-1234/2030",
                 "Октябрь 2017",
                 "12 345,58",
                 null,
                 "23 456,00",
                 "78 912,41",
                 null,
                 "31.08.2019"
             ]
         ]
     },
     "TABLE-FORMATTED":
     {
         "rows":
         [
             {
                 "No": "1",
                 "agreement_number": "Example number 1",
                 "district": "Example district 1",
                 "enterprise": "Example enterprise 1",
                 "disconnect_date": "Example date 1",
                 "address": "Example address 1",
                 "complex_field": "Example of sum 1"
             }
         ]
     }
 }
```

List descriptors

For XML format

The descriptor element lists contains a list of child list elements with the name attribute. The name attribute matches the tag in the template. The tag in the template will be replaced by a paragraph formatted as a list with element data from the list descriptor in the request.

Example:

```
<input-data>
    <lists>
       <list name="BULLET_LIST">
           <items>
               <item>bullet item 1</item>
               <item>bullet item 2</item>
               <item>bullet item 3</item>
           </items>
       </list>
        <list name="NUMBERED_LIST">
           <items>
               <item>numbered item 1</item>
               <item>numbered item 2</item>
               <item>numbered item 3</item>
           </items>
       </list>
   </lists>
</input-data>
```

For JSON format

List descriptors are arrays within the input-data object. The field names of the list objects match the tags in the template.

Example:

```
"input-data":
   {
       "BULLET_LIST": ["bullet item 1", "bullet item 2", "bullet item 3"],
       "NUMBERED_LIST": ["numbered item 1", "numbered item 2", "numbered item 3"]
   }
```

Data to be substituted into image tags

For XML format

The `images` element contains a list of child `image` elements with the `name` attribute. The `name` attribute matches the tag in the template. The images in the template will be replaced by the image data from the request.

Example:

```
<input-data>
    <images>
       <image name="IMAGE-1-JPEG">/9j/4AAQSkZJRgABAQEAYABgAAD/4Q…CigAooAKKACigD//Z</
↪image>
       <image name="IMAGE-2-JPEG">/9j/4AAQSkZJRgABAQEAYABg…KKACigAooAKKACigAooA//9k=</
↪image>
   </images>
</input-data>
```

For JSON format

The data to be substituted into the `image` tags is inside the images object, which is a child object in relation to the `input-data` object.

Example:

```
"input-data":
  {
      "images":
      {
          "IMAGE-1-JPEG":
          {
              "data": "/9j/4AAQSkZJRgABAQEAYABgAAD…igAooAKKACigAooAKKACigD//Z"
          },
          "IMAGE-2-JPEG":
          {
              "data": "/9j/4AAQSkZJRgABAQEAYAB…AooAKKACigAooAKKACigAooA//9k="
          }
      }
  }
```

Block descriptors

For XML format

The `blocks` descriptor element contains a list of `block` child elements with the block- template attribute (matches the name of the block template in the template document). Block has the same elements as `input- data`, except for image and block elements. The block templates in the template document will be used to insert any number of instances of blocks with different data sets into the document. Insertion locations are defined by the tags of the template instance in the template document. Details can be found in the subsection “Working with document → Blocks”.

For JSON format

The list (array) of `block` descriptors blocks is inside the `input-data` object.

Example:

```
"input-data": {
    "blocks": [
        {
            "block-template": "block1",
            "data": {
                "таблица1": {
                    "rows": [{
                            "days": "670",
                            "share": "130",
                            "penalty: "12 564,46",
                            "date_from": "11.02.2021",
                            "period": "Sales document (act, invoice) 0000165 от 31.01.
↪2021 23:59:59",
                            "rate": "7,5",
                            "date_until": "12.12.2022",
                            "formula": "32 505,07 * 670 * 0.08 * 1/130",
                            "debt": "32 505,07"
                        }
                    ]
                },
```

(continues on next page)

(continued from previous page)

```
                "amount_penalty": "12 345,67",
                "amount_debt": "78 910,23",
                "contract_number": "1"
            }
        },
        {
            "block-template": "block2",
            "data": {
                "таблица1": {
                    "rows": [{
                            "days": "670",
                            "penalty: "12 564,46",
                            "date_from": "11.02.2021",
                            "period": "Sales document (act, invoice) 0000165 от 31.01.
↪2021 23:59:59",
                            "rate": "7,5",
                            "date_until": "12.12.2022",
                            "debt": "32 505,07"
                        }
                    ]
                },
                "amount_penalty": "76 543,21",
                "amount_debt": "987 654,32",
                "contract_number": "2"
            }
        }
    ]
}
```

Request options

The request element (object) `options` contains:

- `enable-debug-report-save` - Boolean value which specifies whether to create a copy of the document report and a copy of the request (file name - template identifier) in the local `reports_debug` directory of the service. The default value is `false`.
- `enable-binary-output` - Boolean value which indicates that the service will output the result as binary data, without base64 encoding. In case of an error, the response will correspond to *used by default*. The default value is `false`.
- `formatting` - formatting options object which is only supported for word queries. Object fields: `tables` - object of formatting options for tables. Object fields: `enable-cells-auto-merge` - Boolean value which specifies whether to merge consecutive cells with the same value in one column vertically. The default value is `true`.
- `report-format` - string which indicates the format of the report file if it is different from the format mentioned in the request url. The only non-empty supported value: "`pdf`". Default value: empty string. See "Convert report to PDF" below.

Example for XML format:

```
<options>
    <enable-debug-report-save>true</enable-debug-report-save>
    <enable-binary-output>false</enable-binary-output>
    <formatting>
```

(continues on next page)

(continued from previous page)

```
        <tables>
            <enable-cells-auto-merge>true</enable-cells-auto-merge>
        </tables>
    </formatting>
    <report-format>pdf</report-format>
</options>
```

Example for JSON format:

```
"options":
  {
      "enable-debug-report-save": true,
      "enable-binary-output": false,
      "formatting": {
          "tables": {
              "enable-cells-auto-merge": true
          }
      },
      "report-format": "pdf"
  }
```

Response format

The `response-format` element (object) can take `json` and `xml` values. One can read more about how the response format is set in the "Response Structure" subsection.

Example for XML format:

```
<response-format value="xml"/>
```

Example for JSON format:

```
{"response-format": "json"}
```

Examples of requests

Examples for XML format.

Example #1

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
    <template uri="…" id="…"/>
    <input-data>
        <tags>…</tags>
        <tables>…</tables>
        <lists>…</lists>
        <images>…</images>
        <blocks>…</blocks>
    </input-data>
    <options>…</options>
    <response-format value="…"/>
</request>>
```

Example #2

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
   <template uri="local" id="template_example_1"/>
   <input-data>
       <tags>
           <tag name="ORGANIZATION">AO «xxxxyyyy»</tag>
           <tag name="TAG_IN_HEADER_TEST">Example of a header</tag>
           <tag name="TAG_IN_FOOTER_TEST">Example of a footer</tag>
           <tag name="CONDITIONAL_TAG_TRUE">true</tag>
           <tag name="CONDITIONAL_TAG_FALSE">false</tag>
       </tags>
       <tables>
           <table name="TABLE-FORMATTED">
               <rows>
                   <row>
                       <cell tag="No">1</cell>
                       <cell tag="contract_number">Example number 1</cell>
                       <cell tag="district">Example district 1</cell>
                       <cell tag="enterprise">Example enterprise 1</cell>
                       <cell tag="shutdown_date">Example date 1</cell>
                       <cell tag="address">Example address 1</cell>
                       <cell tag="complex_field">Example of sum 1</cell>
                   </row>
               </rows>
           </table>
           <table name="TABLE-NO-FORMAT">
               <header>
                   <cell>No</cell>
                   <cell>Case No. in the Commercial Court</cell>,
                   <cell>Debt period</cell>,
                   <cell>Principal debt, USD incl.VAT</cell>,
                   <cell>Interests, USD</cell>,
                   <cell>Fee, USD</cell>,
                   <cell>Penalty by court decision</cell>,
                   <cell>Total repayment amount, USD</cell>,
                   <cell>Maturity, not later</cell>
               </header>
               <rows>
                   <row>
                       <cell>1</cell>,
                       <cell>A12-1234/2030</cell>,
                       <cell>October 2017</cell>,
                       <cell>12 345,58</cell>,
                       <cell/>
                       <cell>23 456,00</cell>,
                       <cell>78 912,41</cell>,
                       <cell/>
                       <cell>31.08.2019</cell>
                   </row>
               </rows>
           </table>
       </tables>
       <images>
           <image name="IMAGE-1-JPEG">/9j/4AAQSkZJRgABAQEAYABgAAD…//Z</image>
           <image name="IMAGE-2-JPEG">/9j/4AAQSkZJRgABAQEAYAB…//9k=</image>
       </images>
```

(continues on next page)

(continued from previous page)

```
    </input-data>
    <options>
        <enable-debug-report-save>true</enable-debug-report-save>
        <formatting>
            <tables>
                <enable-cells-auto-merge>true</enable-cells-auto-merge>
            </tables>
        </formatting>
        <!-- <report-format>pdf</report-format> -->
    </options>
</request>
```

Example for JSON format

```
{
    "template":
    {
        "uri": "local",
        "id": "template_example_1"
    },
    "input-data":
    {
        "ORGANIZATION": "JSC «xxxxyyyy»",
        "TAG_IN_HEADER_TEST": "Example header",
        "TAG_IN_FOOTER_TEST": "Example footer",
        "CONDITIONAL_TAG_TRUE": "true",
        "CONDITIONAL_TAG_FALSE": "false",
        "TABLE-NO-FORMAT":
        {
            "header":
            [
                "No",
                "No. of Arbitration Court Case",
                "Debt Period",
                "Principal Debt, USD, including VAT",
                "Interest, USD",
                "Fee, USD",
                "Penalty under court decision",
                "Total repayment amount, USD",
                "Maturity, not later"
            ],
            "rows":
            [
                [
                    "1",
                    "A12-1234/2030",
                    "October 2017",
                    "12 345,58",
                    null,
                    "23 456,00",
                    "78 912,41",
                    null,
                    "31.08.2019"
                ]
            ]
        },
        "TABLE-FORMATTED":
```

(continues on next page)

(continued from previous page)

```
        {
            "rows":
            [
                {
                    "No": "1",
                    "contract_number": "Example number 1",
                    "district ": "Example district 1",
                    "company": "Example company 1",
                    "cutoff_date": "Example date 1",
                    "address": "Example address 1",
                    "xcomplex_field": "Example amount 1"
                }
            ]
        },
        "images":
        {
            "IMAGE-1-JPEG":
            {
                "data": "/9j/4AAQSkZJRgABAQEAYABgAAD…//Z"
            },
            "IMAGE-2-JPEG":
            {
                "data": "/9j/4AAQSkZJRgABAQEAYAB…//9k="
            }
        }
    },
    "options":
    {
        "enable-debug-report-save": true,
        "formatting": {
            "tables": {
                "enable-cells-auto-merge": true
            }
        }
    }
}
```

Examples of XML and JSON queries can also be viewed in the *examples archive*.

Service call example

For XML format

```
curl --request POST --data-binary "@templates/examples/docx/example_from_xml.xml"␣
↪http://localhost:8087/word_report_xml
```

For JSON format

```
curl --request POST --data-binary "@templates/examples/docx/example_from_json.json"␣
↪http://localhost:8087/word_report_json
```

Response structure

The response format (JSON or XML) can be specified in two ways:

1. in the HTTP header Accept. Supported values: `application/json` and `application/xml`.
2. in the request body in the `response-format` element (object). Supported values: `json` and `xml`. Priority is given to the format specified in the request body.

The service response contains an object in JSON or XML format that includes:

1. Error description (error code, error message). In case of successful service response, the value `null` is returned.
2. Result (base64 encoded file of the report document in docx/pdf format).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded docx/pdf file]"
}
```

Example of response

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Working with the document

General information

For documents in DOCX format, the following are available:

- tags that will be replaced by the input data from the request
- tables
- lists
- images
- conditional expressions
- creating a single report from a single template based on multiple input data (multi-reports)
- blocks.

Working with tables

The following table types are supported in the template:

1. Tables without formatting are set only by tag. The table is generated to fit the width of the page with equal width of columns.
2. Tables with formatting are specified by a table with the following parameters:

- optional custom header
- The first cell of the row following the header contains a table tag. This row is deleted during generation.
- The line following the tag line contains column tags, their formatting will be applied to the generated rows. This line is deleted during generation.
- optional fixed strings with any tags (replacement data source - main tags).

Note: tables in document footers are supported.

Cell merge management

The basic merging algorithm is to automatically merge cells with the same content vertically.

One can change the merging algorithm by using the `formatting.tables.enable-cells- auto-merge request option`.

Advanced vertical and horizontal merging options are supported for JSON-formatted requests. An object describing a table row has an optional `formatting object`.

Example:

```
"formatting": {
   "column tag value": {
       "vertical_merge": "restart",
       "column_span": 2
   }
}
```

The field names of this object are the names of the table columns for which you want to apply the merging option. Field values: merge option object. This object has the fields:

- `vertical_merge` - a string which controls the vertical merging of a cell. Supported values: “restart” - prevents merging with a neighboring cell located in the previous row of the same column; “continue” - force merging with a cell higher in the column; “unset” - merging option will be inherited from the cell higher up the column;
- `column_span` - integer. Controls the horizontal merging of a cell. The value specifies the number of cells to merge to the right of the current cell (inclusive).

Note: this is a direct control of the merging settings in the document. To quickly understand what values should be set to which cells, one should first configure the desired behavior in a text editor, save the docx document, unzip it as a zip archive and see what values are applied in the document.xml file.

Lists

Single-level token-type and numbered lists are supported.

