

Natural

Natural API Server

Version 9.3.4

October 2026

This document applies to Natural Version 9.3.4 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1992-2026 Software GmbH, Darmstadt, Germany and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software GmbH product names are either trademarks or registered trademarks of Software GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software GmbH and/or its subsidiaries is located at <https://softwareag.com/licenses>.

Use of this software is subject to adherence to Software GmbH's licensing conditions and terms. These terms are part of the product documentation, located at <https://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software GmbH Products / Copyright and Trademark Notices of Software GmbH Products". These documents are part of the product documentation, located at <https://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software GmbH.

Document ID: NATUX-NATNPI-934-20261013

Table of Contents

Preface	v
1 About this Documentation	1
Document Conventions	2
Online Information and Support	2
Data Protection	3
I Overview	5
2 Overview	7
What is Natural API Server	8
Architecture	8
About Natural API Server	9
II Natural Subprogram Server on Linux	11
3 Installation	13
Prerequisites	14
Installing the Server	14
Starting the Server	15
Verifying the Installation	17
4 Configuration	19
How Configuration is Applied	20
Example of Configuration File	20
Configuration Reference	21
Server and Connections	23
Workers and Throughput	24
Natural Environment	26
5 Operation	29
How Natsubsrv Works	30
Architecture Overview	30
Natural Session Mode	31
Session Isolation Between Requests	32
Stopping and Restarting	33
6 Monitoring	35
Logging	36
Metrics	37
Traces	38
Retrieving Health and Telemetry Data	38
7 Security	41
Natural Security	42
8 Troubleshooting	47
Reporting a Problem	48
Common Problems	48
Crash Reports	49
III Natural OpenAPI Server on Tomcat	51
9 Installation	53
Installing the Natural OpenAPI Server	54

Installing the Natural OpenAPI Server as systemd service	54
10 Configuration	55
General Configuration	56
Natural Error to HTTP Mapping	57
11 Operation	59
Starting and Stopping the Server	60
Deploying Web Application with Tomcat	60
Deploying OpenAPI Resources	61
12 Monitoring	63
13 Security	65
Secure Communication between Natural OpenAPI Server and Subprogram Server	66
Secure Communication between Client and Natural OpenAPI Server	66
Authentication	67
IV Developer Overview	69
14 Installation of the NaturalONE OpenAPI Plugin	71
15 Configuration of the NaturalONE OpenAPI Plugin	73
16 Generate and Deploy Services	77
Generating OpenAPI Resources	78
Viewing OpenAPI Resources via Editor	79
17 Special Considerations	83
Developing Natural Subprograms for Natsubsrv	84

Preface

This documentation describes the Natural API Server

Overview
Natural Subprogram Server on Linux
Natural OpenAPI Server on Tomcat
Developer Overview

1

About this Documentation

■ Document Conventions	2
■ Online Information and Support	2
■ Data Protection	3

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Monospace font	Identifies service names and locations in the format <i>folder.subfolder.service</i> , APIs, Java classes, methods, properties.
<i>Italic</i>	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
Monospace font	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the symbol.
[]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at <https://documentation.softwareag.com>.

Product Training

You can find helpful product training material on our Learning Portal at <https://learn.software-ag.com>.

Tech Community

You can collaborate with Software GmbH experts on our Tech Community website at <https://tech-community.softwareag.com>. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software GmbH news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at <https://github.com/softwareag> and <https://hub.docker.com/publishers/softwareag> and discover additional Software GmbH resources.

Product Support

Support for Software GmbH products is provided to licensed customers via our Empower Portal at <https://empower.softwareag.com>. Many services on this portal require that you have an account. If you do not yet have one, you can request it at <https://empower.softwareag.com/register>. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

I Overview

2 Overview

■ What is Natural API Server	8
■ Architecture	8
■ About Natural API Server	9

This document covers the following topics:

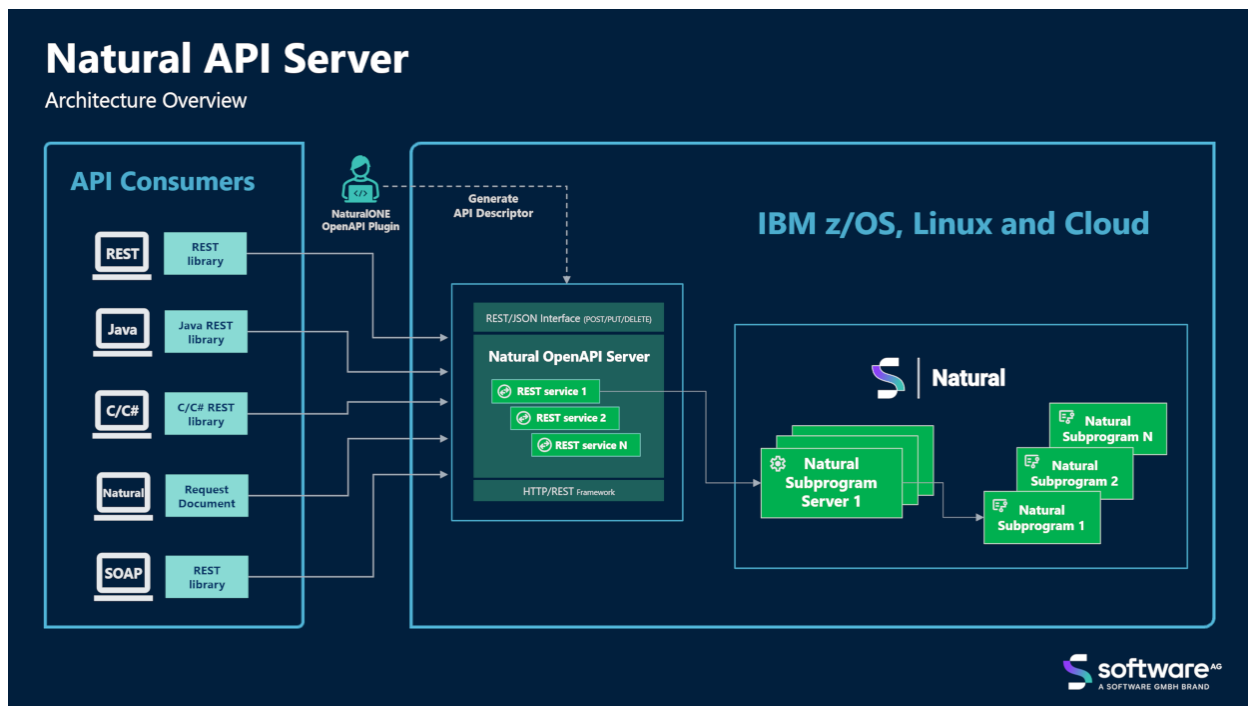
What is Natural API Server

The Natural API Server provides access to Natural subprograms through REST API calls based on OpenAPI standard. This allows external applications to access and reuse Natural business logic without rewriting existing Natural subprograms. The OpenAPI descriptions can be generated automatically.

Architecture

The Natural API Server consists of the following components:

- Natural Subprogram Server for z/OS or Linux and Cloud.
- Natural OpenAPI Server as interface between REST API clients and the Natural Subprogram Server. This component runs standalone as Embedded Tomcat® or is deployed in a Java Application Servers like Apache Tomcat® or Liberty®.
- NaturalONE OpenAPI Plugin to generate and deploy Natural OpenAPI services.



About Natural API Server

The Natural API Server is available as an installed component of your Natural for z/OS or Linux environment, or as an Eclipse Plugin for NaturalONE.

II Natural Subprogram Server on Linux

Natsubsrv is a backend server that executes Natural subprograms via an internal RPC protocol. It is accessed through a middleware component (for example, the Natural OpenAPI Server), which exposes a REST API and forwards requests to Natsubsrv. This allows your Natural business logic to be integrated into modern applications.



Note: Natsubsrv does not provide a REST API for client applications. Direct access from client applications is not supported.

This document covers the following topics:

Installation
Configuration
Operation
Monitoring
Security
Troubleshooting

3 Installation

■ Prerequisites	14
■ Installing the Server	14
■ Starting the Server	15
■ Verifying the Installation	17

This document covers the following topics:

Prerequisites

- Linux (64-bit).
- A working Natural environment. Natsubsv is not a standalone program, it runs Natural subprograms inside a Natural session and therefore needs a complete Natural installation — nucleus, parameter module, and the libraries holding your subprograms. The Software AG Installer installs Natural together with the server, so you do not install it separately; the two belong together and are versioned together. Before starting the server, make sure the Natural environment itself is usable, for example by running one of your subprograms in a normal Natural session.
- A valid Natural API Server license (license file *npi11.xml*). The license is validated at startup; without a valid license, the server logs an error and terminates. If the license expires within the next 90 days, a warning with the remaining days is written to the log.

Installing the Server

Like every other Natural component, Natsubsv is installed with the Software AG Installer. There is no separate archive to unpack.

1. Start the Software AG Installer.
2. Select the **Natural API Server** product.
3. Complete the installation. The Natural environment required by the server is installed along with it.

The installation provides:

File	Location	Used For
npisrv (symbolic link)	<installdir>/Natural/bin	Starting the server ; resolves to the matching binary.
natsubsv, natsubsvsec	<installdir>/Natural/bin	Server binaries without and with Natural Security support.
natenv	<installdir>/Natural/INSTALL	Natural environment, sourced in service operation .
npienv	<installdir>/Natural/INSTALL	Server settings for service operation .
sys/npisrv_sys	<installdir>/Natural/INSTALL	Start script run by the service .
NaturalSubprogramServer, NaturalSubprogramServerNSC	<installdir>/Natural/docker	Container use cases , each with a <i>readme.txt</i> .

Files ending in `.tpl` next to `natenv` and `npieven` are the installer's templates, not the files in use.

Both binaries are always installed. `npisrv` is a symbolic link to the one that matches your installation: `natsubsrvsec` if Natural Security is installed, otherwise `natsubsrv`. Start `npisrv` rather than a binary — the link always resolves to the right one. For more information, see [Natural Security](#).

Starting the Server

The server can be started manually, run as a service, or run as a container. All three use the same executable and the same configuration; they differ only in who starts and supervises the process.

By default, Natsubsrv listens on TCP port 2600. See also [Server Port \(port\)](#).

- [Starting Manually](#)
- [Running as a Service](#)
- [Running as a Container](#)

Starting Manually

Create a JSON configuration file and reference it at startup. Run this in a shell that already has the Natural environment set up.

```
<installdir>/Natural/bin/npisrv config=/path/to/natsubsrv.json
```

Use this for a first check after the installation and for tests. For more information on the contents of the file, see [Configuration](#)

The link can also be bypassed and the binary addressed directly, for example when you want to be certain which variant runs:

```
<installdir>/Natural/bin/natsubsrv config=/path/to/natsubsrv.json
```

Both binaries are always installed, so either variant can be started deliberately. They take the same arguments and configuration, differ only in Natural Security support. For more information, see the [Natural Security](#).

