

JSignPdf User Guide

Josef (kwart) Cacek

3.2.1

Table of Contents

Introduction . . . 1
- License . . . 1
- History . . . 1
- Author . . . 1
- Getting support . . . 1
- Contributing . . . 2

Installation and prerequisites . . . 3
- Downloading JSignPdf . . . 3
- Java . . . 4
- Obtaining a keystore . . . 4
- Demo files . . . 5

Launching JSignPdf . . . 6
- From a native installer (MSI / DEB / RPM / DMG / Flatpak) . . . 6
- From a cross-platform ZIP . . . 6

Using the JavaFX UI . . . 8
- Opening a PDF document . . . 8
- Configuring signature options . . . 9
- Placing a visible signature . . . 10
- Signing the document . . . 11
- Using presets . . . 12
- Adjusting advanced preferences . . . 13

Classic Swing UI . . . 15

Command line (batch mode) . . . 16
- General options . . . 16
- Keystore and key options . . . 17
- Signature options . . . 17
- Output options . . . 18
- Visible signature options . . . 18
- Encryption and rights options . . . 19
- TSA and certificate validation options . . . 20
- Proxy options . . . 21
- Program exit codes . . . 21
- Examples . . . 21
- Reading passwords from standard input . . . 22

Signing options . . . 24
- Keystore selection . . . 24
- Key alias & key password . . . 24
- Signing properties . . . 25

Encryption . . . 26
Visible signature options . . . 26
Signing an existing signature field . . . 29
TSA — timestamps . . . 30
Adding a document timestamp . . . 30
Certificate revocation checking . . . 32
Proxy settings . . . 32
Using hardware tokens for signing . . . 33
Signing engines . . . 34
PAdES & the DSS engine . . . 34
Certification level and encryption with LT/LTA . . . 37
Advanced application configuration . . . 39
Migration from earlier versions . . . 40
Configuration directory . . . 40
Java VM options for the native installer launchers . . . 41
Troubleshooting . . . 43
Out of memory error . . . 43
SSL/TLS certificate errors . . . 44
Certificate not found in keystore . . . 44
Timestamp server unreachable . . . 44
Reading the signing diagnostics . . . 44
Other command line tools . . . 47
InstallCert Tool . . . 47

Introduction

JSignPdf is an open-source application that adds digital signatures to PDF documents. It's written in the Java programming language and can be launched on most current operating systems. Users can control the application using a modern JavaFX graphical interface, the classic Swing GUI, or command line arguments. Main features:

- supports visible signatures
- can set certification level
- supports PDF encryption with setting rights
- timestamp support
- certificate revocation checking (CRL and/or OCSP)

License

JSignPdf is released under LGPL and/or MPL license. It can be freely used for both personal and commercial use. For details look directly to the license files.

History

The project started at the beginning of 2008.

A greater change comes in 2021, where the project was switched to use the OpenPDF library instead of the old version of the iText library.

Starting with version 3.0.0, JSignPdf ships with a new document-centric JavaFX interface as the default graphical frontend. The classic Swing GUI remains available for backward compatibility.

Author

The author of JSignPdf is Czech developer Josef (kwart) Cacek. He works in Java since 2000. Some links to Josef's projects:

- https://github.com/intoolswetrust/
- https://github.com/kwart/

Getting support

If you don't find the relevant information in this document or on the JSignPdf GitHub page (https://github.com/intoolswetrust/jsignpdf), ask the community in the JSignPdf Google Group.

Report bugs and request features in the issue tracker (https://github.com/intoolswetrust/jsignpdf/issues). Please don't use it for usage questions — the Google Group is the place for those. Security vulnerabilities must not be reported as a public issue — use GitHub's private vulnerability reporting instead, as described in the security policy.

Contributing

Pull requests are welcome; CONTRIBUTING.md covers what to include in a bug report and what a pull request needs to pass.

Translations are maintained on Weblate and new languages are always welcome — no development environment required.

Installation and prerequisites

Downloading JSignPdf

JSignPdf releases are available on the GitHub releases page:

https://github.com/intoolswetrust/jsignpdf/releases

Each release provides per-platform installers (each bundles its own Java 21 runtime) plus two cross-platform ZIPs for users who already have Java.

Table 1. Native installers (bundled Java runtime)

Artifact	Description
`jsignpdf-${VERSION}-windows-x64.msi`	Windows 64-bit MSI installer (SignPath-signed). Installs to *Program Files* by default, creates Start Menu shortcuts and registers an Add/Remove Programs entry. Preferred for enterprise deployment via group policy or software-distribution tools.
`jsignpdf-${VERSION}-windows-x64.zip`	Windows 64-bit portable ZIP. Same content as the MSI but unzip-and-run — no installation required.
`jsignpdf-${VERSION}-linux-x64.deb` `jsignpdf-${VERSION}-linux-aarch64.deb`	Debian / Ubuntu packages for x86_64 and ARM64.
`jsignpdf-${VERSION}-linux-x64.rpm` `jsignpdf-${VERSION}-linux-aarch64.rpm`	RPM packages for Fedora / RHEL / openSUSE on x86_64 and ARM64.
`jsignpdf-${VERSION}-linux-x64.zip` `jsignpdf-${VERSION}-linux-aarch64.zip`	Portable Linux app-images. Untar and run; no system install.
`jsignpdf-${VERSION}-linux-x86_64.flatpak` `jsignpdf-${VERSION}-linux-aarch64.flatpak`	Sandboxed Flatpak bundles. Installed with `flatpak install <file>`.
`jsignpdf-${VERSION}-macos-aarch64.dmg`	macOS disk image for Apple Silicon. Currently unsigned — expect a Gatekeeper warning on first launch. Intel Macs: run the Apple Silicon build under Rosetta 2, or use the cross-platform full ZIP with a locally installed Java 21.
`jsignpdf-${VERSION}-macos-aarch64.zip`	Portable macOS `.app` bundle for Apple Silicon.