Example for XML format

```
<lists>
<list name="BULLET_LIST">
 <items>
  <item>bullet item 1</item>
  <item>bullet item 2</item>
  <item>bullet item 3</item>
 </items>
</list>
</lists>
```

A detailed example of a request using lists (lists.json) and its template (lists.docx) can be found in the *examples archive*.

Adding items is possible in any place of the list in the template. In case of an empty list in the input data, the list is deleted from the report

Images

See "Working with images" in the "Working with template documents" subsection.

Conditional expressions

Parts of the template text can be excluded from the report depending on the truth of the condition in the conditional tag. Format of the conditional tag:

```
[#condition]conditional template text[/condition]
```

The opening tag begins with a # character followed by a condition. The closing tag starts with a / followed by the condition of the opening tag.

Note: Conditional expressions in footers are not supported.

In a request, the data to be substituted into conditional expression tags is inside the `input- data` element (object).

Example for XML format:

```
<input-data>
       <tags>
           <tag name="CONDITIONAL_TAG_TRUE">true</tag>
           <tag name="CONDITIONAL_TAG_FALSE">false</tag>
       </tags>
</input-data>
```

Example for JSON format:

```
"input-data":
 {
       "ORGANIZATION": "JSC «xxxxyyyy»",
       "TAG_IN_HEADER_TEST": "Example header",
```

(continues on next page)

(continued from previous page)

```
        "TAG_IN_FOOTER_TEST": "Example header",
        "TAG_IN_FOOTER_TEST": "Example footer",
        "CONDITIONAL_TAG_TRUE": "true",
        "CONDITIONAL_TAG_FALSE": "false"
}
```

A detailed example of a request using conditional expressions (example_from_json.json) and its template (example_from_json.docx) can be found in the *examples archive*.

Blocks

A block is a report fragment containing text/tables/lists (everything except images in the current version). It is created from a template-block (a specific fragment of a template document) any number of times at any needed location in the template with different sets of input data.

A block template in a template document is defined by tags `[block-template:arbitrary block template name]`. Block templates are completely removed from the template document before replacing tags with input data.

Block insertion locations are defined by the tags of certain block templates [block-instances: list of block template names separated by comma] of the template document.

Block template content is inserted into the document (with tags replaced in the document) at the location of the tag of the certain template.

Each block template instance tag specifies which templates should be used as the basis for creating a block in the report (in other words, instantiate a block template). All block inputs are reviewed in order. When replacing tags in the block template, all tag values are taken only from the block data object (relevant for the current version of the report service).

In the request, `block` data is defined by the `blocks` array, which contains block objects with `block-template` fields (matches the name of the block template in the template document) and the data object field. The data object has the same child objects as the `input-data` object, except for images and blocks.

Example:

```
"blocks":
 [
    {
        "block-template": "block1",
        "data": {
            "table1": {
                "rows": [{
                    }
                ]
            },
            "amount_penalty": "12 345,67",
            "amount_debt": "78 910,23",
            "contract_number": "1"
        }
    },
    {
        "block-template": "block2",
        "data": {
            "table1": {
                "rows": [{
                    }
                ]
```

(continues on next page)

(continued from previous page)

```
            },
            "amount_penalty": "76 543,21",
            "amount_debt": "987 654,32",
            "contract_numbe": "2"
        }
    },
    {
        "block-template": "block3",
        "data": {
            "contract_number": "1",
            "contract_title": "Contract Title 1",
            "date_from": "01/01/2000",
            "date_until": "01/01/2100"
        }
    },
    {
        "block-template": "block3",
        "data": {
            "contract_number": "2",
            "contract_title": "Contract Title 2",
            "date_from": "02.02.2020",
            "date_until": "02.02.2080"
        }
    }
]
```

A detailed example of a request using blocks (example_multiblocks.json) and its template (example_multiblocks.docx) can be found in the *examples archive*.

Working with multi-reports

General information

It is possible to create a single report from a single template based on multiple inputs.

Note:

- The current version does not support numbered lists.
- The current version does not support creating a single report from multiple templates.
- Tags in footers should be used wisely, as there is no way to have custom footers for each report.
- Image replacement is only supported globally. All reports will have the set of images that were used last. This is a limitation of the current version of the service.

Request examples

Example for XML format

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
   <template uri="local" id="multi_report_1-N"/>
   <input-data-array>
       <input-data>
           <tags>
```

(continues on next page)

(continued from previous page)

```
                <tag name=="subscriber">subscriber1</tag>
                <tag name="address_subscriber">address_subscriber1</tag>
                <tag name="period">period1</tag>
                <tag name="treaties">treaties1</tag>
                <tag name="total_amount_debt">total_amount_debt1</tag>
                <tag name="amount_indebtedness1">sum_indebtedness1_1</tag>
                <tag name="amount_indebtedness2">amount_indebtedness2_1</tag>
                <tag name="date_penny">date_penny1</tag>
            </tags>
            <tables>
                <table name="TABLE-FORMATTED">
                    <rows>
                        <row>
                            <cell tag="No">1</cell>
                            <cell tag="contract_number">Example number 1</cell>
                            <cell tag="district">Example district 1</cell>
                            <cell tag="enterprise">Example enterprise 1</cell>
                            <cell tag="date_off">Example date 1</cell>
                            <cell tag="address">Address Example 1</cell>
                            <cell tag="complex_field">Sum Example 1</cell>
                        </row>
                    </rows>
                </table>
            </tables>
        </input-data>
        <input-data>
            <tags>
                <tag name="subscriber">subscriber2</tag>
                <tag name="address_subscriber">address_subscriber2</tag>
                <tag name="period">period2</tag>
                <tag name="treaties">treaties2</tag>
                <tag name="total_amount_debt">total_amount_debt2</tag>
                <tag name="amount_debt1">sum_debt1_2</tag>
                <tag name="amount_debt2">sum_debt2_2</tag>
            </tags>
            <tables>
                <table name="TABLE-FORMATTED">
                    <rows>
                        <row>
                            <cell tag="No">1</cell>
                            <cell tag="contract_number">Example number 3</cell>
                            <cell tag="district">District 3 example</cell>
                            <cell tag="enterprise">Example Enterprise 3</cell>
                            <cell tag="date_off">Example date 3</cell>
                            <cell tag="address">Address Example 3</cell>
                            <cell tag="complex_field">Sum Example 3</cell>
                        </row>
                    </rows>
                </table>
            </tables>
        </input-data>
    </input-data-array>
    <options>
        <enable-debug-report-save>true</enable-debug-report-save>
        <formatting>
            <tables>
                <enable-cells-auto-merge>true</enable-cells-auto-merge>
```

(continues on next page)

(continued from previous page)

```
            </tables>
        </formatting>
        <!-- <report-format>pdf</report-format> -->
    </options>
</request>
```

Example for JSON format

```
{
   "template":
   {
       "uri": "local",
       "id": "template_example_1"
   },
   "input-data-array":
   [
       {
           "ORGANIZATION": "JSC «xxxxyyyy1»",
           "CLINAME": "MUNICIPAL ENTERPRISE OF  \"XXXXXXXXXX\"",
           "CTRNUMBER": "1234",
           "TABLE-NO-FORMAT":
           {
               "header":
               [
                   "No",
                   "Arbitration court case number1",
                   "Debt period1",
                   "Principal debt, USD with VAT1",
                   "Interest, USD1", "Fee, USD1",
                   "Court judgment penalty1",
                   "Total repayment amount, USD1",
                   "Maturity, not later1"
               ],
               "rows":
               [
                   [
                       "1_1",
                       "A12-1234/2030_1",
                       "October 2017_1",
                       "12 345,58_1",
                       null,
                       "23 456,00_1",
                       "78 912,41_1",
                       null,
                       "31.08.2019_1"
                   ]
               ]
           },
           "TABLE-FORMATTED":
           {
               "rows":
               [
                   {
                       "No": "1_1",
                       "contract_number": "Example number 1_1",
                       "district": "Example district 1_1",
                       "company": "Example company 1_1",
```

(continues on next page)

(continued from previous page)

```
                "cutoff_date": "Example date 1_1",
                "address": "Example address 1_1",
                "complex_field": "Example amount 1_1"
            }
        ]
    },
    "images":
    {
        "IMAGE-1-JPEG":
        {
            "data": "/9j/4AAQSkZJRgABAQEAYABgAAD…AKKACigD//Z"
        },
        "IMAGE-2-JPEG":
        {
            "data": "/9j/4AAQSkZJRgABAQEAYABg…oAKKACigAooA//9k="
        }
    }
},
{
    "ORGANIZATION": "JSC «xxxxyyyy_2»",
    "CLINAME": "MUNICIPAL ENTERPRISE OF THE CITY\"PASSENGER AUTOMOBILE
    TRANSPORTATION_2\"",
    "CTRNUMBER": "1234_2",
    "RESTPENY": null,
    "TAG_IN_HEADER_TEST": "Example header_2",
    "TAG_IN_FOOTER_TEST": "Example footer_2",
    "CONDITIONAL_TAG_TRUE": "true",
    "CONDITIONAL_TAG_FALSE": "false",
    "TABLE-NO-FORMAT":
    {
        "header":
        [
            "No_2",
            "Case No. in Arbitration Court_2",
            "Debt Period_2",
            "Principal Debt, USD with VAT_2",
            "Interest, USD_2", "Fee, USD_2",
            "Court Order Penalty_2",
            "Total repayment amount, USD_2",
            "Maturity, not later than_2"
        ],
        "rows":
        [
            [
                "1_2",
                "A12-1234/2030_2",
                "October 2017_2",
                "12 345,58_2",
                null,
                "23 456,00_2",
                "78 912,41_2",
                null,
                "31.08.2019_2"
            ]
        ]
    },
    "TABLE-FORMATTED":
```

(continues on next page)

(continued from previous page)

```
                {
                    "rows":
                    [
                        {
                            "No": "1_2",
                            "contract_number": "Example number 1_2",
                            "district": "Example district 1_2",
                            "company": "Example company 1_2",
                            "cutoff_date": "Example date 1_2",
                            "address": "Example address 1_2",
                            "complex_field": "Example amount 1_2".
                        }
                    ]
                }
            }
        ],
        "options":
        {
            "enable-debug-report-save": true,
            "formatting":
            {
                "tables":
                {
                    "enable-cells-auto-merge": true
                }
            }
        }
}
```

A detailed example of a request for generating a multi-report (multi_report_1-N_json.json) and its template (multi_report_1-N_json.docx) can be found in the *examples archive*.

Service call examples

For XML format

```
curl --request POST --data-binary "@templates/examples/docx/multi_report_1-N_xml.xml"␣
↪http://localhost:8087/word_multi_report_1_N_xml
```

For JSON format

```
curl --request POST --data-binary "@templates/examples/docx/multi_report_1-N_json.json
↪" http://localhost:8087/word_multi_report_1_N_json
```

7.1.4 Report generation in XLSX format

Service description

Name of service	Service for generating multi-reports from a template document in the **XLSX** format
Path to service	• [host]:[port]/excel_report_json • [host]:[port]/excel_report_xml • [host]:[port]/excel_report_json/v2
Method	POST
Parameters	The request body must contain a JSON or XML object. Read more about the structure of the request body in the subsection "Request body structure". In response, the service returns a document file in XLSX format encoded in BASE64. When converting the report to PDF format, the service returns a BASE64 encoded PDF file.
Purpose	The service is designed to generate multi-reports from a template document in **XLSX** format. Conversion to PDF is possible.

Request body structure

The body of the request contains an object in JSON or XML format that includes:

1. Template descriptor.
2. Input data to substitute into the template instead of tags.
3. Request options.
4. Response format.

Template descriptor

The element (object) of the `template` descriptor contains:

- `uri` - string which specifies the location of the template document file depending on how the `id` parameter is interpreted. Supported value: `local`.
- `id` - string. Template identifier. It specifies the path to the template document file relative to the `templates` service directory. It is also used to write a report file in debug mode and to identify a request in the log.

Example for XML format:

```xml
<template uri="local" id="Newsletter"/>
```

Example for JSON format:

```json
      "template":
{
    "uri": "local",
    "id": "template_example_1"
}
```

Input-data

The input data element (object) `input-data` contains:

- Data to substitute into the template instead of tags: strings, numbers, formulas, links.
- Table descriptors.
- Block descriptors.
- Data to be substituted into image tags.

Note: The difference between the data structure for JSON and XML is that instead of XML elements, data is represented as objects and lists of objects. Tag names are specified by object field names. Tables and images are child objects with respect to the `input-data` object.

Data to substitute into the template instead of tags

Rows

For XML format

The `tags` element contains a list of child tag elements with the name attribute and the desired specific value with which the `tag` in the template will be replaced when generating the report. The `name` attribute matches the tag in the template.

Example:

```
<input-data>
    <tags>
        <tag name="ORGANIZATION">JSC «xxxxyyyy»</tag>
    </tags>
</input-data>
```

For JSON format

Tag names are given by the names of the objects within the `input-data`.

Example:

```
"input-data":
   {
       "ORGANIZATION": "JSC «xxxxyyyy»"
}
```

Names

For XML format:

The tag element also supports the type attribute. If the type attribute is “num”, the tag value will be interpreted as a floating point number. In this case, a numeric value will be substituted in the template, which allows, for example, the application of formulas.

Example:

```
<tag name="doc_cnt" type="num">2</tag>}
```

For JSON format

To interpret the tag value as a number, one must specify the value without quotation marks, with a period as a separator of the integer and fractional parts.

Example:

```
"input-data":
{
        "total": 3889.98
}
```

Formulas

Instead of a tag, a formula can be inserted into the template to calculate and display values.

The formula in the request is specified by an object with fields: type - string, supported value: "formula". value - string, the values are taken from the used table editor, without the equal sign.

For XML format type is an attribute of a tag element, and value is its value:

```
<tag name="sum" type="formula">SUM(AX2:AX3)</tag>
```

Для формата JSON:

```
"sum":  {
              "type": "formula",
                "value": "SUM(AX2:AX3)"
            }
```

No row number conversions are performed. It is assumed that the row numbers are relevant for the final report, i.e. after inserting the table rows preceding the row containing the formula. The row numbers can be calculated based on the table row numbers in the template document and the number of table rows at the time of generating the formula row in the request.

Example request: formula.json It can also be found in the *examples archive*.

Links

Instead of a tag, a hyperlink can be inserted into the template.

The link in the request is specified by an object with fields:

- `type` - string, supported value: "link".
- `text` - cstring, the link name displayed in the document.
- `value`- string, the value to follow the link.

For the XML format, type and text are attributes of the tag element, and value is its value:

```
<tag name="http://link1" type="link" text="company XSQUARE">"http://xsquare.ru"</tag>
```

For JSON format:

```
"http://link1": {
                        "type": "link",
                        "text": "company XSQUARE",
                        "value": "http://xsquare.ru"
                    }
```

Note: Tags for links should be hyperlinks in the document template. To do this, one can start the tag name with http:// (or https://, ftp://))

Example request: hyperlinks.json. It can also be found in the *examples archive*.

Table descriptors

Table generation based on data from the request is also available for XLSX reports.

Differences from DOCX format:

- tags are set in the `cell-tags` element (object) once for the whole table;
- cell tag maps the column tag to the cell index in the array specifying the table row.

For XML format

The `tables` descriptor element contains a list of child `table` elements with the `name` attribute. The `name` attribute matches the tag in the template. The tag in the template will be replaced by the table created according to the table description in the request.

Example:

```
<input-data>
     <tables>
        <table name="table_insured_individuals">
            <cell-tags>
                <cell-tag name="No" index="0"/>
                <cell-tag name="individual_personal_insurance_number" index="1"/>
                <cell-tag name="sex" index="2"/>
                <cell-tag name="name" index="3"/>
                <cell-tag name="birth_date"  index="4"/>
                <cell-tag name="contract_date"  index="5"/>
                <cell-tag name="contract_number"  index="6"/>
            </cell-tags>
            <rows>
                <row>
                    <cell>1</cell>
                    <cell>001-002-003 00</cell>
                    <cell>F</cell>
                    <cell>Kate_Smith</cell>
                    <cell>01.01.1970</cell>
                    <cell>01.01.1988</cell>
                    <cell>ABC-001-002-003-00</cell>
                </row>
            </rows>
        </table>
```

(continues on next page)

(continued from previous page)

```
    </tables>
</input-data>
```

For JSON format

Table descriptors are child objects with respect to the `input-data` object.

Example:

```
"input-data":
   {
        "таблица_застрахованные_лица":
       {
           "cell-tags": {
               "No": 0,
               "individual_personal_insurance_number": 1,
               "sex": 2,
               "name": 3,
               "birth_date": 4,
               "contract_date": 5,
               "contract_number": 6
           },
           "rows":
           [
               [
                   "1",
                   "001-002-003 00",
                   "F",
                   "Kate Smith",
                   "01.01.1970",
                   "01.01.1988",
                   "ABC-001-002-003-00"
               ]
           ]
       }
   }
```

Cell values in a table can also be formulas and links:

Example JSON:

```
"input-data": {
   "таблица_1": {
     "rows": [
       [
         "1",
         "Heating Company",
         "Enterprise 1",
         161975.87,
         null,
         161975.87,
         12985.51,
         "15.08.2023",
         {
           "type": "link",
```

(continues on next page)

(continued from previous page)

```
          "text": "text":"example of a link in a table 1",
          "value": "http://xsquare.ru"
        }
      ],
      [
        null,
        null,
        "TOTAL FEDERAL BUDGET",
        {
          "type": "formula",
          "format": "raw",
          "value": "SUM(D11:D11)"
        },
        null,
        161975.87,
        12985.51,
        null,
        null
      ]
    ],
    "cell-tags": {
      "name": 1,
      "No": 0,
      "Link": 8,
      "due_date": 7,
      "debt_total": 3,
      "debt_current":  4,
      "company_name": 2,
      "overdue_debt": 5,
      "outstanding_balance_of_overdue_debt": 6
    }
  }
```

Example XML:

```
<tables>
  <table name="table_insured_Individuals">
    <cell-tags>
      <cell-tag name="No"index="0"/>
      <cell-tag name="individual_personal_insurance_number"index="1" />
      <cell-tag name="http://link" index="2" />
    </cell-tags>
    <rows>
      <row>
        <cell>1</cell>
        <cell>001-002-003 00</cell>
        <cell type="link" text="link text1">http://table1.ru</cell>
      </row>
      <row>
        <cell>2</cell>
        <cell>001-002-004 00</cell>
        <cell type="link" text="link text2">http://table2.ru</cell>
      </row>
    </rows>
  </table>
</tables>
```

Example request: `example_with_links.xml`. It can also be found in the *examples archive*.

Block descriptors

Generation of data blocks in a document based on a template block and data from the request is available for XLSX reports.

A block template means the sequence of rows in a document between the cells with tags [block_name] - [/block_name].

A block-template can be repeated many times consecutively in the document. One can also optionally add a page break after each block, which is useful when printing documents.

The `blocks` descriptor element contains a list of `block` child elements with the attribute `template` (matches the name of the block template in the template document).

A block has the same elements as `input-data`, except for image and block elements. The data for the block is located in the data descriptor, and a page break at the end of the block can be set for each block.

The page break is set by specifying `true` in the page-break descriptor.

Block templates in a template document will be used to insert any number of instances of blocks with different data sets. Blocks are inserted sequentially one after another.

Example XML:

```xml
</blocks>
    <block template="block_1">
        <data>
            <tags>
                <tag name="date">01.02.2024</tag>
            </tags>
        </data>
    </block>
    <block template="block_1">
        <page-break>true</page-break>
        <data>
            <tags>
                <tag name="date">01.02.2025</tag>
            </tags>
        </data>
    </block>
    <block template="block_1">
        <page-break>true</page-break>
        <data>
            <tags>
                <tag name="date">01.02.2026</tag>
            </tags>
        </data>
    </block>
</blocks>
```

For requests in JSON format, the name of the block template is specified in the `block-template` descriptor.

JSON:

```json
"blocks": [
  {
    "block-template": "block_1",
    "page-break": true,
```

(continues on next page)

(continued from previous page)

```
    "data": {
      "main2": "main_for_1",
      "main3": "m1_val",
      "main4": "m1_val2",
      "table": {
        "cell-tags": {
          "f2": 0,
          "f3": 1,
          "dt_pay": 2
        },
        "rows": [
          [
            "f2_val1",
            "f3_val1",
            "22.12.2023"
          ],
          [
            "f2_val2",
            "f3_val2",
            "23.12.2023"
          ]
        ]
      }
    }
  },
  {
    "block-template": "block_1",
    "data": {
      "main2": "main_for_2",
      "main3": "m2_val",
      "main4": "m2_val2",
      "table": {
        "cell-tags": {
          "f2": 0,
          "f3": 1,
          "dt_pay": 2
        },
        "rows": [
          [
            "f2_val3",
            "f3_val3",
            "22.11.2022"
          ]
        ]
      }
    }
  }
]
```

Example request: blocks.json. It can also be found in the *examples archive*.

Note: rows with empty cells are not allowed in the block-template. In order to make the cells not empty, it is enough to fill cells of the block with any color, and then cancel the filling and save the template file.

Data to be substituted into image tags

For XML format

The `images` element contains a list of child `image` elements with the `name` attribute. The `name` attribute matches the tag in the template. The images in the template will be replaced by the image data from the request. The new contents of the image file encoded in BASE64 are passed in the request.

Example:

```
<input-data>
    <images>
       <image name="IMAGE-1-JPEG">/9j/4AAQSkZJRgABAQEAYABgAAD/4Q…CigAooAKKACigD//Z</
↪image>
       <image name="IMAGE-2-JPEG">/9j/4AAQSkZJRgABAQEAYABg…KKACigAooAKKACigAooA//9k=</
↪image>
   </images>
</input-data>
```

For JSON format

The data to be substituted into the image tags is inside the child objects of the `input-data` object.

Example:

```
"input-data":
  {
      "images":
      {
          "IMAGE-1-JPEG":
          {
              "data": "/9j/4AAQSkZJRgABAQEAYABgAAD…igAooAKKACigAooAKKACigD//Z"
          },
          "IMAGE-2-JPEG":
          {
              "data": "/9j/4AAQSkZJRgABAQEAYAB…AooAKKACigAooAKKACigAooA//9k="
          }
      }
  }
```

Request options

The request element (object) `options` contains:

- `enable-debug-report-save` - Boolean value specifies whether to create a copy of the document report and a copy of the request (file name - template identifier) in the local `reports_debug` directory of the service. The default value is `false`.
- `enable-binary-output` - Boolean value indicates that the service will output the result as binary data, without base64 encoding. In case of an error, the response will correspond to *used by default*. The default value is - `false`.
- `report-format` - string indicates the format of the report file if it is different from the format mentioned in the request url. The only non-empty supported value: “`pdf`”. Default value: empty string. See “Convert report to PDF” below.

Note: in the document template the print areas ("Set Print Area") must be set, according to them the conversion to PDF will be performed

Example for XML format:

"options":

```
{
    <enable-debug-report-save>true</enable-debug-report-save>
    <enable-binary-output>false</enable-binary-output>
    <report-format>pdf</report-format>
</options>
```

Example for JSON format:

```
<options>
    "enable-debug-report-save": true,
    "enable-binary-output": false,
    "report-format": "pdf"
}
```

Response format

The `response-format` element (object) can take json and xml values. One can read more about how the response format is set in the "Response structure" subsection.

Example for XML format

```
<response-format value="xml"/>
```

Example for JSON format:

```
{"response-format": "json"}
```

Examples of requests

Example for XML format

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
   <response-format value="xml"/>
   <template uri="local" id="template_example"/>
   <input-data>
       <tags>
           <tag name="organization">State Institution-Pension Fund</tag>
           <tag name="number">25</tag>
           <tag name="month">December</tag>
           <tag name="year">2018</tag>
           <tag name="document_number">01</tag>
           <tag name="number_contracts">3</tag>
           <tag name="employee">Mary Jane</tag>
           <tag name="tel_staff">+555(123) 456-78-90</tag>
           <tag name="tag1">tag1_value</tag>
```

(continues on next page)

(continued from previous page)

```
            <tag name="tag2">tag2_value</tag>
        </tags>
        <tables>
            <table name="table_insured_individuals">
                <cell-tags>
                    <cell-tag name="No" index="0"/>
                    <cell-tag name="personal insurance number" index="1"/>
                    <cell-tag name="sex" index="2"/>
                    <cell-tag name="name" index="3"/>
                    <cell-tag name="date_birth" index="4"/>
                    <cell-tag name="date_contract" index="5"/>
                    <cell-tag name="contract_number" index="6"/>
                </cell-tags>
                <rows>
                    <row>
                        <cell>1</cell>
                        <cell>001-002-003 00</cell>
                        <cell>F</cell>
                        <cell>Jane Smith</cell>
                        <cell>01.01.1970</cell>
                        <cell>01.01.1988</cell>
                        <cell>ABC-001-002-003-00</cell>
                    </row>
                </rows>
            </table>
        </tables>
        <images>
            <image name="IMAGE-1-JPEG">/9j/4AAQSkZJRgABAQEAYABgAAD/4QFQR0...
↪KACigAooAKKACigAooAKKACigD//Z</image>
            <image name="IMAGE-2-JPEG">/9j/4AAQSkZJRgABAQEAYABgAAD/4QFQR...
↪oAKKACigAooAKKACigAooA//9k=</image>
        </images>
    </input-data>
    <options>
        <enable-debug-report-save>true</enable-debug-report-save>
        <!-- <report-format>pdf</report-format> -->
    </options>
</request>
```

Example for JSON format

```
{
   "response-format": "json",
   "template":
   {
       "uri": "local",
       "id": "template_example"
   },
   "input-data":
   {
       "organization": "State Institution- Pension Fund,
       "number": "25",
       "month": "December",
       "year": "2018",
       "document_number": "01",
       "number_contracts": "3",
       "employee": "Jane Smith",
```

(continues on next page)

(continued from previous page)

```
        "employee_tel": "+555 (123) 456-78-90",
        "tag1": "tag1_value",
        "tag2": "tag2_value",
        "table_insured_individuals":
        {
            "cell-tags": {
                "No": 0,
                "personal_insurance_number": 1,
                "sex": 2,
                "name": 3,
                "date_birth": 4,
                "date_contract": 5,
                "contract_number": 6
            },
            "rows":
            [
                [
                    "1",
                    "001-002-003 00",
                    "F",
                    "Mary Jane", "01.01.1970",
                    "01.01.1970",
                    "01.01.1988",
                    "ABC-001-002-003-00"
                ]
            ]
        },
        "images":
        {
            "IMAGE-1-JPEG":
            {
                "data": "/9j/4AAQSkZJRgABAQEAYABgAAD/4QFQR0...
↪KACigAooAKKACigAooAKKACigD//Z"
            },
            "IMAGE-2-JPEG":
            {
                "data": "/9j/4AAQSkZJRgABAQEAYABgAAD/4QFQR...oAKKACigAooAKKACigAooA//
↪9k="
            }
        }
    },
    "options":
    {
        "enable-debug-report-save": true
    }
}
```

7.1.5 Service call example

For XML format

```
curl --request POST --data-binary "@templates/examples/xlsx/base_example_xml.xml"␣
↪http://localhost:8087/excel_report_xml
```

For JSON format

```
curl --request POST --data-binary "@templates/examples/xlsx/base_example.json" http://
↪localhost:8087/excel_report_json
```

7.1.6 Response structure

The response format (JSON or XML) can be specified in two ways:

1. in the HTTP Accept header. Supported values: `application/json` and `application/xml`.
2. in the request body in the `response-format` element (object). Supported values: `json` and `xml`.