Always pass `config=` with an absolute path. If the argument is omitted, the server looks for a file named `natsubsrv.json` in the current working directory. If none is found there, the server does not fail — it starts with defaults and environment variables only and writes a corresponding message to the log. A configuration that is silently ignored is hard to detect, so use this fallback only when you intentionally want the server to start without the configuration.. A path given with `config=` that does not exist terminates the server with an error.

Running as a Service

The installer registers a system service and enables it so that the server starts at boot and is supervised by the operating system. This is the recommended form of operation for production. Use `systemctl disable` if you do not want the automatic start.

The service name is `sag<id>npsrv`, where `<id>` is the Software AG Installer ID of your installation directory. The installer numbers installation directories consecutively, starting at 1. All daemons of one installation directory share the same ID — on a machine with several Natural installations, this indicates the directory to which the daemon belongs. The other Natural services use the same scheme. Determine the name once:

```
systemctl list-unit-files 'sag*npsrv*'
```

The examples below use `sag1npsrv`:

```
systemctl status sag1npsrv      # check state
sudo systemctl start sag1npsrv  # start
sudo systemctl restart sag1npsrv # apply configuration changes
sudo systemctl stop sag1npsrv   # stop
journalctl -u sag1npsrv -f      # follow the log
```

The service runs the start script `<installdir>/Natural/INSTALL/sys/npsrv_sys`. The script sources `natenv` for the Natural environment and `npienv` for the server settings, both from `<installdir>/Natural/INSTALL`, and then starts `<installdir>/Natural/bin/npsrv`. Server settings in service operation therefore come from `npienv`: change them there and restart the service.

The server writes its log to standard output, so under a service the log is available through `journalctl`. Enabling `syslog` in addition is only necessary if you collect logs centrally.

Running as a Container

Container use cases are delivered under `<installdir>/Natural/docker`, one directory per variant:

- `NaturalSubprogramServer` — without Natural Security
- `NaturalSubprogramServerNSC` — with Natural Security

Each directory contains a *readme.txt* describing how the container is built and used. Follow that description. In addition:

- The image must contain a complete Natural environment, including the parameter module and the libraries holding your subprograms.
- Passing the configuration through `NATSUBSRV_*` environment variables is usually more practical in containers than mounting a JSON file. See [How Configuration is Applied](#).
- Publish the [server port](#) so the middleware can reach the container.

- Mount the [application data directory](#) as a volume if you want runtime data and crash reports to survive the container.
- The delivered images already start Natural with `ETID=$$`, so every worker gets its own Adabas ETID. Setting `NATSUBSRV_NATARG` yourself replaces that value as a whole. See [Adabas User ID \(ETID\)](#).
- Stopping the container sends `SIGTERM`, which the server handles as an orderly shutdown. See [Stopping and Restarting](#).

Verifying the Installation

At startup, Natsubsrsv writes the following to the log:

- The product version and build information. Keep these values at hand as they are required when [reporting a problem](#).
- A one-line summary of the effective configuration. At `DEBUG` level, each parameter is instead listed on its own line together with its origin — configuration file, environment variable, or default. That is the quickest way to check whether your settings were actually picked up.
- A message once the server listens on its port.

If a configuration value is invalid or a required directory cannot be created, the server terminates during startup with a configuration error naming the affected parameter.

- [Checking the Server with a Ping Request](#)
- [Connecting the Middleware](#)

Checking the Server with a Ping Request

Natsubsrsv answers a ping request without requiring authentication and without executing a Natural subprogram. This is the quickest way to check whether the server is reachable — and when a problem occurs, to determine whether Natsubsrsv or the middleware is at fault.

The request is a fixed 50-character text, so no client program is needed. Replace `<host>` and the port as required.

Without TLS, using Bash only:

[illegible]

Without TLS, if nc is available:

[illegible]

With TLS:

```
printf 'CALLN110PI0000000000000000000000000000000000000000' | openssl s_client -quiet -connect <host>:2600 2>/dev/null; echo
```

The server replies with a 50-character header followed by a JSON object:

```
{
  "schemaVersion": "1.0.0",
  "status": "healthy",
  "ctime": 1755600000,
  "machineName": "myhost",
  "port": 2600,
  "versionNumber": "1.2.3",
  "buildNumber": "123",
  "commitId": "abcdef123456",
  "buildDate": "2025-11-25 14:30:00",
  "buildChannel": "..."
}
```

versionNumber and buildNumber are the values to quote when *reporting a problem*.

Notes:

1. The command returns after a short delay (by default about 250 ms) because the server waits for the client to close the connection first. See [Connection Close Timeout](#).
2. Use the matching command for your setup. A plain TCP command against a TLS-enabled server, or the reverse, produces no usable reply.
3. If the TLS command prints nothing at all, the handshake failed. Repeat it without `2>/dev/null` to see the reason.
4. A reply confirms that the server and at least one worker are running and responsive. It does not execute a Natural subprogram, so it does not prove that a particular library or subprogram can be called.
5. Ping requires no authentication and reveals the host name and product version. This is one reason not to expose the server port to untrusted networks. See [Transport Security \(TLS/SSL\)](#).

Connecting the Middleware

Configure the middleware (for example, the Natural OpenAPI Server) with the host and port of Natsubsvr and issue a request from your application.

4 Configuration

■ How Configuration is Applied	20
■ Example of Configuration File	20
■ Configuration Reference	21
■ Server and Connections	23
■ Workers and Throughput	24
■ Natural Environment	26

This document covers the following topics:

How Configuration is Applied

Configuration is provided in a JSON file and can be overridden by environment variables:

1. Defaults built into the server.
2. Values from the JSON configuration file.
3. Values from environment variables — these have the highest precedence.

The mechanism that applies depends on how the server is started:

- As a *service* - The settings come from `<installdir>/Natural/INSTALL/npienv`, which sets the `NATSUBSRV_*` variables. Change them there and restart the service. Commented-out entries in that file show the optional settings.
- Started *manually* - Use a JSON file and reference it with `config=`.
- As a *container* - As described in the *readme.txt* of the container use case.

Keep the two apart. Because environment variables take precedence, a `NATSUBSRV_*` variable that is set in your shell overrides the corresponding value in your JSON file.

All parameters are optional; each one falls back to its default. Invalid values are not silently corrected: the server reports a configuration error and does not start. Changing the configuration requires a restart of the server.

Example of Configuration File

```
{
  "port": 2600,
  "appDataDir": "/srv/natsubsrv/appdata",
  "numberOfWorkers": 4,
  "workerQueueSize": 10,
  "socketTimeout": 3000,
  "closeDrainTimeout": 250,
  "maxMessageSize": 1024,
  "naturalCommandLine": "parm=NATPARM",
  "loggingMode": "INFO",
  "enableSyslog": true,
  "metrics": false,
  "metricsInterval": 15,
  "tracesStoredPerWorker": 1000,
  "natRpc39IntervalInSeconds": 0,
```

```

"tlsKey": "/path/to/tls.key",
"tlsCertificate": "/path/to/tls.crt",
"pamService": "natssubsvr",
"keepSecuritySession": false,
"backoutTransactionPerRequest": true,
"closeWorkFilesPerRequest": true,
"nscStartupLogonRetryCount": 0,
"nscStartupLogonRetryWaitSeconds": 5
}

```

Configuration Reference

The following parameters are available:

JSON Parameter	Environment Variable	Values / Default
port	NATSUBSRV_PORT	TCP port / 2600
appDataDir	NATSUBSRV_APP_DATA_DIR	absolute path / \$HOME/.natssubsvr/<
unixDomainSocketPath	NATSUBSRV_UDSPATH	absolute path / <appDataDir>/run/n
numberOfWorkers	NATSUBSRV_WORKERS	≥ 1 / 1
workerQueueSize	NATSUBSRV_QUEUE_SIZE	≥ 1 / 10
socketTimeout	NATSUBSRV_TIMEOUT	ms / 3000
closeDrainTimeout	NATSUBSRV_CLOSE_DRAIN_TIMEOUT	ms, min. 10 / 250
maxMessageSize	NATSUBSRV_MAX_MESSAGE_SIZE	1–10240 MB / 1024
naturalCommandLine	NATSUBSRV_NATARG	text / empty
natRpc39IntervalInSeconds	NATSUBSRV_NATRPC39INTERVAL	seconds, 0 = off / 0

JSON Parameter	Environment Variable	Values / Default
loggingMode	NATSUBSRV_LOG	See Logging / INFO
enableSyslog	NATSUBSRV_SYSLOG	true/false / false
metrics	NATSUBSRV_METRICS	true/false / false
metricsInterval	NATSUBSRV_METRICS_INTERVAL	seconds / 15
tracesStoredPerWorker	NATSUBSRV_TRACES_PER_WORKER	≥ 1 / 1000
tlsKey	NATSUBSRV_TLSKEY	path / off
tlsCertificate	NATSUBSRV_TLSCERT	path / off
pamService	NATSUBSRV_PAM	service name / off
keepSecuritySession	NATSUBSRV_KEEP_SECURITY_SESSION	true/false / false
backoutTransactionPerRequest	NATSUBSRV_BACKOUT_TRANSACTION_PER_REQUEST	true/false / true
closeWorkFilesPerRequest	NATSUBSRV_CLOSE_WORK_FILES_PER_REQUEST	true/false / true
nscStartupLogonRetryCount	NATSUBSRV_NSC_STARTUP_LOGON_RETRY_COUNT	0 = off, or 2–10 / 0
nscStartupLogonRetryWaitSeconds	NATSUBSRV_NSC_STARTUP_LOGON_RETRY_WAIT	1–60 seconds / 5

Parameters where default is off are inactive until you configure them. To switch such a parameter off explicitly, set it to an empty string, `false`, `0`, or `OFF`.

Server and Connections

- [Server Port \(port\)](#)
- [Request Timeout \(socketTimeout\)](#)
- [Connection Close Timeout \(closeDrainTimeout\)](#)
- [Maximum Message Size \(maxMessageSize\)](#)

Server Port (port)

TCP port on which Natsubsrvt accepts requests from the middleware.

- **Values:** a free TCP port — **Default:** 2600
- **Environment variable:** NATSUBSRV_PORT
- **Choosing a value:** Leave the default unchanged unless the port is already in use. To run several instances on the same host, give each instance its own port and configure the middleware accordingly. The port is also part of the default `appDataDir`, so instances on different ports keep their runtime data separate.

Request Timeout (socketTimeout)

Limits how long the server waits for request data from the client and for the response to be accepted.