Table 2. Cross-platform ZIPs (bring your own Java 21)

Artifact	Description
`jsignpdf-${VERSION}-full.zip`	Cross-platform ZIP with all dependencies plus JavaFX natives for Windows x64, Linux x64, macOS x64 and macOS Apple Silicon. The `bin/jsignpdf.sh` / `bin\jsignpdf.cmd` launchers pick the matching JavaFX classifier at startup. Use this when you carry the same JSignPdf install across multiple machines / OSes.
`jsignpdf-${VERSION}-minimal.zip`	Cross-platform ZIP without JavaFX. Suitable for headless / CLI signing, Linux aarch64 (where OpenJFX has no native classifier), or as the basis for downstream packagers (Debian, Homebrew). The GUI falls back to the legacy Swing UI automatically when invoked.

Table 3. Package managers

Platform	Command

macOS, Linux (Homebrew)	`brew tap intoolswetrust/tap`
	`brew trust --tap intoolswetrust/tap`
	`brew install jsignpdf`

Linux (Flatpak)	`flatpak install jsignpdf-${VERSION}-linux-x86_64.flatpak`

The Homebrew formula installs the cross-platform distribution against Homebrew's `openjdk@21` package and puts `jsignpdf` on the `PATH` for both the GUI and the CLI. The `brew trust` step is required because Homebrew refuses to load formulae from third-party taps until they are trusted.

`jsignpdf-${VERSION}-SHA256SUMS.txt` covers every artifact above. Releases are also mirrored on SourceForge.



32-bit Windows is no longer supported. Users on 32-bit Windows can still run JSignPdf from the cross-platform ZIPs with their own 32-bit JRE.

Java

The cross-platform ZIPs require Java Runtime Environment (JRE) version 21 or newer. You can download it freely from the web, for instance:

- Eclipse Adoptium
- Azul Zulu



All native installers (MSI / DEB / RPM / DMG / Flatpak) bundle their own JRE, so a separate Java installation is not required when you go that route.

Obtaining a keystore

To sign PDF documents you need a keystore containing your private key. If you don't have one yet:

- **For testing**, you can generate a self-signed certificate with the `keytool` utility that ships with Java:

```
keytool -genkeypair -alias mykey -keyalg RSA -keysize 2048 -keystore keystore.p12
-storetype PKCS12
```

- **For production use**, obtain a certificate from a trusted Certificate Authority (CA). The CA will typically issue a PKCS#12 (`.p12` / `.pfx`) file.

The most common keystore types supported by Java are:

- PKCS#12 — keys stored in `.p12` and `.pfx` files
- PKCS#11 — keys stored usually on hardware modules (see Using hardware tokens for signing)
- JKS (Java Key Store)
- WINDOWS-MY — supported on MS Windows. You can use directly your certificates imported into your system.

JSignPdf has been also extended to support external keystore types like smart cards, or network HSMs. The first example is CloudFoxy (https://gitlab.com/cloudfoxy).

Demo files

Every release bundles a small `demo/` folder so you can try JSignPdf immediately without generating your own certificate or finding a sample PDF:

File	Description
`service-agreement.pdf`	Sample unsigned PDF document. The signature block on page 2 carries an empty signature field named `JaneDoe`, so Signing an existing signature field can be tried out without authoring a form first.
`jsmith.p12`	Demo PKCS#12 keystore with a self-signed certificate for `CN=John Smith`. The keystore and key password is `123456`.
`README.md`	Walk-through of the most common command-line scenarios (basic signature, visible signature, signing into the pre-placed signature field, timestamped signature with FreeTSA, appending a signature, listing keys).

You can find the folder:

- at the top level of either cross-platform ZIP (`jsignpdf-${VERSION}-full.zip` / `jsignpdf-${VERSION}-minimal.zip`), inside `jsignpdf-${VERSION}/demo/`
- under `<install dir>\app\demo\` after installing with the Windows MSI
- under `/opt/jsignpdf/lib/app/demo/` after installing the Linux DEB or RPM
- under `JSignPdf.app/Contents/Resources/demo/` inside the macOS DMG



The bundled keystore is meant purely for experimentation. Do not use it to sign real documents.