Priority is given to the format specified in the request body.

The service response contains an object in JSON or XML format that includes:

1. Error description(error code, error message). In case of successful service response, the value `null` is returned.
2. Result (BASE64 encoded file of the report document in docx/pdf format).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded xlsx file]"
}
```

Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Error codes

The `code` field can take the following values:

“**1**” - error in the request;

“**2**” - report generation error.

7.1.7 Working with the document

General information

The following is available for documents in XLSX format:

- tags that will be replaced by the input data from the response
- tables
- blocks
- images.

Note:

- The input cell data format is text, numbers (integer or real numbers), formulas and hyperlinks.
- Cell references in formulas are not corrected during report generation. Formulas may become incorrect if rows are added/deleted to the report.

Working with tables

There are two modes for working with tables.

v1

API: `/excel_report_json`

A table with formatting is set as follows:

- the first cell of the row contains a tag with the table name. This row is deleted during generation.
- "tag row", the row(s) following the table tag row which contains the column tags, their formatting will be applied to the generated rows (if no formatting row is specified, see below). This row(s) is deleted during generation.
- optional: a row indicating that the next row contains a formatting row which contains one cell with the text "[cell format]". It is used for tables where numeric data should be in cells with number (or general) format, not rows (otherwise it is easier to set the formatting in the tag row).
- optional: "formatting row" specifying the format of cells instead of a tag row. Cell values are ignored. The format is applied to the generated rows. It should be a copy of the tag row as fa as the number of cells and their order is concerned.

The possibility to map a single string of input data to multiple rows in a document is supported. This may be necessary for tables that have vertically merged cells in one logical row. The number of template rows per input row is defined as the maximum number of vertically merged rows in any cell from the first row after the tagged row of the table itself (see function `getNumberOfSheetRowsPerInputDataRow`).

An example template can be found in *examples archive*.

Note: applying formatting to generated rows currently does not include font settings. Background color and cell border settings (e.g. horizontal merging of cells) are supported.

v2

API: `/excel_report_json/v2`

A formatted table is defined as follows:

- the first cell of the row contains a tag with the table name. This row is deleted during generation.
- "tag row" - the row following the tag row containing the column tags, their formatting will be applied to the generated rows (if no formatting row is set, see below). This row is deleted during generation.
- optional: a row indicating that the next row contains a formatting row. Contains one cell with the text "[cell format]". Used for tables where numeric data should be in cells with number (or general) format rather than rows (otherwise it is easier to set the formatting in the tag row).
- optional: "formatting row", a row specifying the format of cells instead of a tag row. Cell values are ignored. The format is applied to the generated row. It must be a complete copy of the tag row in terms of the number of cells and their order.

Example templates can be found in *examples archive*.

- `cell_format_options.xlsx`
- `cell_format_options_typed_cells.xlsx`

Note: currently, applying formatting to generated rows does not include font settings. Background color and cell border settings (e.g. horizontal merging of cells) are supported.

Cell merge control

The basic merging algorithm is to apply horizontal merging of generated cells based on merging of template table cells.

It supports advanced vertical merging options which is defined by an object with a single field:

- `vertical_span` - integer that controls the vertical merging of a cell. The value specifies the number of cells to merge down from the current cell (inclusive).

Options can be mapped to table columns in a template specified by index

```
"formatting": {
   "template-cells": [
       {
           "vertical_span": 2
       },
       ...
   ]
}
```

or to cell tags (this method has a priority, which is convenient for selective correction of merge options for parts of columns):

```
"formatting": {
   "tags": {
       "cell_tag_value": {
           "vertical_span": 2
       },
   }
}
```

7.1.8 Adding rows to the end of a template document

The rows specified by the `rows-to-add` list will be added to the end of the template without saving/applying formatting.

See the example request `add_rows_to_end.json` in the *examples archive*.

7.1.9 Images

See "Working with images" in the "Working with template documents" subsection.

7.1.10 Converting the report to PDF

Activation

One can specify the report format "pdf" in the request options.

Example:

```
<options>
    <report-format>pdf</report-format>
</options>
```

Dependencies

The Libre Office package must be installed. The `soffice` program should be available for calling. It is possible to set the path to it in the command line parameters of the report server or via a configuration file.

Operation principle

```
soffice --convert-to pdf report.docx --outdir temp_dir
soffice --convert-to pdf report.xlsx --outdir temp_dir
```

The report server saves the report to a file and calls the `soffice` utility with the following parameters, for example: then sends the resulting PDF file instead of the original report.

Default settings

The path to the `soffice` utility is searched in the directories specified by the `PATH` environment variable. One can specify a direct path in the report server command line arguments or through a configuration file.

File exchange with `soffice` goes through the temporary folder of the system. One can specify the working directory in the command line arguments of the report server or via a configuration file.

See the command output:

```
xreports --help
```

Performance

By default, the command execution time for a blank document - 2.3 seconds (for cpu i7-3615M, ssd). Most of the time is spent on Libre Office initialization.

```
soffice --convert-to pdf
```

To speed up the conversion, one can use the `quick start` mode, i.e. pre-execute the command.

```
soffice --quickstart
```

In this mode Libre Office is in RAM at all times, which saves about 2 seconds.

Note: for Libre Office 6.3.4.2 in Ubuntu 19.10 `soffice --quickstart` causes the main window to be displayed. The –`minimized` and –`headless` options do not solve this problem because they do not allow to use acceleration after a single conversion. In Windows 10, LO is only visible in the tray at this startup.

7.1.11 Working with barcodes

It is possible to output a line of text as a barcode. For correct work in Word and Excel reports, Libre Barcode fonts must be installed in the operating system. These fonts are available in the repository librebarcode.

Each font represents a specific barcode format. The following formats are supported:

- Code-128
- Code-39
- EAN13

The font can include the original text directly below the barcode (fonts with Text suffix).

Let's look at the Code-128 format as an example.

Code-128

Fonts to install:

- `LibreBarcode128Text-Regular.ttf`, will be available as "Libre Barcode 128 Text"
- `LibreBarcode128-Regular.ttf`, will be available as "Libre Barcode 128"

How to use

In the template for a regular tag, select a font. For example, "Libre Barcode 128 Text".

Test requests

Document in DOCX format

```
curl --request POST --data-binary "@templates/examples/docx/barcode_code-128.json"␣
↪http://localhost:8087/word_report_json
```

Document in XLSX format

```
curl --request POST --data-binary "@templates/examples/xlsx/barcode_code-128.json"␣
↪http://localhost:8087/excel_report_json
```

7.2 PDF report generator

7.2.1 General description of the service

The service generates a report in PDF format by HTTP-request in JSON format. PDF document is generated using special generator commands.

7.2.2 Generating documents using PDF generator

Service description

Name of service	PDF report generator
Path to service	[host]:[port]/pdf_report_json
Method	POST
Parameters	The request body must contain an object in JSON format. Read more about the structure of the request body in the subsection “Request body structure”. The service responds with a base64-encoded PDF file.
Purpose	The service is designed to generate documents in PDF format.

Request body structure

The body of the request contains an object in JSON format that includes:

1. PDF generator descriptor.
2. Input data to substitute into the template instead of tags.
3. Optional: PDF generator.
4. Request options.

Example:

```
{
    "report-generator":
    {
        "uri": "local",
        "id": "example_1"
    },
```

(continues on next page)

(continued from previous page)

```
    "input-data":
    {
        "ORGANIZATION": "Joint Stock Company Non-State Pension Fund «XXXXX»",
        "CLINAME": "Peter Parker"
    },
    "options":
    {
        "enable-debug-report-save": true
    }
}
```

PDF generator descriptor

The PDF `report-generator` descriptor object contains:

- `uri` - string which specifies the position of the generator command source depending on how the `id` parameter is interpreted. Supported values: `local` and `embedded`.
- `id` - string which is the ID of the generator.

There are two possible ways to use the PDF generator:

- – generator as part of the request (all generator commands are placed inside the `embedded-report-generator` object of the request body);
- – generator as a part of the service (all generator commands are placed inside JSON file in a special `pdf_report_gen` folder on the server).

Therefore:

- When `uri` is set to `embedded`, `id` is used only for diagnostic messages and request identification. The generator must be in the root `embedded-report-generator` object of the request. See "Generators as part of a request" below.
- When `uri` is set to `local`, `id` is the name of the file in the `pdf_report_gen` service directory. See "Generators as part of the service" below.

Generator as part of the request

Optionally, the generator can be integrated into the request.

In this case, all generator commands are specified in the `embedded-report-generator` object.

Example:

```
{
  "report-generator":{
     "id":"example",
     "uri":"embedded"
  },
  "embedded-report-generator":{
     "commands":[
        {
           "name":"NewPage"
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "name":"setCoordinateMode",
            "params":{
               "mode":"millimeters"
            }
         },
         {
            "name":"setCellMargin",
            "params":{
               "margin":1
            }
         }
      ]
   },
   "options":{
      "enable-debug-report-save":false
   }
}
```

Generator as part of the service

Goal: not to pass command data within requests in case they do not change from request to request.

In this case, all generator commands are specified in a JSON file, which must be in the `pdf_report_gen` directory located in the service application directory.

Generator format inside the file in the `pdf_report_gen` directory:

```
{
    "report-generator":
    {
         "commands": [...]
    }
}
```

Example of generator in the body of the request:

```
"report-generator":
   {
       "uri": "local",
       "id": "example"
   }
```

where `id` is the name of the file in the `pdf_report_gen` service directory.

An example of a simple request for the generator inside the example.json file in the `pdf_report_gen` directory:

```
{
   "report-generator":
   {
       "uri": "local",
       "id": "example"
   },
   "options":
   {
       "enable-debug-report-save": true
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Input-data

The `input-data` object contains the input data that is substituted into the tags.

Note: PDF generator commands that work with text may contain tags that will be replaced with input data from the request. More details about using tags can be found in the "Tags" section below.

Example:

```
"input-data":
    {
        "ORGANIZATION": "Joint Stock Company Non-State Pension Fund «XXXXX»",
        "CLINAME": "Peter Parker"
    }
```

Optional: PDF generator

PDF `embedded-report-generator` object contains generator commands. It is used optionally when `uri` is set to `local`.

Details can be found in the subsection "Generators as part of a request".

Request options

The request `options` object has fields:

- `enable-debug-report-save` - Boolean value which specifies whether to create a copy of the document report and a copy of the request (file name - template identifier) in the local `reports_debug` directory of the service. The default value is `false`.
- `enable-binary-output` - Boolean value which indicates that the service will output the result as binary data, without base64 encoding. The default value is `false`.
- `enable-debug-pdf-log` - Boolean value which enables extended diagnostic log of PDF creation. Default value is `false`.

Example:

```
"options":
    {
        "enable-debug-merged-doc-save": true,
        "enable-binary-output": false,
        "enable-debug-pdf-log": true
    }
```

7.2.3 Service call examples

```
curl --request POST --data-binary "@templates/examples/pdf/embedded-report-generator/
↪request_example.json" http://localhost:8087/pdf_report_json
```

Response structure

The service response contains an object in JSON format that includes:

1. Error description (error code, error message). In case of successful service response, the value `null` is returned.
2. Result (base64 encoded generated PDF file of the document).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded pdf file]"
}
```

Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Using page templates

One can use ready-made PDF page templates to generate a document with the PDF generator.

There is a special command `setPageTemplate` to set the template of a particular page. One can read more about this command in the “Generator commands” subsection.

Note: Multi-page templates are supported.

To position text and other elements on a template page, one needs to set coordinates. One can read more about specifying coordinates of document elements in the “Coordinates” subsection.

Page `templates` in `PDF` format are stored locally in the `templates/pdf` directory located in the service application directory.

Service directories

PDF generation scripts are stored in the `pdf_report_gen` directory located in the service application directory.

Page `templates` in `PDF` format are stored locally in the ``templates/pdf``directory located in the service application directory.

Requests examples are located in the directory `query_examples/pdf`.

Fonts to be used in the generator are located in the `assets/fonts` directory located in the service application directory. Fonts are loaded at service startup.

Note: One needs to restart the service in order to add new font files.

Examples of requests and templates

Examples of requests are located in the directory `examples/pdf/embedded-report-generator` and `examples/pdf/report-generator` in the *examples archive*.

Examples of templates can be found in the `examples/pdf` directory in the *examples archive*.

7.2.4 Generator commands

A PDF document is generated using special generator commands.

The commands are located inside the ``embedded-report-generator` object if the generator is embedded in a request (see “Generator as part of a request” above).

Commands can also be located in a JSON format file, which is located in the `pdf_report_gen` directory in the service application directory (see “Generator as part of the service” above).

Commands are described inside the `commands` array.

Example:

```
"commands": [
        {
           "name":"NewPage"
        },
        {
           "name":"setCoordinateMode",
           "params":{
              "mode":"millimeters"
           }
        },
        {
           "name":"setCellMargin",
           "params":{
              "margin":1
           }
        }
     ]
```

The `commands` array can include the commands described below.

NewPage

Creates a new page in the document.

Parameters

- `orientation` - string with supported values:
  - "L" - landscape orientation
  - "P" - portrait orientation

The default setting is "P".

Example:

```
{
  "name":"NewPage",
  "params":{
      "orientation":"L"
   }
}
```

setCoordinateMode

Specifies the interpretation of coordinates and dimensions specified in subsequent commands. The default mode is "in millimeters".

Parameters

- `mode` with supported values: "millimeters", "percents", "normalized".

setPageMargins

Sets the indentation from the page borders.

Parameters

`left`, `right`, `top`, `bottom` - floating-point numbers specifying indents from the left, right, top, bottom edges of the page. When the bottom margin is reached in commands of Row_Print type, a new page is automatically created. The command replaces the plpdf functions `setAllMargin` and `SetAutoNewPage`.

Example:

```
{
  "name":"setPageMargins",
  "params":{
     "left":20,
     "right":20,
     "top":30,
     "bottom":15
    }
 }
```

setCurrentX

Sets the X-coordinate of the cursor.

Parameters

- `value` - floating-point number. X-coordinate.

setCurrentY

Sets the Y-coordinate of the cursor.

Parameters

- `value` - floating-point number. Y-coordinate.

setCurrentXY

Sets the coordinates of the cursor.

Parameters

- `x` is a floating-point number. X-coordinate.
- `y` is a floating-point number. Y-coordinate.

saveX

Saves the X-coordinate for later use in command coordinate parameters via the tag `[PDF:savedX]`.

Parameters

- `value` - optional parameter, by default equal to the current X-coordinate. It can be a floating-point number or a string-expression with references to embedded tags. Example of expression: `"[PDF:currentX] - 50"`

saveY

Saves the Y-coordinate for later use in command coordinate parameters via the tag `[PDF:savedY]`.

Note: the value may become irrelevant when creating a new page later.

Parameters

- `value` - optional parameter, by default equal to the current Y-coordinate. It can be a floating-point number or a string-expression with references to embedded tags. Example expression: `"[PDF:currentY] - 50"`

saveCoordinate

Saves the specific coordinate value for later use in the coordinate parameters of commands via the tag `[SAVED:ключ]`. Supplements `saveX/saveY` commands in cases where multiple values need to be accessed.

Parameters

- `key` - string which is random key for use in the tag [SAVED:ключ]
- `value` - can be a floating-point number or a string-expression with references to embedded tags. Example expression: `"[PDF:currentY] - 50"`

Example:

```
{
  "name":"saveCoordinate",
  "params":{
     "key":"x1",
     "value":10
  }
}
```

setRotate

Sets the rotation of all subsequent objects.

Parameters

- `angle` - a floating-point number is the angle of rotation specified in degrees.
- `x`- is a floating-point number. X-coordinate of the rotation point. If the value is null, then X-coordinate of the cursor/
- `y` - a floating-point number. Y-coordinate of the rotation point. If null, then Y-coordinate of the cursor.

Example:

```
{
  "name":"setRotate",
  "params":{
     "angle":45,
     "x":105,
     "y":130,
   }
}
```

Command cancellation:

```
{
  "name":"setRotate",
  "params":{
     "angle":0
  }
}
```

startOpacity

Specifies the degree of transparency for all subsequent objects. The action is canceled by the `endOpacity` command.

Parameters

- `val` - a floating-point number. The acceptable number range is from 0.0 to 1.0, where zero is full transparency.

endOpacity

Cancels the `startOpacity` command.

No parameters.

SetPrintFont

Sets the font and font size for subsequent text commands.

Parameters

- `font_id` supported values: names of font files from `assets/fonts` directory without extension. Fonts are loaded at service startup.
- `size` font size in points in the user space of the pdf document. There are no differences from PLPDF and other text editors.

Note: The only supported charset is cp1251.

setColor4Text

Sets the text color for subsequent text commands.

Input options

- `r`, `g`, `b` integer decimal components of RGB color in the range from 0 to 255. Example of red color: “r”:255, “g”:0, “b”:0.
- `color` string in the format “#rrggbb”, where rgb are hexadecimal components of RGB color in the range from 0 to ff. Example of red color: “color”: “#ff0000”.

setColor4Drawing

Sets the border color for objects output after this command is executed. Examples of objects: table cells, circles, lines, rectangles, etc.

Параметры: аналогично команде `setColor4Text`.

setColor4Filling

Sets the fill color of the internal areas for objects output after this command is executed. Examples of objects: table cells, circles, rectangles, etc.

Parameters are similar to the `setColor4Text` command.

setLineWidth

Sets the thickness of the lines to be output after this command is executed. Examples of objects: table cell borders, lines, rectangles, etc.

Parameters

- `width` - a floating-point number. Line thickness.

setCellMargin

Sets the table cell content indents from the left, top, right, and bottom borders.

Parameters

- `margin` is a floating-point number.

setCellBottomMargin

Sets the table cell content indentation from the bottom border.

Parameters

- `margin` is a floating-point number.

setCellLeftMargin

Sets the table cell content indentation from the left border.

Parameters

- `margin` - is a floating-point number.

setCellTopMargin

Sets the table cell content indentation from the top border.

Parameters

- `margin` is a floating-point number.

setCellRightMargin

Sets the table cell content indentation from the right border.

Parameters

- `margin` is a floating-point number.

PrintCell

Outputs a rectangular cell with text inside. One can specify the color and the presence of borders.

Parameters

- `w` - a floating-point number. The width of the cell rectangle.
- `h` is a floating-point number. Height of the cell rectangle.
- `txt` - text
- `border` - string which sets the visible cell borders. Valid values: 0 - no border, 1 - outer rectangle border, L - left, T - top, R - right, B - lower border. A combination of L,T,R and B values is allowed.
- `align` - string which defines horizontal text alignment. Values: L - left edge. R - right edge, C - center, J - justified, aligns evenly across the cell width.
- `vert_align` - string which defines vertical text alignment. Values: T - top edge. B - bottom edge, C - center. Default value: C.
- `fill` - Boolean value indicating whether to fill the cell with the current color. See `setColor4Filling`. The default value is false.
- `link` - string - a link, such as a URL or an internal link - is not supported in the current version.
- `clipping` - Boolean value specifying whether to clip the text at the cell borders. The default value is false.
- `ln` - integer specifying the cursor position after cell drawing. Valid values: 0 - next to the cell, 1- new line, 2 - under the cell. The default value is 0.

Example:

```
{
  "name":"PrintCell",
  "params":{
     "w":120,
     "h":7,
     "txt":"Personal Data",
     "border":"0",
     "align":"L",
     "vert_align":"C",
     "fill":true,
     "clipping":false,
     "ln":2
   }
}
```

PrintMultiLineCell

Outputs a multi-line cell

Parameters

- `w` - floating-point number. The width of the cell rectangle.
- `h` is a floating-point number. Height of the cell rectangle.
- `txt` - text
- `border` - string which sets the visible cell borders. Valid values: 0 - no border, 1 - outer rectangle border, L - left, T - top, R - right, B - lower border. A combination of L,T,R and B values is allowed.
- `align` - string which defines horizontal text alignment. Values: L - left edge. R - right edge, C - center, J – justified, aligns evenly across the cell width.
- `vert_align` - string which defines vertical text alignment. Values: T - top edge. B - bottom edge, C - center. Default value: C.
- `fill` - Boolean value indicating whether to fill the cell with the current color. See `setColor4Filling`. The default value is false.
- `maxline` - is an array of floating-point numbers. The array element specifies the maximum number of text lines in a multiline cell.
- `link` - string a link, such as a URL or an internal link - is not supported in the current version.
- `clipping` - Boolean value specifying whether to clip the text at the cell borders. The default value is false.
- `indent` is floating-point number which specifies the indent for the first line of the text. The default value is 0.
- `ln` - an integer specifying the cursor position after cell drawing. Valid values: 0 - next to the cell, 1- new line, 2 - under the cell. The default value is 0.

Example:

```
{
  "name":"PrintMultiLineCell",
  "params":{
     "h":5,
     "w":95,
     "txt":"Amount of pension savings received by the fund (taking into account␣
↪investment income received from investment of pension
     funds) when the mandatory pension insurance contract comes into force.",
     "border":"LTR",
     "align":"L",
     "vert_align":"T",
     "fill":true,
     "indent":0,
     "clipping":false,
     "ln":0
   }
}
```

Row_Print

Prints a line in a PDF document. A line consists of multi-line cells (see `PrintMultiLineCell`). The height of the line is determined by the cell with the highest height.

Parameters

- `data` - array of strings. The array element specifies the cell text in the string.
- `input_data_tag` - alternative to data, the name of the array of cell string-values in the query input data.
- `width` is an array of floating-point numbers. The array element specifies the cell width.
- `border` - array of strings. The array element describes the visible borders of the cell. See the command description `PrintMultiLineCell`. By default, all borders are visible.
- `maxline` is an array of floating-point numbers. The array element specifies the maximum number of text lines in a multiline cell. There is no limit by default.
- `align` - array of rows. The array element specifies the horizontal alignment in the cell.
- `vert_align` - array of strings. The array element specifies the vertical alignment in the cell.
- `font` - an array of font descriptor objects. The array element specifies the font parameters in the cell. The fields of each descriptor are similar to the parameters of the `SetPrintFont` command. The default value is null, which means that the font settings set in the previous `SetPrintFont` call are used.
- `h` - floating-point number. The height of the cell. The default value is 5 units.
- `fill` - Boolean value. Indicates whether to fill the cell with the current color to fill. See setColor4Filling. The default value is false.
- `min_height` is a floating-point number. The minimum height of the string. Zero means that the parameter is not used. The default value is 0.
- `clipping` - Boolean value. Specifies whether to clip the text to the cell borders. The default value is false.

Example:

```
{
 "name":"Row_Print",
 "params":{
    "h":4.5,
    "min_height": 27,
    "width":[85, 95],
    "data":[
       "1.4 Funds (part of funds) of the maternity (family) capital (minuspart of the␣
↪maternity (family) capital funds returned to the
       Pension Fundin case the insured person refuses to use them as formation of␣
↪pension savings and choice of other uses,including
       income received from their investment))
       "0,00"
      ],
    "align":["L", "L"],
    "vert_align":["T", "T"],
    "border":["0", "0"],
    "font": [
     {
      "size":6,
      "font_id":"Arial-Bold"
     },
     {
```

(continues on next page)

(continued from previous page)

```
     "size":6,
     "font_id":"Arial"
     }
   ],
    "fill":true,
    "clipping":false
 }
}
```

PrintText

Outputs text at the specified coordinates.

Parameters

- x is a floating-point number. X-coordinate of the beginning of the text
- y is a floating-point number. Y-coordinate of the beginning of the text
- text - text

Example:

```
{
  "name":"PrintText",
  "params":{
    "x":10,
    "y":40,
    "text":"List of documents:"
   }
}
```

underline

Underlines the text printed with the `PrintText` command.

The parameters are the same as `PrintText`, the parameter values must match.

```
{
  "name":"underline",
  "params":{
     "x":10,
     "y":40,
     "text":"List of documents:"
   }
}
```

LineBreak

Line break moves the cursor to the beginning of the next line.

Parameters

- `h` - a floating-point number specifying the indentation from the current Y-coordinate of the cursor.

Line

Draws a straight line between two points on the page.

Parameters

- `x1` is a floating-point number. X-coordinate of the line start.
- `y1` is a floating-point number. Y-coordinate of the line start.
- `x2` is a floating-point number. X-coordinate of the end of the line.
- `y2` is a floating-point number. Y-coordinate of the end of the line.
- `color` string in the format "#rrggbb", where rgb are hexadecimal components of RGB color in the range from 0 to ff. An example of red color: "color": "#ff0000". By default, the color pre-set by the `setColor4Drawing` command is used.
- `width` - floating-point number which defines the thickness of the line. The default value is the value previously set with the `setLineWidth` command.

Example:

```
{
  "name":"Line",
  "params":{
        "x1":100,
        "y1":230,
        "x2":150,
        "y2":230,
        "color":"#000000",
        "width":0.2
    }
}
```

Circle

Draws a circle on the page.

Parameters

- `x` is a floating-point number. X-coordinate of the center of the circle.
- `y` is a floating-point number. Y-coordinate of the center of the circle.
- `r` is a floating-point number specifying the radius of the circle.
- `draw` - Boolean value which controls drawing of the circle outline. The default value is true.
- `fill` - Boolean value which controls the fill color of the inner area of the circle. The default value is false.
- `draw_color` defines the color of the circle outline. A string in the format "#rrggbb", where rgb is the hexadecimal RGB color components in the range from 0 to ff. Example of red color: "color": "#ff0000". By default, the color previously set with the `setColor4Drawing` command is used.

- `fill_color` is the color of the circle outline. A string in the format "#rrggbb", where rgb is the hexadecimal components of RGB color in the range from 0 to ff. Example of red color: "color": "#ff0000". The default color is the color previously set with the `setColor4Filling` command.
- `linewidth` - is a floating-point number which defines the thickness of the circle outline. The default value is the value previously set with the `setLineWidth` command.

Example:

```
{
  "name": "Circle",
  "params":
      {
        "x": 135,
        "y": 250,
        "r": 15,
        "draw": false,
        "fill": false,
        "draw_color": "#ff0000",
        "fill_color": "#ff0000",
        "linewidth": 0.1
    }
 }
```

putImage

Parameters

- `name` - string. The name of the image or image ID. If the name matches the one specified in the previous call, the image specified in the same call is used.
- `data` - string. Base64 encoded image file. If `name` is the same as the one specified in the previous call, this parameter is not specified.
- `x` - is a floating-point number. X-coordinate of the image.
- `y` - is a floating-point number. Y-coordinate of the image.
- `w` - a floating point number which defines the width of the image in units specified by the command `setCoordinateMode`. If the parameter is not specified, the image is displayed in full width, which is calculated based on the pixel density of the PDF document (72 dots per inch) and the width of the image according to the image file. *Example*: a 72x72 dots image will be 1 inch by 1 inch in a PDF document.
- `h` - floating-point number which defines the image height. Units: similar to the parameter w.
- `link` - string - a URL or ID of internal link is not supported in the current version.

Example:

```
{
  "name":"putImage",
  "params":{
      "name":"img",
      "data":"base64 encoded image",
      "x":100,
      "y":205,
      "w":50,
      "h":37.5
    }
}
```

setPageTemplate

Parameters

- `template` - object in format

```
{
 "uri": "local",
 "id": "template_example"
}
```

where template ID is the path to the pdf document relative to the `templates/pdf` directory without extension.

- `page` - an integer defines the page number of the template to output. It ranges from one to the number of pages in the template.

Example command for a multi-page template:

```
{
 "name":"setPageTemplate",
 "params":{
    "template":{
       "id":"example_multipage_template",
       "uri":"local"
     },
    "page":1
    }
}
```

Note: Duplication of font resources in the report occurs when using the `setPageTemplate` command when the font matches the one loaded by the `setPrintFont` command. This may increase the size of the report file.

checkPageBreak

If adding the specified height to the current Y-coordinate results in a page overflow (i.e. going beyond the bottom page indent), the command adds a new page and returns true (the return value has a value only in the `if` command)

Parameters

- `h` - floating-point number which defines the height to check for page overflow.
- `newpage` - Boolean value which specifies whether to create a new page in the document in case of an overflow. The default value is true.

if

The command allows to execute a set of subcommands if the condition is true immediately at the moment of command execution (unlike `register_auto_new_page_commands`).

The supported command set to use as a condition: `checkPageBreak`.

- `condition` - object describing the command that returns a logical value.
- `commands` - array of commands to be executed in case the command specified in the `condition` parameter returned true.

Example:

```
{
 "name": "if",
 "params": {
     "condition": {
         "name":"checkPageBreak",
         "params": {
             "h": 4.5,
             "newpage": true
         }
     },
     "commands": [
         {
             "name":"setPageTemplate",
             "params":{
                "template":{
                   "id":"example_template",
                   "uri":"local"
                }
             }
         },
         {
             "name":"setCurrentXY",
             "params":{
                 "x":10,
                 "y":30
             }
         }
     ]
  }
}
```

Typical use: before attempting to display an element or set of elements at the bottom of a page, when there is a risk of exceeding the bottom margin of the page. The `h` value to check is usually taken from the size of the element to be displayed.

Note: The if command works directly in the place where it is called, with the current Y-coordinate. That is, if it is called at the moment when there is a vertical space of height h on the page, the list of subcommands will not be executed.

Note: It is not recommended to use the if command to output table rows. It can be useful for outputting a unique element when rendering is closer to the bottom of the page. In case of tables, it is better to use `register_auto_new_page_commands`.

register_auto_new_page_commands

Allows to register a set of subcommands to be executed when execution of `Row_Print` type commands results in automatic creation of a new document page. Can be useful for table header rendering. It operates from the moment it is called. It can be canceled by calling the same command with an empty list of subcommands.

Parameters

- `commands` - an array of commands to execute.

Example:

```
{
  "name": "register_auto_new_page_commands",
  "params": {
      "commands": [
          {
            "name":"setPageTemplate",
            "params":{
                 "template":{
                    "id":"example_template",
                    "uri":"local"
                 }
             }
          },
          {
            "name":"setCurrentXY",
            "params":{
                 "x":10,
                 "y":30
           }
        }
      ]
   }
}
```

Command cancellation:

```
{
  "name": "register_auto_new_page_commands",
  "params": {
      "commands": []
  }
}
```

7.2.5 Coordinates

Commands that accept coordinates and dimensions as parameters interpret the values based on the call to the `setCoordinateMode` command. Millimeters are used by default.

All such parameters can be specified numerically or textually.

In text form, expressions and special tags `PDF:currentX/PDF:currentY` are supported.

Example:

```
{
    "name": "Line",
    "params":
    {
        "x1": "[PDF:currentX]",
        "y1": "[PDF:currentY]",
        "x2": "[PDF:currentX] + 50",
        "y2": "[PDF:currentY]",
        "color": "#ff0000",
        "width": 0.1
    }
}
```

7.2.6 Tags

Commands that work with text (`PrintText`, `PrintCell`, etc.) can contain tags that will be replaced with input data from the request.

The input data is contained in the body of the request with the `input-dat` object: `{...}` (see the "Input data" item in the "Query structure" subsection).

Tag format

The tag is specified in square brackets. For example, [ORGANIZATION].

Example:

```
{
  "name":"PrintMultiLineCell",
  "params":{
     "h":4,
     "w":77,
     "ln":2,
     "txt":"[ORGANIZATION], hereinafter referred to as the "Fund", represented by the␣
↪General Director John Smith, acting on the basis of power
      of attorney №11142 dated 02.04.2018, with. on the one hand        on the one␣
↪hand, and [CLINAME], hereinafter referred to as «Participant», on
      the other hand, have entered into this Agreement as follows:",
     "fill":false,
     "align":"J",
     "border":"0",
     "indent":15,
     "clipping":false
  }
}
```

No brackets are specified in the request (see the example from the "Input data" item of the "Request structure" subsection).

Embedded tags

Some tags are embedded in the PDF generator.

1. `PDF:currPageNum`, current page number.
2. `PDF:currentX/PDF:currentY`, X and Y-coordinates of the cursor.
3. `PDF:savedX/PDF:savedY`, available after calling `saveX/saveY` commands.
4. `[SAVED:ключ]`, available after the `saveCoordinate` command.

Example:

```
{
  "name":"Line",
  "params":{
     "x1":"[PDF:savedX]",
     "x2":"[SAVED:endLineX]",
     "y1":"[PDF:currentY]",
     "y2":"[PDF:currentY]",
     "color":"#E8E8E8",
     "width":0.8
```

(continues on next page)

(continued from previous page)

```
  }
},
{
  "name":"PrintText",
  "params":{
     "x":"[SAVED:pageNumX]",
     "y":"[SAVED:pageNumY]",
     "text":"Номер страницы: [PDF:currPageNum]"
  }
}
```

7.2.7 Working with barcodes

It is possible to output information as a barcode by using the Libre Barcode family of fonts. The fonts are available in the librebarcode. Each font represents a specific barcode format. The following formats are supported:

- Code-128
- Code-39
- EAN13

The font can include the original text directly below the barcode (fonts with Text suffix).

Let's look at the Code-128 format as an example.

Code-128

Fonts to be placed in the `assets/fonts` folder of the service:

- LibreBarcode128Text-Regular.ttf
- LibreBarcode128-Regular.ttf

How to use

In the command generator, select the desired font, for example:

```
{
    "name": "SetPrintFont",
    "params":
    {
        "font_id": "LibreBarcode128Text-Regular",
        "size": 30
    }
}
```

Test request

```
curl --request POST --data-binary "@templates/examples/pdf/embedded-report-generator/
↪request_barcode_code-128.json" http://localhost:8087/pdf_report_json
```

The PDF `report-generator` descriptor object contains:

- `uri` - string specifing the position of the generator command source depending on how the `id` parameter is interpreted. Supported values: `local` and `embedded`.
- `id` - string. ID of the generator.

There are two possible uses for the PDF generator:

- generator as part of the request (all generator commands are placed inside the `embedded-report-generator` object of the request body);
- generator as a part of the service (all generator commands are placed inside JSON file in a special `pdf_report_gen` folder on the server).

Therefore:

- When `uri` is set to `embedded`, `id` is used only for diagnostic messages and request identification. The generator must be located in the `embedded-report-generator` root object of the request. See “Generators as part of a request” below.
- When `uri` is set to `local`, `id` is the name of the file in the `pdf_report_gen` service directory. See “Generators as part of the service” below.

Generator as part of the request

Optionally, the generator can be integrated into the request.

In this case, all generator commands are specified in the `embedded-report-generator` object.

Example:

```
{
  "report-generator":{
     "id":"example",
     "uri":"embedded"
  },
  "embedded-report-generator":{
     "commands":[
        {
           "name":"NewPage"
        },
        {
           "name":"setCoordinateMode",
           "params":{
              "mode":"millimeters"
           }
        },
        {
           "name":"setCellMargin",
           "params":{
              "margin":1
           }
        }
```

(continues on next page)

(continued from previous page)

```
      ]
   },
   "options":{
      "enable-debug-report-save":false
   }
}
```

Generator as part of the service

Goal: not to pass command data within requests unless it changes from request to request.

In this case, all generator commands are specified in a JSON file, which must be located in the `pdf_report_gen` directory located in the service application directory.

Generator format inside the file in the `pdf_report_gen` directory:

```
{
    "report-generator":
    {
         "commands": [...]
    }
}
```

Generator example in the body of the request:

```
"report-generator":
   {
       "uri": "local",
       "id": "example"
   }
```

where `id` is the name of the file in the `pdf_report_gen` service directory.

An example of a simple request for the generator inside the example.json file in the `pdf_report_gen` directory:

```
{
   "report-generator":
   {
       "uri": "local",
       "id": "example"
   },
   "options":
   {
       "enable-debug-report-save": true
   }
}
```

Input data

The `input-data` object contains the input data that is substituted into the tags.

Note: PDF generator commands that work with text may contain tags that will be replaced with input data from the request. More details about using tags can be found in the "Tags" section below.

Example:

```
"input-data":
    {
        "ORGANIZATION": "Joint Stock Company Non-State Pension Fund.«XXXXX»",
        "CLINAME": "Peter Parker"
    }
```

Optional: PDF generator

PDF `embedded-report-generator` object contains generator commands. It is used optionally when `uri` is set to `local`.

It is described in more detail in "Generators as part of a request".

Request options

The request `options` object has fields:

- `enable-debug-report-save` - Boolean value specifying whether to create a copy of the document report and a copy of the request (file name - template identifier) in the local `reports_debug` directory of the service. The default value is `false`.
- `enable-binary-output` - Boolean value indicating that the service will output the result as binary data, without base64 encoding. The default value is `false`.
- `enable-debug-pdf-log` - Boolean value enabling extended diagnostic log of PDF creation. Default value is `false`.

Example:

```
"options":
    {
        "enable-debug-merged-doc-save": true,
        "enable-binary-output": false,
        "enable-debug-pdf-log": true
    }
```

7.2.8 Service call examples

```
curl --request POST --data-binary "@templates/examples/pdf/embedded-report-generator/
↪request_example.json" http://localhost:8087/pdf_report_json
```

Response structure

The service response contains an object in JSON format that includes:

1. Error description (error code, error message). In case of successful service response, the value `null` is returned.
2. Result (base64 encoded generated PDF file of the document).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded pdf file]"
}
```

Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Using page templates

One can use ready-made PDF page templates to generate a document with the PDF generator.

There is a special command `setPageTemplate` to set the template of a particular page. One can read more about this command in the "Generator Commands" subsection.

Note: Multi-page templates are supported.

To position text and other elements on a template page, one must set coordinates. You can read more about specifying coordinates of document elements in the "Coordinates" subsection.

Page `templates` in PDF format are stored locally in the ``templates/pdf``directory located in the service application directory.

Service directories

PDF generation scripts are stored in the `pdf_report_gen` directory located in the service application directory.

Page templates in PDF format are stored locally in the `templates/pdf` directory located in the service application directory.

Examples of requests are located in the directory request_examples/pdf.

Fonts for use in the generator are located in the `assets/fonts` directory located in the service application directory. Fonts are loaded at service startup.

Note: One needs to restart the service in order to add new font files.

Examples of requests and templates

Examples of requests are located in the directory `examples/pdf/embedded-report-generator` and `examples/pdf/report-generator` in the *examples archive*.

Example of templates can be found in the `examples/pdf` directory in the *examples archive*.

7.3 PDF document merging service

7.3.1 General description of the service

The service combines documents in PDF format into one document by HTTP request in JSON format.

Note: XML format is not supported in the current version.

7.3.2 Merging PDF documents

Service Description

Service name	CPDF merge service
Path to service	[host]:[port]/pdf_merge_json
Method	POST
Parameters	The request body must contain an object in JSON format. Read more about the structure of the request body in the subsection “Request body structure”. The service responds with a base64-encoded PDF file.
Purpose	The service is designed to combine several documents in PDF format into one document.

Request body structure

The body of the request contains an object in JSON format that includes:

1. Request ID.
2. Input data: list of document descriptors to merge.
3. Request options.

Example:

```
{
    "request-id": "…",
    "input-data": {
        "document-descriptors": [
            {…},
            {…}
        ]
    },
    "options": {…}
}
```

Request ID

The `request-id` is used to write the merged file in debug mode and to identify the request in the log.

Input data

The `input-data` object contains:

A list (array) of `document-descriptors` to merge.

Example:

```
"input-data": {
        "document-descriptors": [
            {…},
            {…}
        ]
    }
```

Input document descriptor

The input document descriptor object supports fields:

- `id` - string. Document identifier used for references to the input parameter in error messages and diagnostic log.
- `content-type` - string indicating how the content parameter is interpreted. Supported value: base64.
- `content` - string. If the content type is base64, then base64 encoded PDF document. Note: There is no limitation on the number of pages in the document.

Example:

```
{
    "id": "doc1",
    "content-type": "base64",
    "content": "[doc 1 in base64 format]"
}
```

Request options

The request `options` object has fields:

- `enable-debug-merged-doc-save` - Boolean value specifying whether to create a copy of the document report and a copy of the request (file name - template identifier) in the local `reports_debug` directory of the `service`. The default value is `false`.
- `enable-binary-output` - Boolean value indicating that the service will output the result as binary data, without base64 encoding. The default value is `false`.
- `enable-debug-pdf-log` - Boolean value enabling extended diagnostic log of PDF creation. Default value is `false`.