- **Values:** milliseconds — **Default:** 3000
- **Environment variable:** NATSUBSRV_TIMEOUT
- **Choosing a value:** The default is sufficient for clients in the same network. Increase it if the middleware connects over a high-latency network and requests are aborted although the subprogram itself completes. Note that this timeout covers the network transfer, not the runtime of the subprogram.

Connection Close Timeout (closeDrainTimeout)

Limits how long the server waits for the client to close the connection after the response has been sent.

- **Values:** milliseconds, minimum 10 — **Default:** 250
- **Environment variable:** NATSUBSRV_CLOSE_DRAIN_TIMEOUT
- **Choosing a value:** 250 ms suits most installations. Use 50–100 ms when the middleware runs in the same data center and the connection rate is high, so workers become available again faster.

Increase the value only for WAN connections when the log repeatedly reports a timeout while closing the connection.

Allowing the client to close first ensures that it has received the complete response. Values below 50 ms are accepted but produce a startup warning because remote or TLS clients may not complete the close in time. If the timeout expires, the server logs a warning and closes the connection itself — the response has already been sent, so no data is lost.

Maximum Message Size (`maxMessageSize`)

Limits the size of a single request that the server accepts.

- **Values:** 1–10240 — **Default:** 1024
- **Environment variable:** NATSUBSRV_MAX_MESSAGE_SIZE
- **Choosing a value:** Increase the limit if the subprograms exchange large data volumes and requests are rejected as too large. Lower it if your workloads only use small payloads.

The limit protects the server. Each request declares the size of its data buffers and the server allocates memory accordingly. Without an upper bound, a faulty or malicious client could exhaust the server's memory. An oversized request results in an error; the server keeps running.

Workers and Throughput

- [Number of Workers \(`numberOfWorkers`\)](#)
- [Worker Queue Size \(`workerQueueSize`\)](#)
- [Application Data Directory \(`appDataDir`\)](#)
- [Unix Domain Socket Path \(`unixDomainSocketPath`\)](#)

Number of Workers (`numberOfWorkers`)

Number of worker processes that handle requests in parallel. Each worker runs its own Natural session.

- **Values:** ≥ 1 — **Default:** 1
- **Environment variable:** NATSUBSRV_WORKERS
- **Choosing a value:** Base the value on the number of (virtual) CPUs and the workload type — roughly 0.5–1× the number of CPUs for CPU-bound workloads, and up to 3× for I/O-bound workloads such as database access. On an 8-CPU machine, this means about 4 workers for CPU-bound and up to 24 for I/O-bound workloads. Every worker consumes memory for its own Natural session, so increase the value in steps and observe the memory usage.

Worker Queue Size (`workerQueueSize`)

Number of incoming connections that may wait per worker until a worker accepts them.

- **Values:** ≥ 1 — **Default:** 10
- **Environment variable:** NATSUBSRV_QUEUESIZE
- **Choosing a value:** Leave the default unchanged. Increase it only for short load peaks when clients are rejected although the workers are not fully utilized. A larger queue does not increase throughput — it only buffers longer, which increases the response times instead of rejecting requests.

Application Data Directory (`appDataDir`)

Base directory for the runtime data managed by Natsubsrsv.

- **Values:** absolute path — **Default:** `$HOME/.natsubsrsv/<port>`
- **Environment variable:** NATSUBSRV_APP_DATA_DIR
- **Choosing a value:** Leave the default unchanged for test installations. For production installations, configure a dedicated directory that is writable only by the runtime user.

Natsubsrsv manages these subdirectories:

```
<appDataDir>/
  run/      runtime files, for example the socket
  crash/    crash reports
```

The directories are created at startup with restrictive permissions. If you configure `appDataDir` explicitly, its parent directory must already exist — Natsubsrsv creates only the directory itself and its subdirectories, not an arbitrary chain of parent directories. See [File System Security](#).

Unix Domain Socket Path (`unixDomainSocketPath`)

Path to the socket used internally between the server and its workers.

- **Values:** absolute path — **Default:** `<appDataDir>/run/natsubsrsv.sock`
- **Environment variable:** NATSUBSRV_UDSPATH
- **Choosing a value:** Leave the default unchanged. Configure a path of your own only if `appDataDir` is located on a file system that does not support sockets. Keep the path short as it is subject to the length limit for Unix domain sockets (107 characters on Linux).

Natural Environment

- Natural Startup Parameters ([naturalCommandLine](#))
- Adabas User ID (ETID)
- User Area Size (USIZE)
- Periodic SYSTEM/NATRPC39 ([natRpc39IntervallInSeconds](#))

Natural Startup Parameters ([naturalCommandLine](#))

Parameters passed to the Natural session of each worker, for example the parameter module to use.

- **Values:** Natural command line, for example `parm=NATPARM` — **Default:** empty
- **Environment variable:** NATSUBSRV_NATARG
- **Choosing a value:** Use the parameter module of the application that provides the subprograms. All workers start with the same parameters.

Adabas User ID (ETID)

Every worker runs its own Natural session, and every session identifies itself to Adabas with an ETID. If neither the parameter module nor the startup parameters set an ETID, Natural derives it from `*INIT-USER`, which is identical in all workers. They then all open Adabas with the same ETID: the first worker succeeds, the others fail with NAT3048.

With more than one worker, set `ETID=$$`. Natural replaces `$$` with the process ID, so every worker receives an ETID of its own. Set it in the parameter module you use, or in [naturalCommandLine](#), which takes precedence over the parameter module:

```
"naturalCommandLine": "parm=NATPARM ETID=$$"
```

Check the parameter module before changing anything — if it already contains `ETID=$$`, several workers run without further configuration, and a parameter module that works in one environment can hide a missing ETID in another.

Natural substitutes `$$` only when it reads exactly those two characters. Anything else is taken as a literal ETID — an escape sequence that survived into the value as much as a `$$` a shell already replaced with its own process ID. Both put all workers back on one shared ETID without an error message. How to write it depends on where it is set:

Set in	Write
JSON configuration file	"naturalCommandLine": "parm=NATPARM ETID=\$\$ "
Shell or npientv	NATSUBSRV_NATARG='parm=NATPARM ETID=\$\$' — in single quotes, otherwise the shell substitutes its own process ID
Dockerfile	ENV NATSUBSRV_NATARG="parm=NATPARM ETID=\$\$" — \$ is the Dockerfile escape for a literal \$
Docker Compose	NATSUBSRV_NATARG=parm=NATPARM ETID=\$\$\$\$ — Compose reads \$\$ as an escaped \$

The startup log shows the command line Natural actually received — check there that it arrived as \$\$.

User Area Size (USIZE)

For data-intensive subprograms, increase Natural's USIZE (user area size) in your Natural parameters. If USIZE is too small, requests fail with NAT6010: Operation could not be completed. Memory allocation failed.

Periodic SYSTEM/NATRPC39 (natRpc39IntervallInSeconds)

Each worker can run SYSTEM/NATRPC39 periodically to keep its database context alive and avoid NAT3009.

- **Values:** seconds, 0 disables the call — **Default:** 0
- **Environment variable:** NATSUBSRV_NATRPC39INTERVAL
- **Choosing a value:** Enable this only if you see NAT3009 during idle periods. Choose an interval clearly below the timeout configured in your database — for example, half of it.

The call runs independently in each worker and always under the worker's startup user. SYSTEM/NATRPC39 must be available in your Natural environment (for example, copied from SYSRPC to SYSTEM on FUSER). If it is missing, the worker logs a warning and continues.

5 Operation

■ How Natsubsv Works	30
■ Architecture Overview	30
■ Natural Session Mode	31
■ Session Isolation Between Requests	32
■ Stopping and Restarting	33

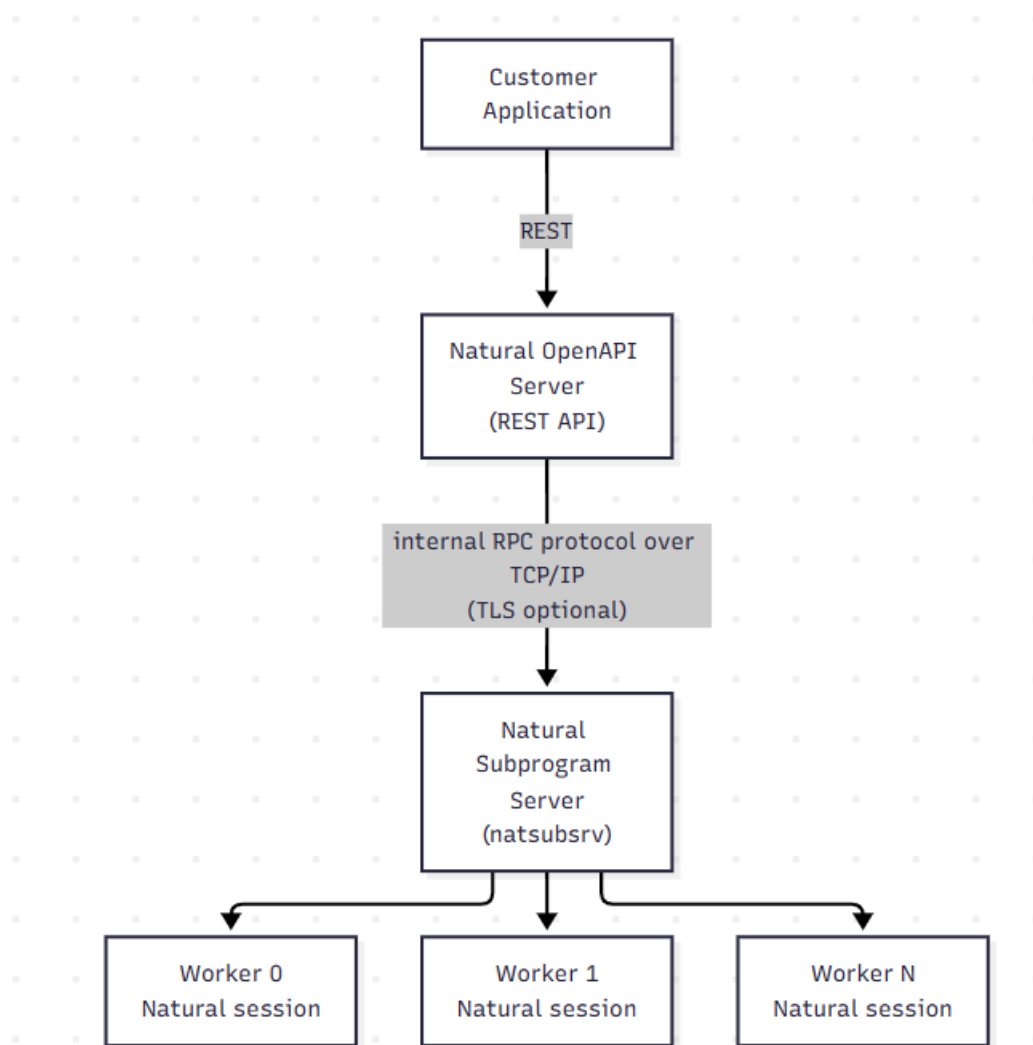
This document covers the following topics:

How Natsubsv Works

- Natsubsv uses a prefork model: a parent process starts the configured number of worker processes and supervises them.
- Each worker runs its own Natural nucleus and runtime environment and handles requests independently.
- All workers listen on the same TCP port, so incoming connections are distributed across them by the operating system.
- A worker reads and validates the request, executes the requested Natural subprogram, and returns the result to the middleware.
- Errors are written to the log and returned to the caller as an error response.
- Natural's limit counters (`MADIO`, `MAXCL`, `MAXERR`, and the `LT` record-read limit) are reset before each request. Usage counted during an earlier request can therefore never cause an unrelated later request to fail.