Launching JSignPdf

From a native installer (MSI / DEB / RPM / DMG / Flatpak)

When JSignPdf is installed via one of the native installers, four launchers are registered with the desktop environment and (on Linux and Windows) added to `PATH`:

Launcher	Purpose
`JSignPdf` (Windows: `JSignPdf.exe`, Linux: `JSignPdf` in `/opt/jsignpdf/bin/` + a `jsignpdf` symlink on `PATH`)	Starts the JavaFX graphical interface (the default). Also exposed as a Start Menu / application-menu entry.
`JSignPdf-swing`	Starts the classic Swing graphical interface (equivalent to passing `-Djsignpdf.swing=true` on the command line).
`JSignPdfC`	Console-mode launcher for the signer. Use this in batch scripts, CI pipelines or whenever you want the output to appear in the current terminal window.
`InstallCert`	Command-line tool for importing a server certificate into the bundled JRE's `cacerts` truststore (see InstallCert Tool).

All four launchers share the bundled JRE — you do **not** need to have Java installed separately.

From a cross-platform ZIP

After extracting `jsignpdf-${VERSION}-full.zip` (or `-minimal.zip`), launch JSignPdf from inside the extracted folder:

```
$ bin/jsignpdf.sh        # Linux / macOS
> bin\jsignpdf.cmd       # Windows
```

The launcher scripts find your JRE via `JAVA_HOME`, falling back to `java` on `PATH`. They invoke a small bootstrap class (`com.intoolswetrust.jsignpdf.Bootstrap`) which checks the Java version, picks the matching JavaFX platform jars from `lib/javafx/` and then starts the application.

By default, JSignPdf opens the JavaFX graphical interface (see Using the JavaFX UI). To start the classic Swing GUI instead, set the `jsignpdf.swing` system property via `JAVA_OPTS`:

```
$ JAVA_OPTS=-Djsignpdf.swing=true bin/jsignpdf.sh
```

If command line arguments for signing are provided, JSignPdf runs in batch mode without opening

any GUI (see Command line (batch mode)).



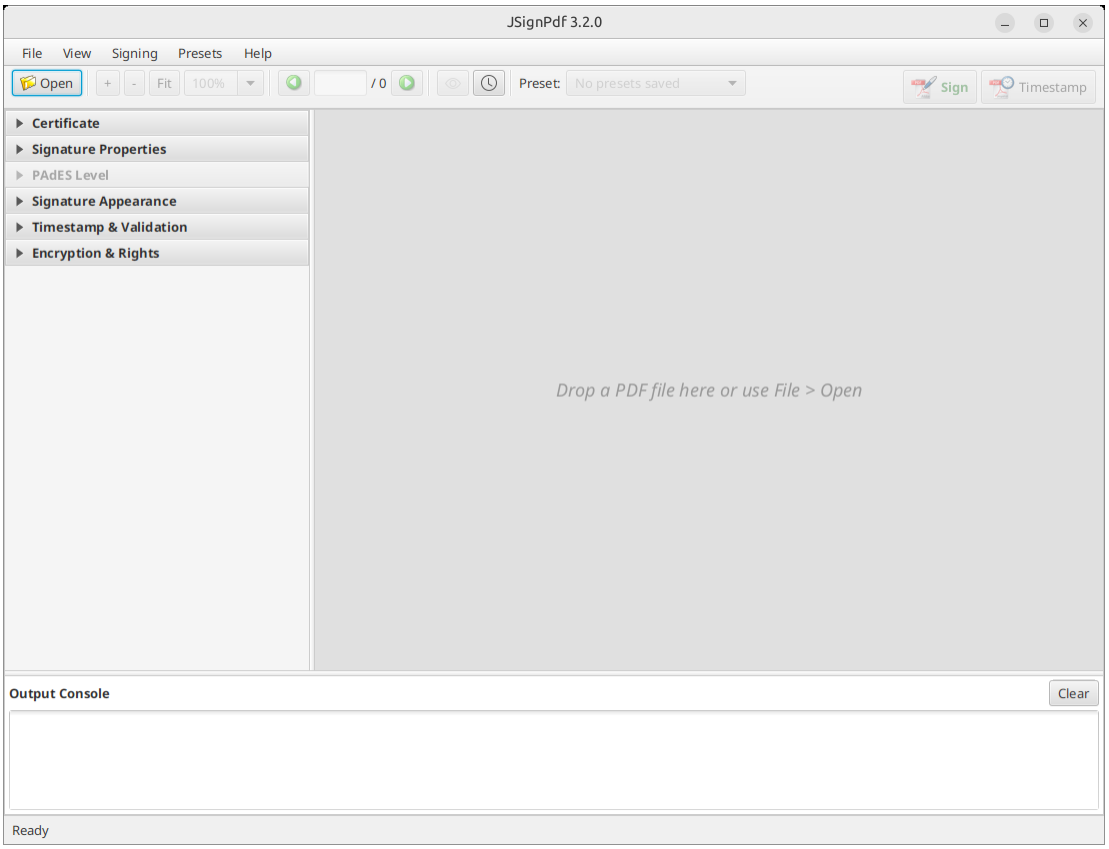
On Linux aarch64 the cross-platform ZIPs always start the Swing GUI — OpenJFX 21 does not publish a `linux-aarch64` native classifier on Maven Central. The Linux aarch64 DEB / RPM / Flatpak builds are unaffected because their bundled Azul Zulu+FX runtime supplies JavaFX as JDK modules.