Example:

```
"options":
    {
        "enable-debug-merged-doc-save": true,
        "enable-binary-output": false,
        "enable-debug-pdf-log": true
    }
```

Request body example

```
{
    "request-id": "pdf_merge_example1",
    "input-data": {
        "document-descriptors": [
            {
                "id": "doc1",
                "content-type": "base64",
                "content": "[doc 1 in base64 format]"
            },
            {
                "id": "doc2",
                "content-type": "base64",
                "content": "[doc 2 in base64 format]"
            }
        ]
    },
    "options":
    {
        "enable-debug-merged-doc-save": true,
        "enable-debug-pdf-log": true
    }
}
```

Example requests are located in the `examples/pdf/pdf_merge` directory in the *examples archive*.

Service call examples

```
curl --request POST --data-binary "@templates/examples/pdf_merge/request_example.json
↪" http://localhost:8087/pdf_merge_json
```

Response structure

The service response contains an object in JSON format that includes:

1. Error description (error code, error message). In case of successful service response, the value `null` is returned.
2. Result (base64 encoded PDF file of the merged document).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded pdf file]"
}
```

Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Note: The merged document may be larger than the sum of the merged documents. This is due to duplication of the same resources (e.g., fonts) used in incoming documents.

7.4 Print form generation service

7.4.1 General description of the service

The service generates a printed form of the document in PDF format by HTTP-request in JSON format:

- A signature table is generated on the last page of the document.
- Intermediate pages display a footer with brief information about the document.

Note: XML format is not supported in the current version.

7.4.2 Generating a printed form of a document in PDF format

Service description

Service	Print form generation service
Path to service	[host]:[port]/print_form_pdf_json
Method	POST
Parameters	The request body must contain an object in JSON format. Read more about the structure of the request body in the subsection "Request body structure". In response, the service gives a base64 encoded PDF file of the printed form of the document.
Purpose	The service is designed to generate a printed form of the document in PDF format, including footers with brief information about the document and a signature table on the last page.

Request body structure

The body of the request contains an object in JSON format that includes:

1. Request ID.
2. Inputs:
   - Descriptor of the input document to create the printed form (the original PDF document to create the printed form).
   - Descriptor of the signature table and footer of intermediate pages (data to be inserted in the signature table and footer of intermediate pages with data about the document to be signed).
   - Options for the appearance of generated tables (color/size of fonts).
3. Request options.

Example:

```
{
    "request-id": "…",
    "input-data":
    {
        "input-document": {…},
        "signatures-info": {…},
        "draw-options": {…}
    },
    "options": {…}
  }
```

Request ID

The `request-id` is used to write the file in debug mode and to identify the request in the log.

Input data

The `input-data` object contains:

- Descriptor of the input document to create the printed form.
- Descriptor of the signature table and footer of the intermediate pages.
- Options for the appearance of tables to be generated.

Example:

```
"input-data":
   {
       "input-document": {…},
       "signatures-info": {…},
       "draw-options": {…}
   }
```

Input document descriptor

The `input-document` descriptor object for creating a printed form has fields:

- `id` - string is a document identifier used for references to the input parameter in error messages and diagnostic log.
- `content-type` - a string indicating how the content parameter is interpreted. Supported value: base64.
- `content` - string. If the content type is base64, then base64 encoded PDF document. Note: There is no limitation on the number of pages in the document.

Example:

```
"input-document":
        {
         "id": "doc1",
         "content-type": "base64",
         "content": "[base64 encoded pdf file]"
        }
```

Descriptor of the signature table and footer of intermediate pages

The descriptor object of the signature table and footer of intermediate pages `signatures-info` contains the following fields:

- `signature-summary-text` - a string which contains a general description of the signature table to output before the signers' data.
- `table-header-texts`- an array of strings which contains the list of signature table headers. Note: in the current version there can be 4 columns only (limitation of the `signatures array` element described below).
- `table-header-column-width-list` - an array of floating-point numbers which controls the width of table columns. The sum of all normalized widths must equal one. The array size must match the size of the `table-header-texts` field

- `signatures` - an array of objects with signers' data. See "Signer data descriptor" below for details.
- `intermediate-page-footer-info` - data to be displayed in the footer on all pages of the document except the last page, where the signature table is located. For details, see "Descriptor of footer of intermediate pages with data on the document to be signed" below.
- `logo-asset-name` - a string. Optional parameter. Name of the image file (with extension) to be displayed in the upper right corner of the signature table. The path is set relative to the `assets/images/print_forms` directory in the working directory of the service.

Example:

```
"signatures-info":  {
    "signature-summary-text": "…",
    "table-header-texts": […],
    "table-header-column-width-list": […],
    "signatures": […],
    "intermediate-page-footer-info": {…},
    "logo-asset-name": "…"
}
```

Signer data descriptor

The array element of the `signatures` data descriptor has fields:

- `description` - a string which contains general description of the signature's ownership. The text is displayed in the first column of the signature table.
- `certificate-owner` - an object with string fields `organization` and `employee`, describing organization, full name and position of certificate-owner. The text is displayed in the second column.
- `certificate` - an object with string fields `serial-number` and `validity` describing serial number and validity period of the certificate used for signing. The text is output in the third column.
- `signing-timestamp` - a string which contains a description of the exact time of signing. The text is output in the fourth column.

Example:

```
{
     "description": "Sender signature",
     "certificate-owner":
     {
         "organization": "LLC CIRCLE",
         "employee": "John Smith, General Director"
     },
     "certificate":
     {
         "serial-number": "11255807011ABCBBE4DBF93ECCCB96C45",
         "validity": "с 01.02.2001 18:48 по 01.02.2031 18:48 GMT+03:00"
     },
     "signing-timestamp": "01.02.2021 19:00 GMT+03:00"
 }
```

Footer descriptor of intermediate pages with data on the document to be signed

Intermediate page footer descriptor object with data about the document to be signed `Intermediate-page-footer-info` has fields:

- `lines`- an array of lines. Specific data describing the document to be signed. It is output as a table with invisible borders. The last line will be a description of the current page number, if enabled by the `print-page-number` field.
- `logo-asset-name` - a string. Optional parameter. Name of the image file (with extension) for output in the right part of the document data. The path is set relative to the directory `assets/images/print_forms` in the working directory of the service.
- `print-page-number` - Boolean value specifying whether to print the current page number of the document under the document data.

Example:

```
{
     "lines": [
         "Передан через XXX 01.02.2003 15:20 GMT+03:00",
         "abcdef00-0000-1111-2222-abcdefabcdef"
     ],
     "logo-asset-name": "golang-gopher.png",
     "print-page-number": true
 }
```

Appearance options of the generated tables

The `` ` ``draw-options object of the generated table appearance options has fields:

- `font-color` is a string in the format "#rrrggbb", where rgb is the hexadecimal components of RGB color in the range from 0 to ff. Example of red color: "font-color": "#ff0000".
- `font-size` - an integer. The font size of the signers' data in the signature table. The unit is equal to 1/72 of an inch.
- `large-font-size` is an integer. The font size of signature-summary-text output in the signature tab. The unit is equal to 1/72 of an inch.
- `table-border-color` - a string in the format "#rrggbb", where rgb - hexadecimal components of RGB color in the range from 0 to ff. Example of red color: "font-color": "#ff0000". Signature table border color.

Example:

```
{
    "font-color": "#0054BE",
    "font-size": 8,
    "large-font-size": 14,
    "table-border-color": "#0054BE"
}
```

Request options

The request `options` object has fields:

`enable-debug-merged-doc-save` - Boolean value specifying whether to create a copy of the document report and a copy of the request (file name - template ID) in the local `reports_debug` directory of the service. The default value is `false`. `enable-binary-output` - Boolean value indicating that the service will output the result as binary data without base64 encoding. The default value is `false`. `enable-debug-pdf-log` - Boolean value which enables extended diagnostic log of PDF creation. Default value is `false`.

Example:

```
"options":
   {
       "enable-debug-doc-save": true,
       "enable-binary-output": false,
       "enable-debug-pdf-log": true
   }
```

Example of a request body

```
{
   "request-id": "print_form_pdf_example1",
   "input-data":
   {
       "input-document":
       {
           "id": "doc1",
           "content-type": "base64",
           "content": "JVBERi0xLjcKJYGBgYEKCjYgMCB….VmCjI1ODI5CiUlRU9G"
       },
       "signatures-info":
       {
           "signature-summary-text": "The document is signed and transmitted through␣
↪the operator of the JSC␣'→"Example",
           "table-header-texts": ["", "Владелец сертификата: организация, сотрудник",
↪"Certificate: serial number, validity period", "Date and time of signing"],
           "table-header-column-width-list": [0.15, 0.33, 0.32, 0.20],
           "signatures": [
               {
                   "description": "Sender signature",
                   "certificate-owner":
                   {
                       "organization": "LLC CIRCLE"",
                       "employee": "John Smith, General Director"
                   },
                   "certificate":
                   {
                       "serial-number": "11255807011ABCBBE4DBF93ECCCB96C45",
                       "validity": "с 01.02.2001 18:48 по 01.02.2031 18:48 GMT+03:00"
                   },
                   "signing-timestamp": "01.02.2021 19:00 GMT+03:00"
               },
               {
                   "description": "Recipient signature",
                   "certificate-owner":
```

(continues on next page)

(continued from previous page)

```
                    {
                        "organization": "LLC "CONCESSION SUPPLY"",
                        "employee": "Peter Parker, Director"
                    },
                    "certificate":
                    {
                        "serial-number": "3610D7230004000503BF",
                        "validity": "с 04.05.2006 16:07 по 31.12.2021 23:59 GMT+03:00"
                    },
                    "signing-timestamp": "02.02.2021 13:00 GMT+03:00"
                }
            ],
            "intermediate-page-footer-info":
            {
                "lines": [
                    "Передан через XXX 01.02.2003 15:20 GMT+03:00",
                    "8f109134-403b-4de5-aa09-6d10462ec071"
                ],
                "logo-asset-name": "golang-gopher.png",
                "print-page-number": true
            },
            "logo-asset-name": "golang-gopher.png"
        },
        "draw-options":
        {
            "font-color": "#0054BE",
            "font-size": 8,
            "large-font-size": 14,
            "table-border-color": "#0054BE"
        }
    },
    "options":
    {
        "enable-debug-doc-save": true,
        "enable-debug-pdf-log": true
    }
}
```

Example requests are located in the `examples/pdf/print_form` directory in the *examples archive*.

Service call example

```
curl --request POST --data-binary "@templates/examples/pdf/print_form/request_
↪example1.json" http://localhost:8087/print_form_pdf_json
```

Response structure

The service response contains an object in JSON format that includes:

1. Error description (error code, error message). In case of successful service response the value `null` is returned.
2. Result (base64 encoded PDF file of the printed form of the document).

Response format

```
{
   error: {
       code
       message
   }
   result: "[base64 encoded pdf file]"
}
```

Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

Service directories

The images to be used as the logo are located in the `assets/images/print_forms` directory located in the service application directory.

7.5 Service for converting XLSX documents to JSON, XML, CSV

7.5.1 General description of the service

The service converts XLSX document to JSON, XML, CSV formats by HTTP-request in JSON format or by HTTP-request in multipart/form-data format.

In response, the service gives the content of the input document presented in one of the target formats.

Note: The conversion service is available in the report server starting with version 3.3.1.1

Service Description

Service name	Service for converting XLSX documents to JSON, XML, CSV
Path to service	For multipart/form-data requests • [host]:[port]/xlsx_convert For json requests • [host]:[port]/xlsx_convert_json
Method	POST
Parameters	The request body must contain an object in JSON format or objects in multipart/form-data format. One can read more about the structure of the request body in the "Request body structure" subsection. In response, the service returns the document content in the target format.
Purpose	The service is designed to convert XLSX document to the following formats: JSON, XML, CSV

Request body structure

In general, JSON or multipart/form-data requests include:

Document descriptor. Request options. Example for JSON request:

```
    {
"document":
{
    "name": "",
    "data": ""
},
"options": {…}
    }
```

Document descriptor

The document descriptor for the JSON format has fields:

- `name` - string. Name is a document identifier. It is used for references to the input parameter in error messages and diagnostic log.
- `data` - string. BASE64 encoded XLSX-document.

The document descriptor for the multipart/form-data format consists of the parameter **xlsx="@[XLSX-document path]**

For example:

```
curl --request POST -F xlsx="@convert_example.xlsx" -F output-format="xml" http://
↪localhost:8087/xlsx_convert
```

Request options

The request can contain the following query options:

- `output-format` - string. Target format of conversion. Supported values: "JSON", "XML", "CSV". The default value is `JSON`.
- `process-sheets-numbers` - an array of integers. It defines sequential page numbers for conversion from the input document (For example: [1,2,3] - for processing the first three pages). The numbering starts with "1". Default value - `all pages` for JSON and XML, `first page` for CSV.
- `process-from-start-of-sheet` - Boolean value which specifies to start processing the document from the very first cell (A1) regardless of whether it contains data or not. Otherwise, processing will be performed from the first cell with data. The default value is `false`.
- `no-extend-line` - Boolean value. This option is responsible for extending each line of the source document to the maximum line length in the document in order to obtain a two-dimensional data array (this is the default behavior of the converter). If true, the option specifies not to extend lines and the output will be an array of data arrays. The default value is `false`.
- `process-start-cell` - string. This option allows one to specify the start cell for processing (in cell naming format, e.g. "E5"), i.e. sets the left upper boundary of processing by row-column, which allows one to convert only the data range of interest. The default value is the `first cell with data` (will be defined automatically).
- `process-end-cell` - string. This option allows one to specify the final cell for processing (in cell naming format, e.g. "G8"), i.e. it specifies the right lower boundary of processing by row-column, which allows one to convert only the data range of interest. The default value is the `last cell with data` (will be defined automatically).
- `encoding` - string. This option allows you to specify the target encoding of the processing result. Supported values: WIN1250, WIN1251, WIN1252, WIN1253, WIN1254, WIN1255, WIN1256, WIN1257, WIN874, WIN866, KOI8R, ISO_8859_5. The default value is `UTF8`.
- `enable-debug-report-save` - Boolean value which specifies whether to create a copy of the request in the local `reports_debug` directory of the service. The default value is `false`.

For the requests in JSON format, options are passed in the `options` object

Example:

```
"options":
{
  "output-format": "json",
  "no-extend-line": false,
  "process-sheets-numbers":[1,2],
  "process-from-start-of-sheet": false,
  "process-start-cell":"C2",
  "process-end-cell":"G5",
  "enable-debug-report-save": true
}
```

For *multipart/form-data requests*, each option is passed as a separate parameter, all parameters are specified as strings.

Example:

```
curl --request POST -F "xlsx=@convert_example.xlsx" -F output-format="json" -F no-
↪extend-line="false" -F process-sheets-numbers="1,2" -F process-from-start-of-sheet=
↪"false"
-F process-start-cell="C2" -F process-end-cell="G5" http://localhost:8087/xlsx_convert
```

Example queries can be found in the `examples/xslx` directory in the *examples archive*.

Service call example

For requests in JSON format:

```
curl --request POST --data-binary "@convert_example.json" http://localhost:8087/xlsx_
↪convert_json > convert_example_from_json.json
```

For multipart/form-data requests:

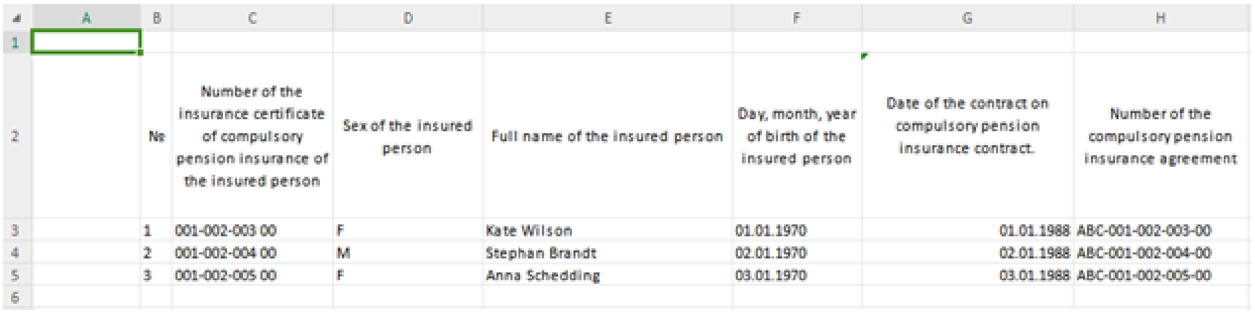
```
curl --request POST -F "xlsx=@convert_example.xlsx" -F output-format="xml" http://
↪localhost:8087/xlsx_convert > convert_example1.xml

curl --request POST -F "xlsx=@convert_example.xlsx" -F output-format="csv" -F process-
↪from-start-of-sheet="false" -F process-start-cell="C2" -F process-end-cell="G5"
http://localhost:8087/xlsx_convert > convert_example2.csv

curl --request POST -F "xlsx=@convert_example.xlsx" -F output-format="json" -F no-
↪extend-line="true" -F process-sheets-numbers="1,2" -F process-from-start-of-sheet=
↪"false"
http://localhost:8087/xlsx_convert > convert_example2.json
```

Response structure

The service response contains a text stream of data in the target encoding.

Conversion example

As an example, let us review the conversion of the range "D2": "G4" of the second page of the document `convert_example.xlsx` from the *examples archive*:



	A	B	C	D	E	F	G	H
1								
2		№	Number of the insurance certificate of compulsory pension insurance of the insured person	Sex of the insured person	Full name of the insured person	Day, month, year of birth of the insured person	Date of the contract on compulsory pension insurance contract.	Number of the compulsory pension insurance agreement
3		1	001-002-003 00	F	Kate Wilson	01.01.1970	01.01.1988	ABC-001-002-003-00
4		2	001-002-004 00	M	Stephan Brandt	02.01.1970	02.01.1988	ABC-001-002-004-00
5		3	001-002-005 00	F	Anna Schedding	03.01.1970	03.01.1988	ABC-001-002-005-00
6								

The request in multipart/form-data format:

```
curl --request POST -F "xlsx=@convert_example.xlsx" -F output-format="csv" -F process-
↪sheets-numbers="2" -F process-start-cell="D2" -F process-end-cell="G4"
http://localhost:8087/xlsx_convert
```

Conversion result:

```
Sex of the insured person; Full name of the insured person, Day, month, year of birth␣
↪of the insured person; Date of the contract on compulsory pension insurance
F;Kate Wilson;01.01.1970;01.01.1988
M;Stephan Brandt;02.01.1970;02.01.1988
```

CHAPTER

EIGHT

EXAMPLES

8.1 Examples archive

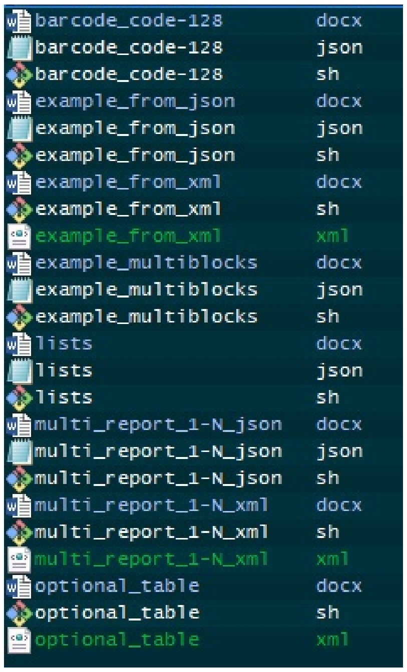
In this section, one can see examples of requests and templates for the service.

`examples archive` should be unpacked to the **templates** directory in the root of the service. The unpacked archive contains the following directories:

- – `pdf` - contains PDF templates of pages used in the *PDF report generator*
- – `examples` - represents the following directory tree:

```
├───docx
├───pdf
│   ├───embedded-report-generator
│   ├───pdf_merge
│   ├───print_form
│   └───report-generator
│       └───pdf_report_gen
└───xlsx
```

The catalogs **docx,xlsx** contain a set of requests (in json/xml formats), - template documents (.docx and *.xlsx) and scripts (.sh) for receiving reports



```
barcode_code-128        docx
barcode_code-128        json
barcode_code-128        sh
example_from_json       docx
example_from_json       json
example_from_json       sh
example_from_xml        docx
example_from_xml        sh
example_from_xml        xml
example_multiblocks     docx
example_multiblocks     json
example_multiblocks     sh
lists                   docx
lists                   json
lists                   sh
multi_report_1-N_json   docx
multi_report_1-N_json   json
multi_report_1-N_json   sh
multi_report_1-N_xml    docx
multi_report_1-N_xml    sh
multi_report_1-N_xml    xml
optional_table          docx
optional_table          sh
optional_table          xml
```

The **pdf** catalog contains the following catalogs:

- `embedded-report-generator` -requests (.json) and scripts (*.sh) for generating reports in format .pdf via *PDF report generator*.
- `pdf_merge` -requests (.json) and scripts (*.sh) to *merge PDF documents*.
- `print_form` -requests (.json) and scripts (*.sh) for *print form generation*.
- `report-generator` – requests (.json) and scripts (*.sh) for generating reports via *PDF report generator* with an example of generator as part of the service. The `pdf_report_gen` directory with examples of generators should be located in the root directory of the service.

8.2 8.2 Generating a report from the examples archive

Let us review generating a report from a docx catalog based on a lists.json request

```
{
    "template":
    {
        "uri": "local",
        "id": "examples/docx/lists"
    },
    "input-data":
    {
        "CONDITIONAL_TAG_TRUE": "true",
        "CONDITIONAL_TAG_FALSE": "false",
        "BULLET_LIST": ["bullet item 1", "bullet item 2", "bullet item 3"],
        "NUMBERED_LIST": ["numbered item 1", "numbered item 2", "numbered item 3"],
        "EMPTY_BULLET_LIST": [],
        "EMPTY_NUMBERED_LIST": []

    },
```

(continues on next page)

(continued from previous page)

```
    "options":
    {
        "enable-debug-report-save": false,
      "enable-binary-output": true,
        "formatting":
        {
            "tables":
            {
                "enable-cells-auto-merge": true
            }
        }
    }
}
```

As a result of running the lists.sh script

```
curl --request POST --data-binary "@lists.json" http://localhost:8087/word_report_
↪json --output "lists_report.docx"
```

the report file lists_report.docx was written in the templates/examples/docx directory

Note: scripts may need to be assigned execution permission. This can be done with the following command: chmod +x *.sh

CHAPTER

NINE

MY FIRST REPORT

9.1 General Description

This subsection will cover the process of creating a simple report using the report server. Before starting work, One needs to make sure that the installation of the report server was successful.

9.2 First template

To generate a report, it is necessary to create a document template in DOCX or XLSX format using MS Word/Excel, P7-Office, LibreOffice Writer/Calc, etc.

Document templates can contain tags that will be replaced with input data from the request. The tag is specified in square brackets. Example: [debt]. To learn more about creating templates in DOCX and XLSX format, see 'Service for generating reports from a template document'.

For the first report, let's create a simple template in DOCX format that contains a few tags: first_report_template.docx. Examples of more complex templates can be found in the examples archive.

The created template is placed on the server in the `templates` directory located in the service application directory.

Note: PDF page `templates` are stored in the `templates/pdf` directory located in the service application directory.

9.3 First request

The next step is to form an HTTP request.

The request for the first report in JSON format will look like this: first_report.json. The template document created earlier is specified in the id attribute of the template element. Examples of more complex requests can be found in the examples archive.

Note. XML format can also be used for requests in DOCX or XLSX format.

To get a report file, use the service of report generation from a template document.

To get the first report based on the created DOCX template, we use the URI http://localhost:8087/word_report_json.

9.3.1 Description

Service name	Print form generation service
Путь к сервису	[host]:[port]/print_form_pdf_json
Method	POST

Parameters	The body of the request must contain a JSON object:
	```
	{
	    "template":
	    {
	        "uri": "local",
	        "id": "first_report_template"
	    },
	    "input-data":
	    {
	        "ORGANIZATION": "Example JSC",
	        "DATE": 01/01/2023",
	        "EMP": "Peter Parker"
	    },
	    "options":
	    {
	        "formatting": {
	            "tables": {
	                "enable-cells-auto-merge":␣
	↪true
	            }
	        }
	    }
	}
	```
	 A `template` object has attributes (fields):
	• `uri` - string specifying the location of the template document file depending on how the `id` parameter is interpreted. Supported value: `local`.
	• `id` - string which is a template identifier. It specifies the path to the template document file relative to the templates service directory. It is also used to write a report file in debug mode and to identify a request in the log.

continues on next page

Table 1 – continued from previous page

	`input-data` - `input` data to be inserted into the template.
	An `input-data` object can contain:
	• tags of simple string data;
	• table descriptor tags;
	• list descriptor tags;
	• image tags;
	• block tags.
	`options` - request options.
	One can read more about JSON structure in the section Service for generating reports from a template document'. The service outputs the report document file in DOCX format encoded in BASE64. If it is necessary to receive the result without BASE64 encoding, the enable-binary-output flag should be set to true in the request options.
	`{`
	`"template":`
	`{`
	`"uri": "local",`
	`"id": "first_report_template"`
	`},`
	`"input-data":`
	`{`
	`"ORGANIZATION": "Example JSC",`
	`"DATE": "01/01/2023",`
	`"EMP": "Peter Parker"`
	`},`
	`"options":`
	`{`
	`"enable-binary-output": true,`
	`"formatting": {`
	`"tables": {`
	`"enable-cells-auto-merge": true`
	`}`
	`}`
	`}`
	`}`

Purpose	The service is designed to generate a printed form of the document in PDF format, including footers with brief information about the document and a signature table on the last page.

9.3.2 Example request

```
curl -X POST \
'http://localhost:8087/word_report_json' \
-H 'Content-Type: application/json' \
-d '{
   "template":
   {
       "uri": "local",
       "id": "first_report_template"
   },
   "input-data":
```

(continues on next page)

(continued from previous page)

```
    {
        "ORGANIZATION": "АО «Пример»",
        "DATE": "01.01.2023",
        "EMP": "Иванов Иван Иванович"
    },
    "options":
    {
        "formatting": {
            "tables": {
                "enable-cells-auto-merge": true
            }
        }
    }
}'
```

9.3.3 Example of a request to save the result to a file without base64 encoding

```
curl -X POST \
'http://localhost:8087/word_report_json' \
-H 'Content-Type: application/json' \
-d '{
    "template":
    {
        "uri": "local",
        "id": "first_report_template"
    },
    "input-data":
    {
        "ORGANIZATION": "Example JSC",
        "DATE": "01/01/2023",
        "EMP": "Peter Parker"
    },
    "options":
    {
        "enable-binary-output": true,
        "formatting": {
            "tables": {
                "enable-cells-auto-merge": true
            }
        }
    }
}'
-o first_report.docx
```

9.4 Receiving the first response

9.4.1 The format of the returned response

```
{
   error: {
       code:
       message:
   }
   result: "[base64 encoded docx/pdf file]"
}
```

9.4.2 Response example

```
{
   "error": null,
   "result": "UEsDBBQACAAIAPdKo...JwAAAAA="
}
```

9.5 Completion

As a result, we get a base64-encoded DOCX file of the first report document or an error message.