Architecture Overview

The middleware and Natsubsv can run on the same machine or on different machines; they communicate over TCP/IP, optionally secured with TLS.



Natural Session Mode

The Natural runtime used by Natssrv is initialized as a batch session and runs in server mode. The system variable `*SERVER-TYPE` is `CALLNAT`.

- Subprograms must not expect interactive prompts or terminal I/O — statements such as `INPUT` or `DISPLAY` do not behave as in an interactive session.
- Operations that rely on a terminal may fail or need to be adapted.
- Use a report specification (REP notation) to emit logging and debugging output.
- The report specification `VIDEO` is not supported.

A worker keeps its Natural session for its entire lifetime and serves any number of requests with it, so requests of different end users run one after another in the same session. Write your subprograms accordingly:

- Keep them stateless. A subprogram must not carry data over from one call to the next, and it must not expect data left behind by an earlier call.
- Do not depend on the worker. Which worker executes a request is decided by the operating system, so two consecutive calls of the same user may run in different sessions, and two calls of different users may run in the same one.
- End your own transactions. Complete every unit of work with `END TRANSACTION` or `BACKOUT TRANSACTION` before the subprogram returns. The server does not commit for you. What it does do, unless you switch it off, is back out an open transaction before the next request starts (see [Session Isolation Between Requests](#)). Rely on that as a safety net, not as your transaction logic: a unit of work that is never committed is lost either way.

Session Isolation Between Requests

Because a worker reuses one Natural session for requests of different end users, the server resets that session at the start of every request: application-independent variables, global data areas, session parameters, the library search order and the retained sets are put back to a defined state, so no request can see what its predecessor left behind.

If your subprograms already clean up after themselves, you can disable the two parts of the reset to avoid unnecessary overhead.

- [Transaction Backout](#) (`backoutTransactionPerRequest`)
- [Work File Close](#) (`closeWorkFilesPerRequest`)

Transaction Backout (`backoutTransactionPerRequest`)

Backs out the open transaction of every connected database before the request runs.

- **Values:** `true/false` — **Default:** `true`
- **Environment variable:** `NATSUBSRV_BACKOUT_TRANSACTION_PER_REQUEST`
- **Choosing a value:** Leave it on unless you have measured that it costs you. Switch it off only where every subprogram reliably ends its own unit of work.

A Natural `LOGON` issues neither `END TRANSACTION` nor `BACKOUT TRANSACTION`. Without this option, a transaction that one request leaves open remains open when the next request starts. An `END TRANSACTION` in that next request then commits the previous end user's uncommitted work. The backout covers every database type in one step, Adabas and SQL databases alike, including the branches of a distributed transaction.

It only ends the transaction. Database connections stay open, so the option does not slow down the first database access of the next request.

Work File Close (`closeWorkFilesPerRequest`)

Closes the work files that the previous request left open.

- **Values:** `true/false` — **Default:** `true`
- **Environment variable:** `NATSUBSRV_CLOSE_WORK_FILES_PER_REQUEST`
- **Choosing a value:** Leave it on unless your subprograms close their own work files or use none at all.

A `LOGON` closes no work file either. Without this option, a work file opened by one request stays open into the requests of other end users, who then write into it — and the records the first request wrote last stay in an unwritten buffer.

The close is performed at the end of a Natural session, not at the end of a program: it also closes work files configured with `CLOSE MODE=USER`, whose purpose is to outlive a program. On a sub-program server, it is the end of the request that a work file must not outlive.

Stopping and Restarting

- Send `SIGTERM` (for example, `kill <pid>` or through your service manager) or press `Ctrl+C` (`SIGINT`) to stop the server. The parent process shuts its workers down in an orderly manner.
- Requests that are already being processed are completed; the server then terminates.
- Configuration changes take effect only after a restart. Change the configuration file or the environment variables and start the server again.
- If a worker terminates unexpectedly, the event is written to the log. See [Crash Reports](#).

6 Monitoring

■ Logging	36
■ Metrics	37
■ Traces	38
■ Retrieving Health and Telemetry Data	38

This document covers the following topics:

Logging

Log messages are written to standard output. The level is set with `loggingMode` (environment variable `NATSUBSRV_LOG`).

Level	Use
OFF (alias QUIET)	No log output.
CRITICAL	Only errors that terminate the server.
ERROR	Failed requests and server errors.
WARN	Situations that need attention but do not stop processing.
INFO (alias ON)	Default. Startup, configuration, shutdown, and important events.
DEBUG	Detailed processing information; request and response payloads truncated to the first and last 512 bytes.
TRACE	Similar to <code>DEBUG</code> , but with complete request and response payloads.

Use `INFO` in production. `DEBUG` and `TRACE` produce large amounts of output and slow down processing — enable them only for analysis and switch them off afterwards.

Passwords are never written to the log. The password field of a request is always replaced with asterisks at every log level including `TRACE`, and this cannot be switched off. All other payload content is not masked, however, `TRACE` writes complete request and response payloads and therefore exposes the data your subprograms exchange. Use test data rather than production data when logging at `TRACE`.

Syslog

Set `enableSyslog` to `true` (environment variable `NATSUBSRV_SYSLOG`) to send log messages to the system's syslog service in addition to standard output. On Linux, you can then read them with `journalctl` or in `/var/log/messages`, depending on your system configuration.

Debug-level messages reach syslog only if `loggingMode` is set to `DEBUG` and your syslog daemon forwards that level. For `rsyslog`, adjust `/etc/rsyslog.conf`:

```
*.debug;mail.none;authpriv.none;cron.none /var/log/messages
```

Then, restart the service:

```
sudo systemctl restart rsyslog
```

Metrics

Set `metrics` to `true` (environment variable `NATSUBSRV_METRICS`) to enable metrics collection. Each worker then reports its values to the parent process every `metricsInterval` seconds (default: 15). Shorter intervals give a finer resolution at the cost of slightly more overhead.

Scope of this release: metrics collection is implemented in the server and can be switched on. This release does not include a mechanism for exporting the collected values to an external monitoring system, so the values cannot be read out of a running server. Please take this into account when planning your monitoring. See [Retrieving Health and Telemetry Data](#).

The following metrics are provided:

Metric	Type	Description
<code>total.connections</code>	Counter	Client connections established since server start.
<code>subprogram.calls</code>	Counter	Natural subprogram invocations.
<code>calls.per.subprogram</code>	Counter	Calls per subprogram, tagged by library and subprogram name.
<code>invalid.subprogram.calls</code>	Counter	Failed or invalid subprogram calls.
<code>invalid.calls.per.subprogram</code>	Counter	Invalid calls per subprogram, tagged by library and subprogram name.
<code>cpu.uptime / cpu.stime</code>	Gauge	Cumulative CPU user/system time in seconds across all workers.
<code>cpu.uptime.normalized / cpu.stime.normalized</code>	Gauge	CPU user/system time as a percentage (0–100) over the interval.
<code>memory.rss.kb</code>	Gauge	Physical memory used by all workers, in kilobytes.
<code>memory.vms.kb</code>	Gauge	Virtual memory used by all workers, in kilobytes.
<code>connection.dur.min.ms / max / avg</code>	Gauge	Minimum, maximum, and average connection duration in milliseconds.
<code>connection.dur.sample.count</code>	Counter	Sample size for the connection duration average.
<code>request.size.min.bytes / max / avg</code>	Gauge	Minimum, maximum, and average request payload size in bytes.
<code>request.size.sample.count</code>	Counter	Sample size for the request size average.
<code>listen.socket.pending.accepts</code>	Gauge	Connections waiting to be accepted.
<code>requests.per.interval</code>	Gauge	Requests processed in the current interval.

Metric	Type	Description
<code>amount.of.workers</code>	Gauge	Currently active worker processes.
<code>network.throughput.sent.bytes</code>	Counter	Bytes sent over the network.
<code>network.throughput.received.bytes</code>	Counter	Bytes received over the network.

Counters increase continuously and are reset when the server restarts; gauges represent a current value.

Useful starting points: `invalid.subprogram.calls` and `invalid.calls.per.subprogram` to detect failing subprograms, `listen.socket.pending.accepts` together with `connection.dur.avg.ms` to decide whether *numberOfWorkers* needs to be increased, and `memory.rss.kb` to size the machine.

Traces

A trace records the individual processing steps of a single request and thereby shows where its time is spent.

Tracing is activated per request. The request must carry a valid W3C `traceparent` value, which the middleware adds. Additionally, `metrics` must be enabled — otherwise traces are created but not forwarded to the parent process.

`tracesStoredPerWorker` (environment variable `NATSUBSRV_TRACES_PER_WORKER`, default 1000) determines how many traces a worker buffers before the oldest entries are overwritten. Increase the value if traces are collected less frequently than they are produced.

The log level does not activate tracing, it only controls whether the measured times additionally appear as log messages. The log output is the only way to inspect the measured times. See [Retrieving Health and Telemetry Data](#).

Retrieving Health and Telemetry Data

Natsubsrv does not offer an HTTP endpoint of its own. Its health status is made available by the surrounding components: the middleware (for example, the Natural OpenAPI Server) checks the availability of Natsubsrv and reports it through its own interfaces.

Metrics and traces: this release does not include a mechanism for exporting the collected metrics and traces to an external monitoring system. Collection in the server itself is in place — the metrics listed above are gathered and traces are recorded while the server is running — but the collected data cannot be read out from outside the server.

Base your operational monitoring on:

- the health check of the middleware for availability,
- log output (see [Logging](#), optionally through [Syslog](#)) for errors, startup and shutdown events, and request-level detail,
- the operating system: monitor the server process and its listening port, and take CPU and memory figures from the process itself.

The collected values cannot be read out and thus, enabling `metrics` yields no additional insight for monitoring in this release. The overhead of enabling it is small, so either setting is unproblematic.