Using the JavaFX UI

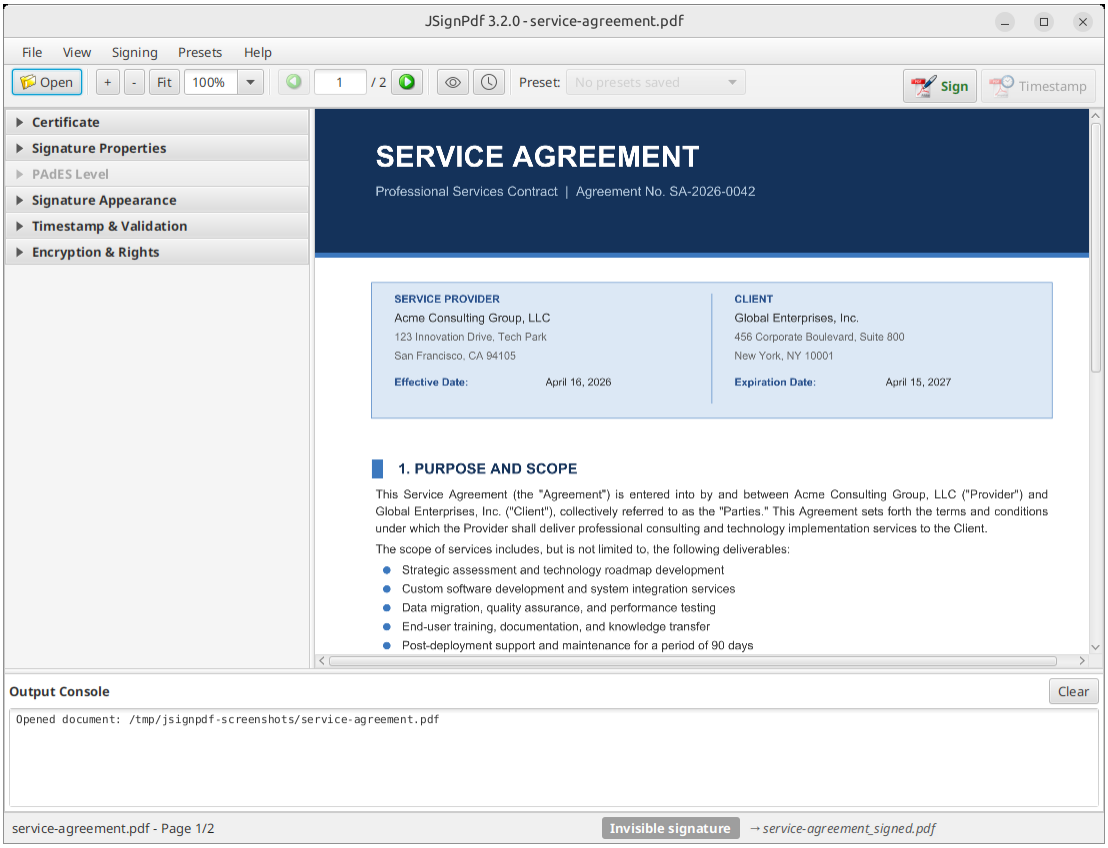
The JavaFX interface is the default graphical frontend for JSignPdf. It uses a document-centric workflow: you open a single PDF, configure the signature, and sign it. For signing multiple files in one go, use the command line.

Opening a PDF document

When JSignPdf starts, the main window is empty and invites you to load a document. You can either drag and drop a PDF file onto the window or use *File* > *Open* (or the *Open* button in the toolbar).