7 Security

■ Natural Security	42
--------------------------	----

This document covers the following topics:

Natural Security

Natural Security is in use when it is part of the installation; `npisrv` then points to the `natsubsvrsec` binary. The Natural runtime starts with `auto=on`, and the initial session runs under the system user, which must exist in Natural Security.

Natural Security is enforced only by the `natsubsvrsec` binary. Since both binaries are always installed, make sure that service operation and any manual start go through `npisrv` or `natsubsvrsec` — starting `natsubsvr` in a Natural Security installation results in a server that does not apply Natural Security.

Incoming requests must be authenticated with Natural Security credentials, or through PAM when it is configured. Successful PAM authentication can provide single sign-on into Natural Security.

- [Keep Security Session \(`keepSecuritySession`\)](#)
- [Startup Logon Retry \(`nscStartupLogonRetryCount`\)](#)
- [Authentication \(PAM\)](#)
- [Transport Security \(TLS/SSL\)](#)
- [File System Security](#)

Keep Security Session (`keepSecuritySession`)

Controls whether the Natural Security session is kept between requests instead of being ended and rebuilt after every request.

- **Values:** `true/false` — **Default:** `false`
- **Environment variable:** `NATSUBSRV_KEEP_SECURITY_SESSION`
- **Choosing a value:** Enable it when many requests arrive for the same user — for example, when the middleware calls `Natsubsvr` with a single technical user, or with high request rates and short-running subprograms. The fewer distinct users, the larger the benefit; with constantly changing users there is little to gain.

Credentials are still verified by Natural Security on every request; requests with wrong credentials are rejected as before. Only the teardown and reload of the security context is skipped, which makes consecutive requests of the same user considerably faster. Each worker keeps its own session, so with several workers the same user may still need a full logon per worker.

Consider these before enabling:

- **Audit:** Natural Security records one continuous logon per worker instead of a logon/logoff pair per request. If your organization requires per-request records, keep the option switched off.

- Idle sessions: While a worker is idle, the session of its last user remains logged on.
- Periodic SYSTEM/NATRPC39: If that call is configured, the kept session is ended beforehand so the call runs under the worker's startup user. The next request performs a full logon once.

Startup Logon Retry (`nscStartupLogonRetryCount`)

Number of Natural Security logon attempts a worker makes while starting up. The value counts attempts including the first one, so 3 means one logon plus two retries.

- **Values:** 0 (no retry) or 2–10 — **Default:** 0
- **Environment variable:** NATSUBSRV_NSC_STARTUP_LOGON_RETRY_COUNT
- **Choosing a value:** Enable it when Natsubsrv and its database are started together and the startup order is not guaranteed, for example in a container environment. Size the retry budget to the time your database realistically needs to become ready — for instance, 4 attempts every 5 seconds for a database that starts within about 15 seconds. Leave it off if the database is always up beforehand.

The wait between two attempts is set with `nscStartupLogonRetryWaitSeconds` (environment variable NATSUBSRV_NSC_STARTUP_LOGON_RETRY_WAIT, 1–60 seconds, default 5). The maximum time a worker spends retrying is $(nscStartupLogonRetryCount - 1) \times nscStartupLogonRetryWaitSeconds$, because there is no wait after the last attempt.

This option applies only to the startup logon. Logons performed later, when a client request is dispatched, are never retried: a failing request logon is reported to the client with its Natural error.

Only NAT3148 — the database is not active — is retried. All other errors, in particular wrong credentials, terminate the server immediately because waiting cannot change the result and Natural Security counts unsuccessful logons and can lock a user ID.

Consider these before enabling:

- Delayed error reporting: If the database is permanently unreachable, the startup error appears only after the full retry budget has elapsed. Keep the budget small so genuine configuration errors are still noticed quickly.
- Clients during the retry phase: The server already accepts connections on its TCP port while workers are still retrying, so a middleware connecting in this phase waits and runs into its own timeout instead of being refused right away.
- Shutdown: A stop request during a waiting phase is honored within about a second.

Authentication (PAM)

Natsubsv can authenticate users through PAM (Pluggable Authentication Modules). Set `pamService` (environment variable `NATSUBSRV_PAM`) to the name of a PAM service defined in `/etc/pam.d/`.

For example, to authenticate with an OIDC-based PAM module, create `/etc/pam.d/natsubsv`:

```
auth required pam_oidc.so jwks=/path/to/jwks.json issuer=https://your-issuer ↵
userClaim=given_name aud=your_audience
```

Adjust `jwks`, `issuer`, `userClaim`, and `aud` for your environment.

PAM authentication uses the user name exactly as supplied, preserving case. For the Natural runtime, the name is converted to uppercase and truncated to eight characters to match Natural's user field constraints.

Transport Security (TLS/SSL)

The connection between the middleware and Natsubsv can be encrypted with TLS.

- **Values:** `tlsKey` — path to the private key; `tlsCertificate` — path to the certificate — **Default:** `off`
- **Environment variable:** `NATSUBSRV_TLSKEY`, `NATSUBSRV_TLSCERT`
- **Choosing a value:** Enable TLS whenever the middleware and Natsubsv do not run on the same machine, or whenever the connection crosses a network you do not fully control.

Restrict access to the server port to the hosts running the middleware. The port speaks an internal protocol and answers *ping requests* without authentication, so it does not belong in an untrusted network.

Both values must be configured together — configuring only one causes a configuration error. Both files must exist and be readable by the runtime user; restrict access to the private key to that user. Supported protocol versions are TLS 1.2 and TLS 1.3. The middleware must be configured for TLS as well.

File System Security

Natsubsv protects the directories and files it creates:

- `appDataDir` and its `run/` and `crash/` subdirectories are created with restrictive permissions and must be writable by the runtime user only.
- Paths are resolved without following symbolic links. A symbolic link within `appDataDir` or within the parent directory of the socket path causes startup to fail.
- Paths containing `.` or `..` components are rejected.
- The parent directory of the socket must not be world-readable or world-writable.
- Crash files are created so that only the runtime user can read them.
- Run the server under a dedicated, unprivileged operating system user, and grant that user only the access required for your Natural environment.

8 Troubleshooting

■ Reporting a Problem	48
■ Common Problems	48
■ Crash Reports	49

This document covers the following topics:

Reporting a Problem

The more completely the following information is provided, the faster a problem can be analyzed:

1. A minimal example. A subprogram that reproduces the problem with as few dependencies as possible — no database access, no calls to further subprograms, hard-coded test data instead of production data. Include the parameter definition (number, type, and length of the parameters) and an example request.
2. Logs at TRACE level. Set `loggingMode` to TRACE, restart the server, reproduce the problem, and provide the complete log from the start onwards — it contains the version and the effective configuration. Note that TRACE writes complete request and response payloads — passwords are masked, other content is not — so use test data only, and switch the level back afterwards.
3. The crash report, if it exists: `crash.log` from `<appDataDir>/crash/`. See [Crash Reports](#).
4. The configuration: the `natsubsrv.json` in use and any NATSUBSRV_* environment variables that are set. Remove passwords beforehand.
5. Version and environment: Natsubsrv version and build number — from the startup log or from a [ping requests](#) — the Natural version, the Linux distribution and version, and the middleware in use with its version.
6. Classification: Does the problem occur on every call or sporadically, and since when? What changed most recently — a Natural update, a configuration change, or an increase in load?

Common Problems

Symptom	Cause	Solution
NAT6010 — memory allocation failed	Natural's user area is too small for the subprogram	Increase usize
NAT3009 after idle periods	The database context has expired	Configure natRpc39IntervalInSeconds
NAT3048 in all workers but one	The workers open Adabas with the same ETID	Set ETID=\$\$ in the Natural startup parameters
NAT3148 at startup, server terminates	The database holding FSEC/FUSER is not reachable yet	Ensure the startup order, or configure nscStartupLogonRetryCount
Request rejected as too large	The request exceeds the configured limit	Increase maxMessageSize
Repeated warnings while closing connections	The wait time is too short for the network latency	Increase closeDrainTimeout

Symptom	Cause	Solution
Requests time out under load, workers not fully utilized	Too few workers, connections queue up	Increase numberOfWorkers , check <code>listen.socket.pending.accepts</code>
Service fails immediately after starting	The Natural environment could not be set up, or a required file is missing	Inspect <code>journalctl -u sag<id>npsrv</code> — the start script names a missing <code>natenv</code> , <code>npienv</code> , or <code>npsrv</code> . See Running as a Service
Server runs, but ignores the configuration	The configuration file was not found, so defaults and environment variables were used	Pass <code>config=</code> with an absolute path; the log reports the file it looked for when none was found, and <code>DEBUG</code> shows the origin of every value
Server does not start, configuration error in the log	An invalid value, or a directory that cannot be created or written	The message names the affected parameter or directory
Server does not start, license error	The license file is missing or expired	Install a valid <i>Natural API Server</i> license
Middleware cannot connect	Wrong port, server still starting up, or a firewall in between	Send a ping requests from the middleware host to tell the two sides apart; compare the configured ports

If the startup logon is being retried, the log shows the reason and each attempt:

```
[warning] NSC LOGON failed with NAT3148 (attempt 1 of 3), retrying in 5 second(s).
[warning] NSC LOGON failed with NAT3148 (attempt 2 of 3), retrying in 5 second(s).
[error] NSC LOGON still failing with NAT3148 after 3 attempt(s), giving up. The ↵
database holding FSEC/FUSER could not be reached.
```

Crash Reports

If the server or one of its workers terminates on a fatal signal (for example `SIGSEGV` or `SIGABRT`), `Natsubsrv` captures the crash data and stores it under `<appDataDir>/crash/`. The data is first written in a compact binary form and is converted into the readable report `crash.log` afterwards — for a worker by the parent process, for the server itself at its next start.

Provide `crash.log` when [reporting a problem](#). The file is appended to, so it can contain several crashes; keep the whole file rather than an excerpt.

III

Natural OpenAPI Server on Tomcat

This document covers the following topics:

Installation
Configuration
Operation
Monitoring
Security

9 Installation

- Installing the Natural OpenAPI Server 54
- Installing the Natural OpenAPI Server as systemd service 54

Instructions on how to install the Natural OpenAPI Server.

Installing the Natural OpenAPI Server

➤ To install the Natural OpenAPI Server

- 1 Open the Software AG Installer and navigate to the installation tree.
- 2 Expand the **Natural** node and select **Natural OpenAPI Server**.
- 3 Follow the onscreen instructions to complete the installation.

Installing the Natural OpenAPI Server as systemd service

The Natural OpenAPI Server can be configured as a systemd service so that it starts automatically when the operating system starts.

➤ To configure Natural OpenAPI Server as a systemd service

- 1 Copy the *sag_noa.service* file from the */NaturalOpenAPIServer/INSTALL* directory to the systemd service directory:

```
sudo cp /NaturalOpenAPIServer/INSTALL/sag_noa.service ↵  
/usr/lib/systemd/system/sag_noa.service
```

- 2 Enable the service so that it starts automatically during system startup:

```
sudo systemctl enable sag_noa
```

10 Configuration

■ General Configuration	56
■ Natural Error to HTTP Mapping	57

This document covers the following topics:

General Configuration

The following configurations have to be done before running the Natural OpenAPI server.

In the *application.properties* file, define the following properties:

Parameter	Description
<code>com.softwareag.natural.openapi.server.config.host.address</code>	The IP address of the Natural Subprogram Server.
<code>com.softwareag.natural.openapi.server.config.host.port</code>	The port to connect to the Natural Subprogram Server.
<code>natural.integration.services.files</code>	A list of JSON files that should be accessed by the Natural OpenAPI Server (full file name and path). The files should be separated by comma. For examples, <code>/opt/softwareag/NaturalOpenAPIServer/services/naturalIntegrationServices.json</code> <code>/opt/softwareag/NaturalOpenAPIServer/services/naturalIntegrationServices2.json</code>
<code>natural.integration.services.directory</code>	The location of folder that contains all the JSON definition files.
<code>com.softwareag.natural.openapi.server.config.host.ssl.enabled</code>	Define whether to use SSL or not.
<code>com.softwareag.natural.openapi.server.config.host.ssl.certsChain[0]</code>	Location of the public certificate files to be used by the Natural Subprogram Server using SSL.
<code>com.softwareag.natural.openapi.web.config.authenticationType</code>	The Authentication used to connect to the Natural Subprogram Server.
<code>com.softwareag.natural.openapi.web.config.oAuth2Config.tokenUrl</code>	The token URL of the credentials provider. The token is a Bearer token when testing with the Swagger client.
<code>com.softwareag.natural.openapi.web.config.oAuth2Config.scopes</code>	The scopes to request from the credentials provider. The token is a Bearer token when testing with the Swagger client.
<code>com.softwareag.natural.openapi.server.config.host.charset</code>	The charset to use when encoding the request and response of the Natural Subprogram Server.
<code>management.endpoints.web.exposure.include</code>	This parameter controls the exposure of the service endpoint. If not defined, this parameter is added with the value <code>reloadService</code> . If defined, append <code>reloadServices</code> to the existing comma-separated values.



Notes:

1. You can define a file list in the `natural.integration.services.files` property and define another folder in the `natural.integration.services.directory` property. The application will use the resources from both locations.
2. All the JSON files created by the Natural OpenAPI Eclipse Plug-In should be copied to the service folder (defined in the `application.properties` file by the `natural.integration.services.directory` property, or they should be defined in the `natural.integration.services.files` property).

Natural Error to HTTP Mapping

The Natural OpenAPI Server can map backend Natural error codes to HTTP status codes.

The mapping resolution order is:

1. Exact match (for example, NAT3009)
2. Matching numeric range (for example, 0..99)
3. Global default mapping

Default mappings are predefined in the Natural OpenAPI Server.

You can override or extend mappings in the `application.properties` file using the `error-mapping.*` prefix.

Example: Override one Natural specific code

```
error-mapping.sources.natural.specific.NAT3009.http-code=422
error-mapping.sources.natural.specific.NAT3009.http-message=Unprocessable Entity
error-mapping.sources.natural.specific.NAT3009.category=VALIDATION_ERROR
error-mapping.sources.natural.specific.NAT3009.log-level=WARN
```

Example: Define a Natural range mapping

```
error-mapping.sources.natural.ranges.syntax.from=0
error-mapping.sources.natural.ranges.syntax.to=49
error-mapping.sources.natural.ranges.syntax.http-code=400
error-mapping.sources.natural.ranges.syntax.http-message=Bad Request
error-mapping.sources.natural.ranges.syntax.category=SYNTAX_ERROR
error-mapping.sources.natural.ranges.syntax.log-level=WARN
```

Example: Override global fallback

```
error-mapping.default-mapping.http-code=500
error-mapping.default-mapping.http-message=Internal Server Error
error-mapping.default-mapping.category=SYSTEM_ERROR
error-mapping.default-mapping.log-level=ERROR
```

In addition to Natural code mappings, transport failures are mapped by the WebFlux error handler:

- Backend connection failure: 502 Bad Gateway
- Backend timeout: 504 Gateway Timeout
- Unknown/unmapped errors: 500 Internal Server Error

11

Operation

■ Starting and Stopping the Server	60
■ Deploying Web Application with Tomcat	60
■ Deploying OpenAPI Resources	61

This document covers the following topics:

Starting and Stopping the Server

To start or stop the Natural OpenAPI Server using the embedded webserver delivered with the *NaturalOpenAPIServer.war* file, navigate to the installation folder */NaturalOpenAPIServer/bin/* and run the `startup.sh` or `shutdown.sh` shell script respectively.

Deploying Web Application with Tomcat

The following are some specific configurations for Apache Tomcat® if you want to deploy the Natural OpenAPI Web application with your own Tomcat.

- [Deploying War File](#)
- [Configuring Server.XML](#)

Deploying War File

The *NaturalOpenAPIServer.war* must be copied to the */tomcat/webapps/* directory.

In addition, the `SPRING_CONFIG_LOCATION` environment variable must be configured to point to the location of the *application.properties* file.

Configuring Server.XML

Alternatively, the following configuration can be added to the Tomcat's *server.xml* file when deploying the Natural OpenAPI Server.

```
<Context path="/noa1" docBase="noa1" override="true">
  <Environment name="SPRING_CONFIG_LOCATION" ↵
value="/your-path/conf/application.properties" type="java.lang.String" />
</Context>
```

Deploying OpenAPI Resources

To deploy the services described in the *Developer Overview* > [Generating OpenAPI Resources](#) section, the Natural OpenAPI Server must be configured with the location of the service files.

The service file location can be configured in the following ways:

- Using `natural.integration.services.directory` – Set this property in the *application.properties* file to specify the directory containing the service files. Place the service files in the specified directory.
- Using `natural.integration.services.files` – Set this property to specify the names of the specific service files to be deployed. The specified files must be located in the classpath of the application.

When the Natural OpenAPI Server starts, it loads the configured service files and registers the OpenAPI services defined in those files.

Each service file contains the definition and mapping information required to expose a Natural subprogram as a REST API service. Once the server has successfully started, all services defined in the loaded service files are available to clients through the Natural OpenAPI Server. For more information, see the `management.endpoints.web.exposure.include` parameter in the [General Configuration](#) section.

Adding or modifying a service therefore requires updating the corresponding service file and re-starting the server or using the reload actuator endpoint for the changes to be loaded and the services to become available.

12

Monitoring

OpenTelemetry is an open standard for collecting and managing observability data from applications and services. It provides a common framework that helps monitor and analyze the behavior, performance, and health of a system.

Our product uses the three main types of telemetry data defined by OpenTelemetry:

- **Traces** – Provides information about the flow of a request through the system. Traces help follow an operation across different components and identify where time is spent or where failures occur.
- **Metrics** – Provides numerical measurements about the system, such as resource usage, request counts, response times, and other performance-related values. Metrics can be used to monitor the overall behavior and health of the application.
- **Logs** – Provides detailed records of events that occur during application execution. Logs can help investigate errors, warnings, and other relevant runtime events.

By collecting traces, metrics, and logs, OpenTelemetry provides a more complete view of the system and makes it easier to monitor application behavior, identify performance issues, and troubleshoot problems.

13

Security

■ Secure Communication between Natural OpenAPI Server and Subprogram Server	66
■ Secure Communication between Client and Natural OpenAPI Server	66
■ Authentication	67

The Natural OpenAPI Server provides configuration options to secure communication:

- between the Natural OpenAPI server and the Subprogram Server.
- between the client and the Natural OpenAPI Server.

This document covers the following topics:

Secure Communication between Natural OpenAPI Server and Subprogram Server

The Natural OpenAPI Server can use SSL/TLS to encrypt communication between the Natural OpenAPI Server and the subprogram servers.

SSL/TLS can be enabled using the following property:

```
com.softwareag.natural.openapi.server.config.host.ssl.enabled=true
```

When SSL/TLS is enabled, the server uses the configured certificate chain to establish secure connections. The certificate can be configured using the `certsChain` property:

```
com.softwareag.natural.openapi.server.config.host.ssl.certsChain[0]=classpath:natsubsrv.server.cert.crt
```

The property specifies the location of the server certificate. The `classpath:` prefix indicates that the certificate is loaded from the application's classpath.

Multiple certificates can be specified as part of the certificate chain by using additional indexed properties, for example, `certsChain[1]`, `certsChain[2]`, and so on.



Note: The SSL certificate configuration is relevant only when

`com.softwareag.natural.openapi.server.config.host.ssl.enabled` is set to `true`.

Secure Communication between Client and Natural OpenAPI Server

The Natural OpenAPI Server supports secure communication using SSL/TLS. When SSL/TLS is enabled, clients connect to the server using HTTPS instead of HTTP.

The default port used by the server is 8080. A different port can be configured for HTTPS, for example, 8443, by setting the following property:

```
server.port=8443
```

To enable SSL/TLS, set the following property:

```
server.ssl.enabled=true
```

The following properties configure the SSL certificate and key store used by the server:

Key Store Path

This specifies the path to the key store containing the SSL certificate and private key. For example:

```
server.ssl.key-store=file:///opt/softwareag/NaturalOpenAPIServer/conf/keystore.p12
```

Key Store Password

This specifies the password required to access the key store. For example:

```
server.ssl.key-store-password=password
```

Key Store Type

This specifies the type of key store. The supported types are JKS and PKCS12. For example:

```
server.ssl.key-store-type=PKCS12
```

Key Store Alias

This specifies the alias used to identify the certificate and private key within the key store. For example:

```
server.ssl.key-alias=noa-server
```



Note: The key store must contain a valid SSL/TLS certificate and its corresponding private key.

Authentication

The Natural OpenAPI server does not authenticate the client sending the request by itself, but rely on the authentication module of the Natural Subprogram Server. However, if the Natural Subprogram Server is configured to use an authentication method, you must define the appropriate configuration in the Natural OpenAPI Server. Defining the correct configuration will create the related definition in the OpenAPI document located at http://localhost:OpenAPIServer_port/v3/api-docs.

There are 3 different ways to define the authentication:

Using Basic Authentication

Setting the `com.softwareag.natural.openapi.web.config.authenticationType` property to `basic` will require the client application to send username and password encoded with Base64 encoding (defined in RFC-7617).