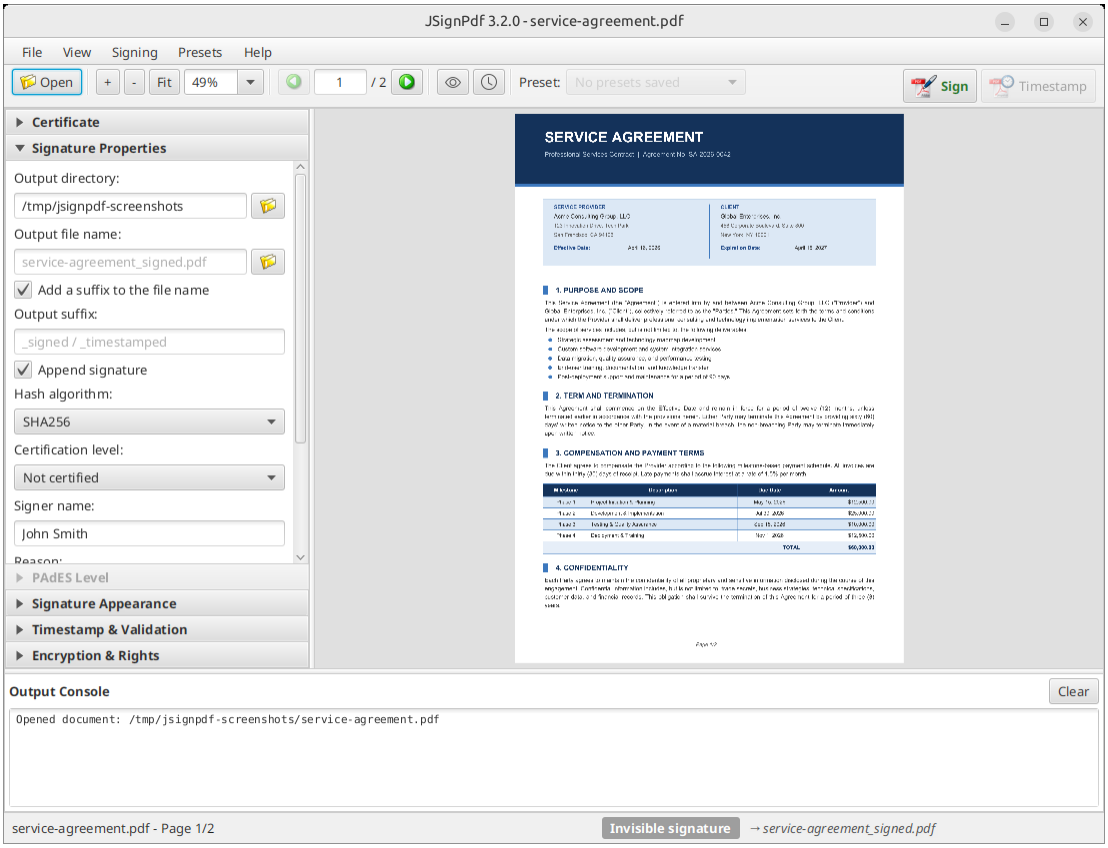
Once a PDF is loaded, the document preview appears on the right, and the configuration panels on the left become active. The status bar at the bottom shows the file name and page count.



Configuring signature options

The left sidebar contains five collapsible panels that expose all signing options. Expand a panel by clicking its header.

- **Certificate** — keystore type, keystore file & password, key alias & password (see Keystore selection and Key alias & key password)
- **Signature Properties** — hash algorithm, certification level, append mode, reason / location / contact, input & output file paths (see Signing properties)
- **Signature Appearance** — visible signature toggle, render mode, texts, images, font size (see Visible signature options)
- **Timestamp & Validation** — TSA server, CRL, OCSP (see TSA — timestamps and Certificate revocation checking)
- **Encryption & Rights** — encryption mode, passwords, certificate, document rights (see Encryption)



At a minimum, you need to select a keystore and provide its password in the *Certificate* panel. All other options have sensible defaults.

Placing a visible signature

To add a visible signature to the PDF, enable *Visible signature* in the *Signature Appearance* panel. You can then configure the render mode, signature text, status text, images, and font size.

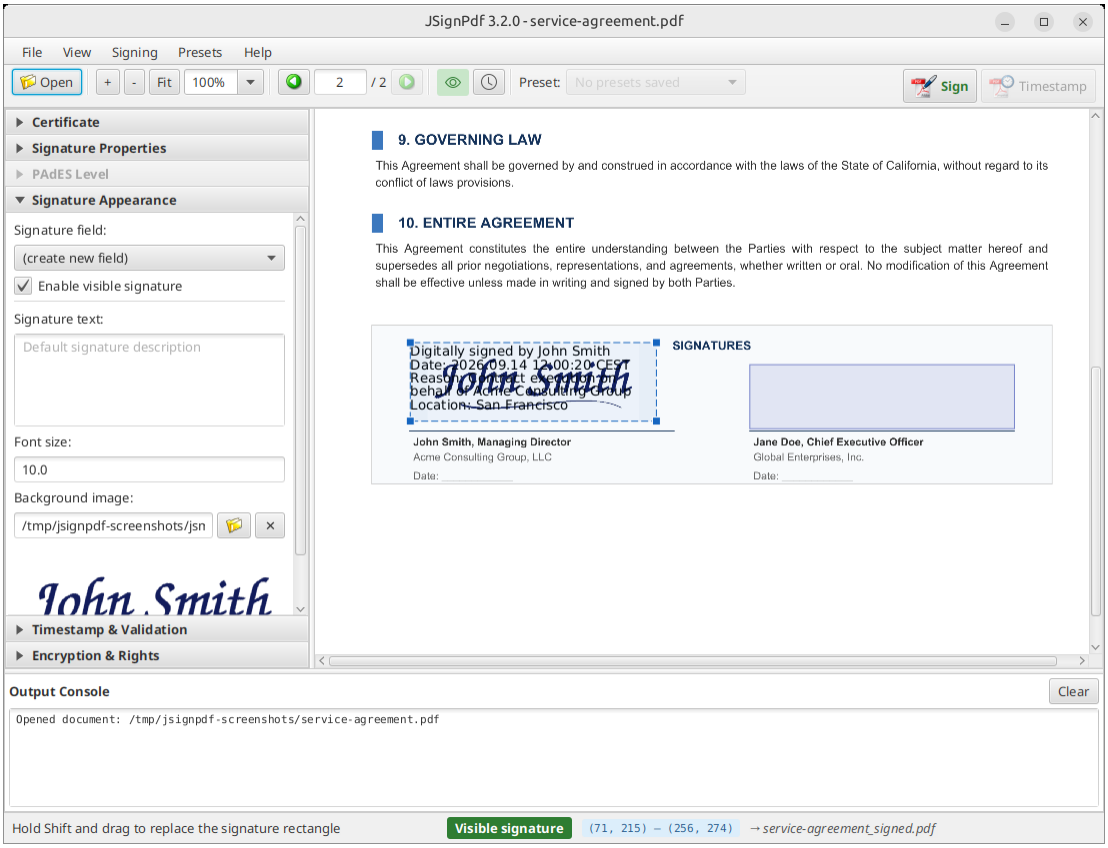
To create the signature rectangle, hold **Shift** and drag directly on the document preview. The coordinates are updated automatically in the sidebar.