Using OAuth2 Token

Token retrieval is the responsibility of the client application, while token validation is performed by the Natural Subprogram Server authentication module.

Setting the `com.softwareag.natural.openapi.web.config.authenticationType` property to `oauth2` adds a requirement to the OpenAPI document for the client application to send a Bearer token.

The client application is required to include the special HTTP header *X-Impersonate-User* to specify the user to impersonate, alongside the provided token.

The following configuration can be used when testing with the Swagger UI web client. The Swagger UI embedded in the Natural OpenAPI Server currently supports only the Client Credentials flow. When OAuth2 authentication is enabled, you can define the `tokenURL` and required scopes. These values are exposed in the OpenAPI document but are not used by the Natural OpenAPI Server for authentication. In the case of impersonation, it must be performed after a successful token verification.

Using Anonymous Requests

When an authentication type is not defined in the *application.properties* file, the requests are considered anonymous by default and should not be authenticated. It is also possible to execute the request as another user using the special HTTP header *X-Impersonate-User*.

IV

Developer Overview

This document covers the following topic:

Installation of the NaturalONE OpenAPI Plugin
Configuration of the NaturalONE OpenAPI Plugin
Generate and Deploy Services
Special Considerations

14

Installation of the NaturalONE OpenAPI Plugin

The NaturalONE OpenAPI Plugin is required to create and manage Natural OpenAPI services.

The plugin runs in the NaturalONE IDE. If NaturalONE is not already installed, install it when installing the NaturalONE OpenAPI Plugin.

➤ To install the NaturalONE OpenAPI Plugin

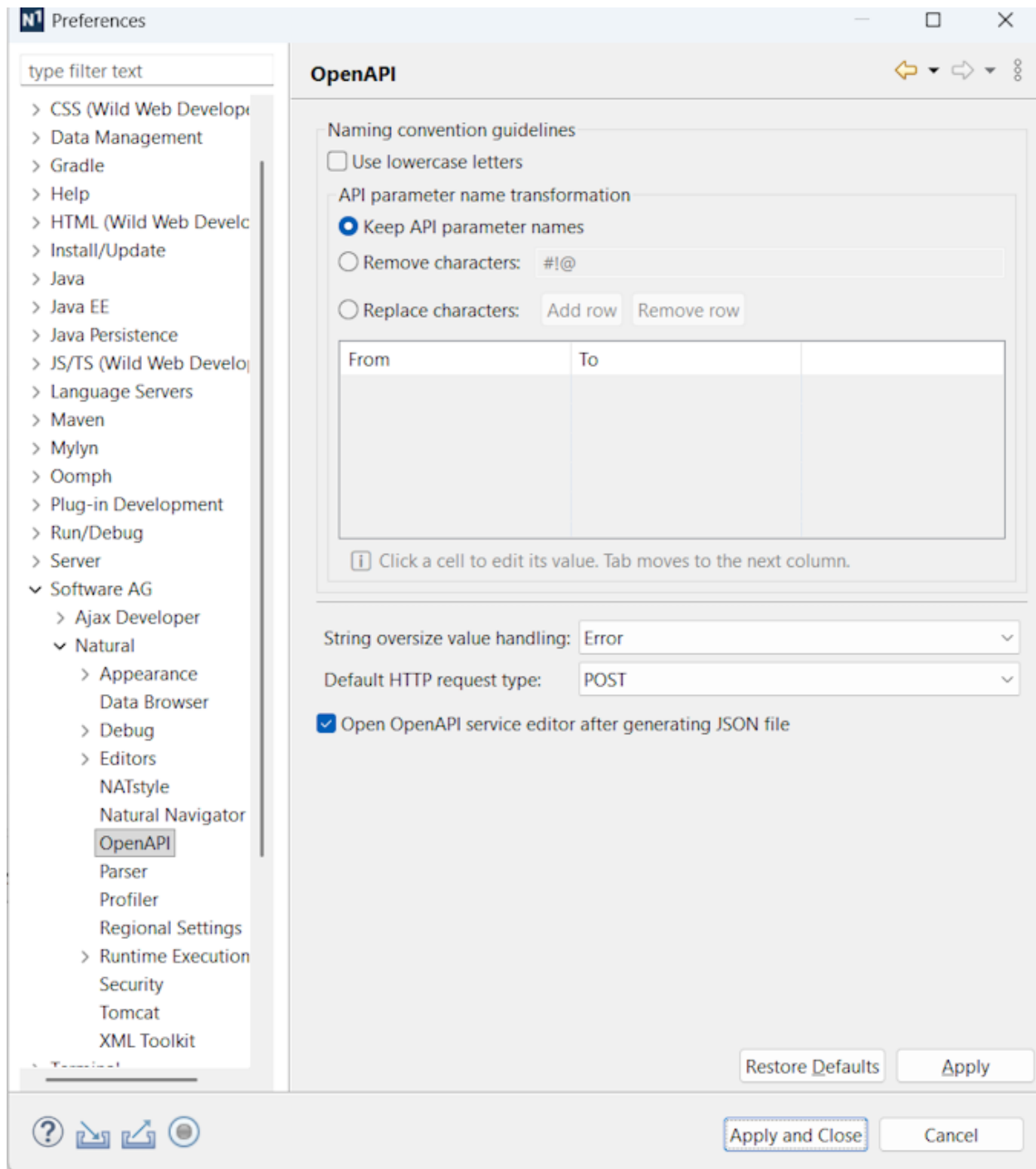
- 1 Open the Software AG Installer and navigate to the installation tree.
- 2 Expand the **NaturalONE** node and select **OpenAPI Development**.

15

Configuration of the NaturalONE OpenAPI Plugin

The global default settings for OpenAPI service generation can be configured in the Eclipse **Preferences** window. These settings apply to all service files generated by the plugin.

To access these settings, go to **Window** on the main menu and select **Preferences**. On the navigation tree, expand the **Software AG** node, **Natural** node, and then select **OpenAPI**.



The settings in the **Preferences** window define the default values used during OpenAPI service generation when a corresponding property has not been explicitly defined in the subprogram itself.

The following settings are available:

Use lowercase letters

Select this checkbox to automatically convert parameter names to lowercase during the OpenAPI service generation process.

API parameter name transformation

This setting provides options to automatically transform OpenAPI parameter names. The following options are available:

- **Keep API parameter names** – This option keeps the parameter names unchanged.
- **Remove characters** – This option removes specified characters from parameter names. An accompanying input field allows you to specify the characters to remove (for example, #, !, or @).
- **Replace characters** – This option allows you to define mappings that specify which characters or strings to replace and the characters or strings to use as replacements instead of removing them.

String oversize value handling

This setting defines the default behavior for handling oversized string values. This setting can be overridden for an individual service in the OpenAPI Editor.

The following options are available:

- **Truncate** – This option truncates the value when it exceeds the allowed size.
- **Error** – This option generates an error when the value exceeds the allowed size.

Default HTTP request type

This setting defines the default HTTP request method that is selected when the OpenAPI Editor is opened, if no HTTP request type was defined in the subprogram.

Open OpenAPI service editor after generating JSON file

This setting controls whether the OpenAPI Editor opens automatically when `Generate OpenAPI Service` is triggered.

By default, this checkbox is selected. When it is cleared, the service file is generated immediately without opening the editor.

When multiple subprograms are selected and `Generate OpenAPI Service` is triggered, this setting is bypassed and the OpenAPI Editor is not opened.

16

Generate and Deploy Services

■ Generating OpenAPI Resources	78
■ Viewing OpenAPI Resources via Editor	79

This document covers the following topics:

Generating OpenAPI Resources

Creating OpenAPI Services

To create OpenAPI services, the Natural OpenAPI Server needs to read the definition of a Natural subprogram. This information is modeled and stored in a service JSON file.

Each service JSON file represents a single Natural subprogram and contains the following information:

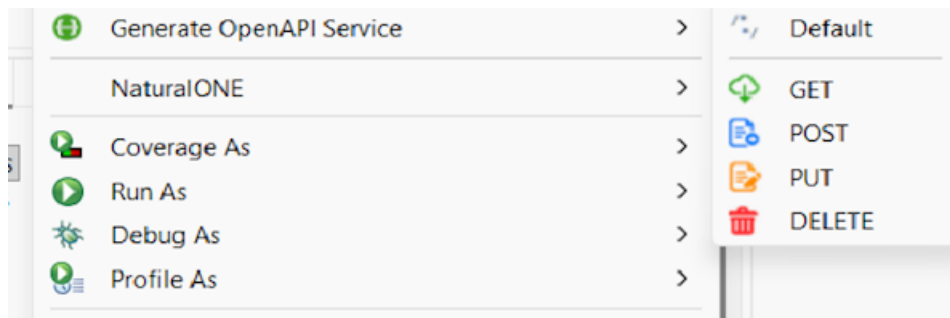
- Natural subprogram and library names
- OpenAPI schema mappings between OpenAPI parameters and Natural input and output parameters
- Query parameters that can also be mapped to Natural parameters



Note: Query parameters currently do not support array mappings.

➤ To create an OpenAPI service

- 1 Right-click the desired subprogram.
- 2 Select **Generate OpenAPI Service**. The following menu is displayed with these options:

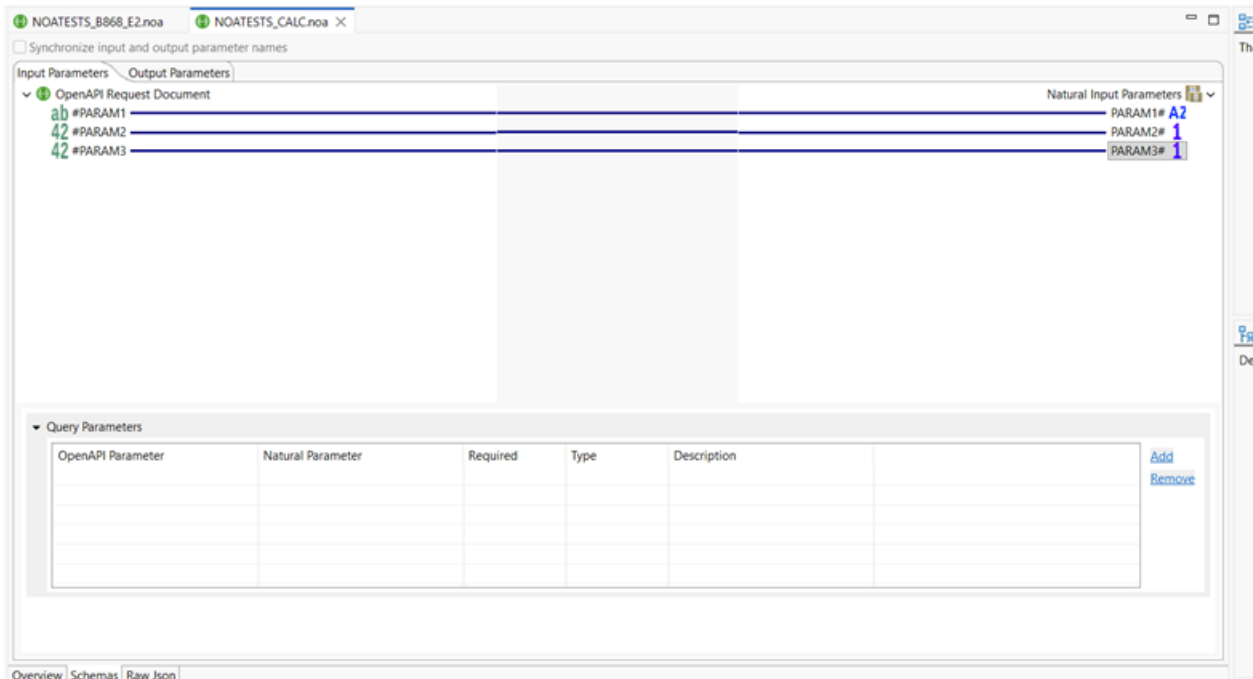


- **Default** – Uses the HTTP method defined in the subprogram. If no HTTP method is defined in the subprogram, the value configured in **Preferences** is used.
- **GET / POST / PUT / DELETE** – Selecting a specific HTTP method overrides the default method and defines the method that the service will use.