Once the signature rectangle is visible, it can be adjusted directly in the preview:

- drag inside the rectangle to move the signature;
- drag a corner handle to resize it.

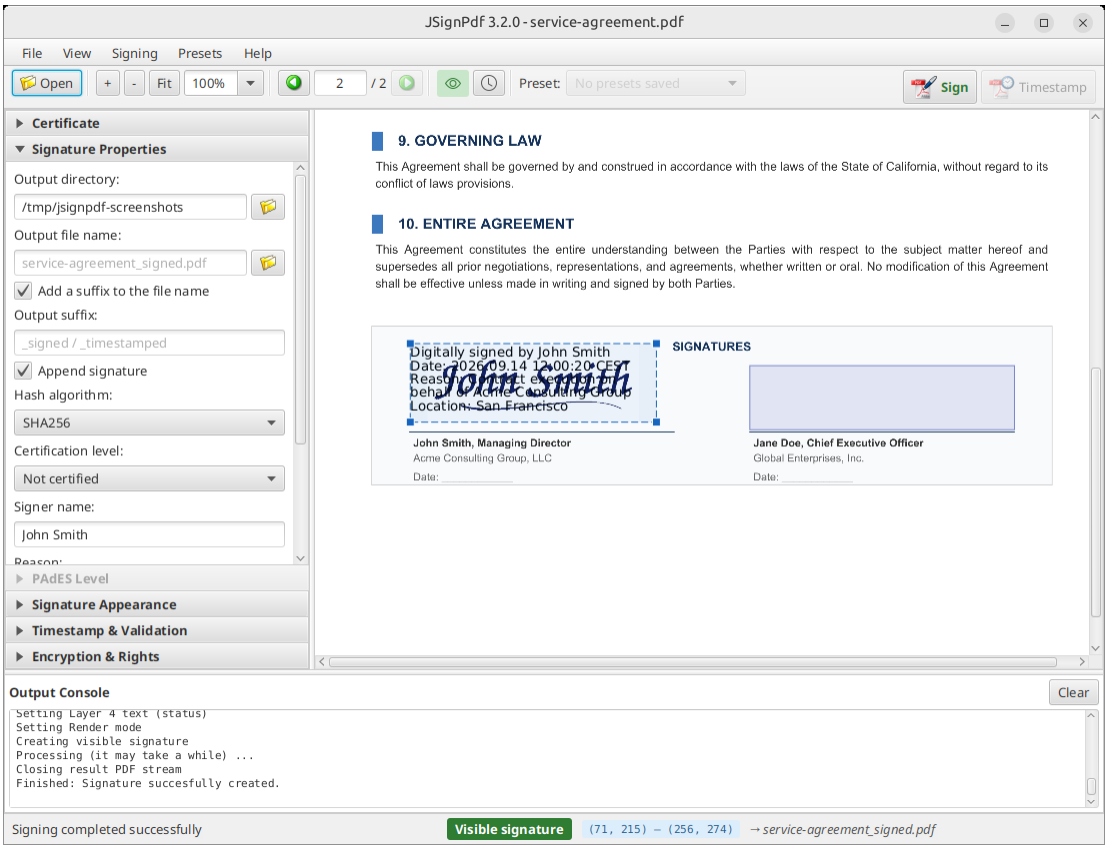
The mouse cursor changes according to the available operation. The preview also renders the configured signature text and images inside the rectangle, making it easier to position and size the signature before signing.

The preview reproduces the appearance of the OpenPDF engine. It is an approximation of the final result: the DSS engine wraps and insets the signature text slightly differently, so with that engine the preview shows the right content and placement, but the text may not break across lines exactly as it does in the signed document.



Signing the document

When you are ready, click the *Sign* button in the toolbar. JSignPdf signs the document and displays the result in the Output Console at the bottom of the window.



The signed PDF is saved to the output path shown in the *Signature Properties* panel. By default, the

input file name is combined with the *Output suffix* from the same panel, so `document.pdf` becomes `document_signed.pdf`.