Note: Array parameters currently do not support the **GET** and **DELETE** HTTP methods.

Viewing OpenAPI Resources via Editor



The OpenAPI Service Editor displays information and configuration settings of the services. It consists of three tabs:

- Overview Tab
- Schemas Tab
- Raw Json Tab

Overview Tab

The **Overview** tab displays general information and configuration settings for the service.

This tab also allows you to define additional general information, such as summary, description, and tags.

HTTP Request Type

Indicates the HTTP method used by the service. The supported methods are **GET**, **POST**, **PUT**, and **DELETE**.

Behavior When a Value Is Too Large

Defines how the service handles an incoming OpenAPI value that exceeds the size of the corresponding mapped Natural parameter.

This setting can be configured for each individual service in the editor. The default value is based on the option selected at *String oversize value handling* in the Eclipse **Preferences** window.

Schemas Tab

This tab consists of two subtabs: **Input Parameters** and **Output Parameters**. The names of the properties on these two tabs can be synchronized by selecting the **Synchronize input and output parameter names** checkbox.

Input Parameters Tab

This tab displays the OpenAPI Schema layout for the Natural input parameters. The layout provides a mapping interface between Natural input parameters and OpenAPI request document.

The OpenAPI Schema layout consists of three areas and one section:

1. **OpenAPI Request Document** (left tree area): Displays the OpenAPI document. Right-clicking an OpenAPI element in the tree displays the following options:
 - **Rename** – Changes the name of the OpenAPI element.
 - **Restore Mapping** – Restores a mapping that was previously removed.
 - **Delete** – Deletes the OpenAPI parameter if it is not currently mapped.
2. Mapping Controls area: Provides the option to remove a mapping. Removing a mapping means that the OpenAPI element is no longer used. The corresponding Natural parameter receives its default value (for example, an empty string for alphanumeric parameters or 0 for numeric parameters).
3. **Natural Input Parameters** (right tree area): Displays the Natural parameters. Right-clicking a Natural parameter displays the following options:
 - **Map to Document** – Creates or restores a mapping to an element of the OpenAPI document.
 - **Map to Query Parameter** – Creates a query parameter mapped to the Natural parameter (enabled only if the Natural parameter is not mapped).
4. **Query Parameters** section: Allows you to add and/or remove mappings between query parameters and Natural input parameters. Query parameters are passed as part of the URL and are not included in the REST request body.

Output Parameters Tab

This tab displays the OpenAPI Schema layout for the Natural output parameters. The layout provides a mapping interface between Natural output parameters and OpenAPI response document.

Raw Json Tab

This tab displays the JSON file of the service.

17

Special Considerations

■ Developing Natural Subprograms for Natsubsvr	84
--	----

This section covers topics of relevance for developers.

Developing Natural Subprograms for Natsubsrv

This document lists the differences when a Natural subprogram is executed by the Natural Subprogram Server (Natsubsrv) instead of by a terminal session.

It is not an introduction to Natural programming — for that, see the Natural documentation.

These differences are caused by two properties:

- No terminal. The session is a server session; there is no screen and no keyboard.
- No state between calls. One session serves the requests of many end users, one after another, and is reset at every request boundary.

This document covers the following topics:

- [Execution Model](#)
- [Mapping OpenAPI Names](#)
- [Restricted Statements](#)
- [Statelessness](#)
- [Transactions and Work Files](#)
- [Error Handling](#)
- [System Variables](#)
- [Checklist](#)

Execution Model

- The subprogram is entered by a `CALLNAT` and leaves by its normal end or by `ESCAPE ROUTINE`.
- The `DEFINE DATA PARAMETER` block is the only data channel into and out of the subprogram.
- Every request names its own library and its own user; both are established by the server before the `CALLNAT`.
- The session is reset at the request boundary, so every call starts from a defined state and leaves nothing behind.

Mapping OpenAPI Names

In a subprogram, you can define the name of the API parameter that will be mapped to a Natural parameter.

This is done by adding the prefix `oapi.n:` followed by the desired property name in the comment of the field.

For example, in Natural:

```
#PARAM1 (A1) /* oapi.n:nameParm IN
```

This creates an API parameter named `nameParm`, which is mapped to the `#PARAM1` Natural parameter.

Restricted Statements

- **Must not be used**
- **Restricted use**
- **Must not block**

Must not be used

Statement	Reason
TERMINATE	Ends the session the server owns. The request is abandoned and output parameters are not written back.
STOP	Leaves the <code>CALLNAT</code> without writing the output parameters back. Use <code>ESCAPE ROUTINE</code> .
FETCH	Starts a program inside the request. The rest of the subprogram does not run and output parameters are not written back.
RUN	Requires source and a compiler in a session that serves requests.
LOGON	The library environment is established by the server per request.
REINPUT	Requires a terminal.
Maps (INPUT USING MAP, WRITE USING MAP), PROCESS PAGE	Requires a terminal.
SET KEY	Requires a terminal. PF key settings are reset per request.
SET CONTROL	Requires a terminal.
OPEN CONVERSATION, CLOSE CONVERSATION	Conversational RPC is not supported.
DEFINE DATA CONTEXT	Context variables are the data of a conversation, and there is none. The area belongs to the single <code>CALLNAT</code> that creates it and is released with it, so it carries nothing from one call to the next. Use the <code>PARAMETER</code> block.

Restricted use

Statement	What is Possible
STACK	Data can be placed on the stack and read back within the same request. Commands are not executed.
INPUT	Only without terminal I/O, that is, reading data from the stack. Any INPUT that would address a screen fails.

Must not block

A blocked subprogram blocks the session it runs in, and with it every request that would have been served by it.

- Do not wait for user interaction, external events, or locks held elsewhere.
- Keep CALLs to 3GL modules short. The server cannot interrupt a running 3GL call, so the session stays occupied for as long as the call lasts.
- Keep units of work short. Database locks are held until `END TRANSACTION` or `BACKOUT TRANSACTION`, so a long unit of work keeps other requests away from the same records.

The same applies to the subprogram as a whole: its runtime is the time the session is unavailable to everyone else.

Statelessness

Every call must stand on its own: it must not carry data forward, and it must not expect data left behind by an earlier call.

The following are put back to a defined state at every request boundary:

- Global data areas, including the COMMON GDA.
- Application independent variables (`DEFINE DATA INDEPENDENT`).
- Session parameters, including those changed with `SET GLOBALS` and `*LANGUAGE`.
- The limit counters `MADIO`, `MAXCL`, `MAXERR`, and `LT`.
- Steplib chains.
- Retained sets and command IDs.
- PF key settings.

A GDA is empty at the start of every request unless the subprogram fills it itself, and a set retained by `RETAIN` in one request does not exist in the next.

Context variables (`DEFINE DATA CONTEXT`) are no way around this either. Without a conversation, they are not a data area that spans calls: the context exists for the one `CALLNAT` and is gone when

it returns, so it can neither carry state forward nor be read by the next request. For more information, see *Must not be used*.

*ETID and the ET data stored under it belong to the session, not the end user. They are shared by all requests the session serves and must not be used to carry per-user state.

Transactions and Work Files

End your own units of work. Complete every one with `END TRANSACTION` or `BACKOUT TRANSACTION` before returning. The server does not commit.

At the request boundary, the server normally backs out the transaction (`BACKOUT TRANSACTION`) and closes open work files. On Linux, you can disable either behavior with `backoutTransactionPerRequest` and `closeWorkFilesPerRequest`; otherwise both behaviors always occur.



Notes:

1. A unit of work that is not committed within the request is lost.
2. Records must not be held across calls.
3. A work file must be opened, used and closed within one request. `CLOSE MODE=USER` does not keep it open across the request boundary.

Error Handling

Errors that occur while a request is being processed are intercepted by the server and reported to the caller with error number, line, status, and object name.

It is recommended to use `ON ERROR` to return errors as data the caller can act on — an error code and message in the output parameters — and reserve unhandled errors for genuinely unexpected conditions. Clean up on the error path exactly as on the success path.

System Variables

Variable	Value
*SERVER-TYPE	CALLNAT
*USER	The user of the current request in uppercase, up to 8 characters.
*NET-USER	The user of the current request, like *USER. On z/OS, it holds the user of the session the server started instead — the same value as *INIT-USER — and therefore does not identify the requesting user.
*LIBRARY-ID	The library named by the current request.
*INIT-USER, *INIT-ID, *ETID	The session the server started, not the requesting user. On z/OS, *INIT-ID holds a session number that the server increments for each new session.

Variable	Value
*DEVICE	A batch device.

Checklist

- ☐ No TERMINATE, STOP, FETCH, RUN, and LOGON.
- ☐ No REINPUT, no maps, no SET KEY, no SET CONTROL; INPUT only without terminal I/O.
- ☐ No OPEN CONVERSATION / CLOSE CONVERSATION, no DEFINE DATA CONTEXT.
- ☐ The subprogram read only values that it had set itself; it did not read values from a GDA, independent variables, or values retained from an earlier call.
- ☐ Every unit of work ended with END TRANSACTION or BACKOUT TRANSACTION, on every path.
- ☐ Every work file opened is closed, on every path.
- ☐ Nothing blocks — neither the subprogram nor a 3GL module it calls.
- ☐ The end user is identified through *USER, not through *NET-USER, *INIT-USER, or *ETID.