The suffix is used exactly as typed, so it has to include the separator: `signed` produces `document_signed.pdf` and `-DL` produces `document-DL.pdf`. Because it becomes part of a file name, the field does not accept the path separators or reserved characters `/ \ : * ? " < > |` (or leading/trailing spaces), and is limited to 40 characters. Leaving the field empty falls back to the configured default, which the field shows as its prompt text. The starting value comes from the `output.suffix` key, set by the **Output file suffix** field in the Preferences dialog; changing it in the *Signature Properties* panel overrides it for the current document, is remembered for the next session, and is stored in a preset along with the other signing settings. Typing a full path into *Output file* always wins over the suffix.

Using presets

A **preset** is a named bundle of signing settings that you can save once and reload with a single click. Presets are useful when you regularly switch between different configurations — for example, one with a timestamp authority enabled and one without, or different visible-signature layouts for different document types.

Presets capture everything in the Signing options reference (keystore, signature properties, visible-signature options, TSA, encryption and rights, OCSP/CRL, proxy). Passwords are included only if *Store passwords* is enabled at the time you save the preset; in that case they are encrypted with a machine-local seed tied to the current user account and are silently ignored on load if the preset file is used under a different `user.home`. If you want a preset without stored passwords, disable *Store passwords* before saving — no password keys are written to the file in that case. Loading a preset without passwords also clears any passwords already entered in the UI, so switching to a password-less preset never silently reuses credentials from an earlier selection. The currently loaded input and output file paths are not part of a preset.

Loading a preset

The *Preset* drop-down in the toolbar shows every saved preset, sorted by name. Picking an entry immediately loads its settings into the UI — no confirmation, no extra click. After the load, the drop-down returns to its *Load preset…* placeholder. There is no "currently active preset" tracked anywhere: the drop-down is just a launcher, and whatever you do after loading (edits, saving as a new preset, signing) is independent of the preset you came from.

If no presets exist yet, the drop-down is disabled and labelled *No presets saved*.

Saving a new preset

Choose *Presets* > *Save current as new preset…* from the menu bar. Enter a name (up to 60 characters) and confirm. The preset appears in the toolbar drop-down immediately.

Managing existing presets

Choose *Presets* > *Manage presets…* to open the management dialog. Each row offers three actions:

- **Rename** — change the display name. The underlying file is not renamed; the name is stored

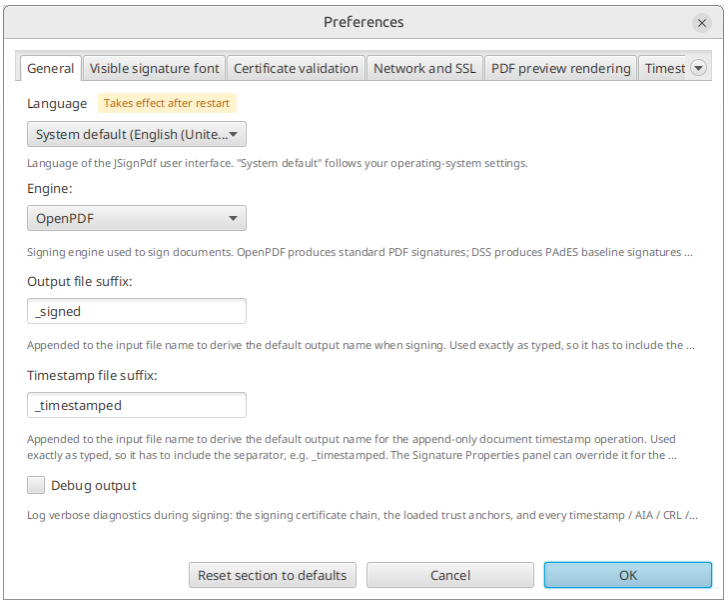
inside the file.

- **Overwrite** — replace the preset's stored settings with the current live settings. The preset's name and creation date are preserved. Asks for confirmation because the previous contents are lost.
- **Delete** — remove the preset file. Asks for confirmation.

Presets are plain Java properties files under the configuration directory.

Adjusting advanced preferences

Open *File* > *Preferences…* (shortcut `Ctrl+,` / `⌘,`) for application-wide tweaks that fall outside the per-document signing options.



The dialog has the following tabs:

- **General** — the interface language, the default signing engine, the **Output file suffix** used to derive output file names, and the **Debug output** toggle that enables the verbose signing diagnostics (see Reading the signing diagnostics)
- **Visible Signature** — font file, font name and encoding for the visible-signature L2 text
- **Certificates** — which certificate checks (validity, key usage, critical extensions) are applied when listing aliases
- **Network** — relax SSL security for TSA / OCSP / CRL traffic
- **PDF Render** — order and selection of preview backends (JPedal, Apache PDFBox, OpenPDF)
- **TSA** — default hash algorithm for timestamp requests
- **PKCS#11** — raw text of the SunPKCS11 provider configuration (see Using hardware tokens for signing)

Most settings apply immediately on *OK*. The two tabs marked *Takes effect after restart* — Network and PKCS#11 — only take effect on the next launch. *Reset section to defaults* restores the bundled defaults for the active tab only; other tabs keep their current values.

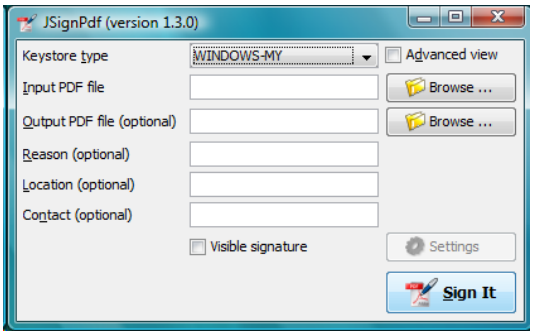
The values persist in `<config-dir>/advanced.properties` (see Advanced application configuration).

Classic Swing UI

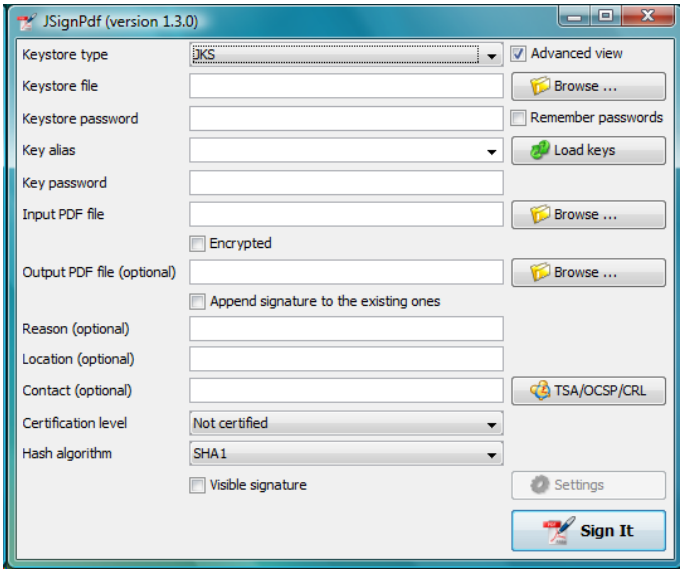
The classic Swing GUI is the original graphical interface from earlier versions of JSignPdf. It remains available for backward compatibility. From a native installer use the `JSignPdf-swing` launcher; from a cross-platform ZIP set the `jsignpdf.swing` system property:

```
$ JAVA_OPTS=-Djsignpdf.swing=true bin/jsignpdf.sh
```

The Swing GUI opens in a simplified view. Fill in the keystore type, keystore file and password, the input PDF file, and press *Sign It*.



Select the *Advanced view* checkbox to access additional options such as key alias, append signature, certification level, hash algorithm, encryption, visible signature, TSA, and certificate revocation settings.



All options available in the Swing GUI correspond to the options described in the Signing options chapter.

Command line (batch mode)

JSignPdf can be used in batch mode to sign one or more PDF files from the command line. This is useful for scripting and for signing multiple files at once.

The options are grouped by category below. For detailed descriptions of each option, see the Signing options chapter.

```
usage: jsignpdf [file1.pdf [file2.pdf ...]] [-a] [--bg-path
       <file>] [--bg-scale <scale>] [-c <contact>] [-cl <level>] [--crl] [-d
       <path>] [--disable-acrobat6-layer-mode] [--disable-assembly]
       [--disable-copy] [--disable-fill] [--disable-modify-annotations]
       [--disable-modify-content] [--disable-screen-readers] [-e] [-ec <file>]
       [--enable-stdin-passwords] [-eng <engine>] [-fs <size>] [--gui] [-h]
       [-ha <algorithm>] [--img-path <file>] [-ka <alias>]
       [-ki <index>] [-kp <password>] [-ksf <file>] [-ksp <password>] [-kst
       <type>] [-l <location>] [--l2-text <text>] [--l4-text <text>] [-le] [-lk]
       [-lkt] [-llx <position>] [-lly <position>] [-lp] [-lpf <file>] [-o
       <key=value>] [--ocsp]
       [--ocsp-server-url <responderUrl>] [-op <prefix>] [-opwd <password>] [-os
       <suffix>] [--overwrite] [-pe <mode>] [-pg <pageNumber>] [-pr <right>] [--proxy-
host
       <hostname>] [--proxy-port <port>] [-pl <level>] [--proxy-type <type>] [-q] [-r
       <reason>] [--render-mode <mode>] [-sn <signer>] [-ta <method>] [-ts <URL>]
       [--tsa-policy-oid <policyOID>] [-tscf <file>] [-tscp <password>] [-tsct
       <ks-type>] [-tsh <algorithm>] [-tso] [-tsp <password>] [-tsu <username>]
       [-upwd <password>] [-urx <position>] [-ury <position>] [-v] [-V]
```

General options

Option	Description
`-h, --help`	Prints the help screen.
`-v, --version`	Shows the application version.
`-q, --quiet`	Quiet mode — suppresses info messages during signing.
`--gui`	(Beta) Show the GUI even when other command line arguments are provided.
`-lkt, --list-keystore-types`	Lists available keystore types (use as values for `-kst`).
`-lk, --list-keys`	Lists keys in the chosen keystore.
`-le, --list-engines`	Lists the available signing engines (use as values for `-eng`). See Signing engines.

Option	Description
`-lsf, --list-sig-fields`	Lists the signature fields of the given PDF file(s) — number, name, page, rectangle and whether the field is still empty — and exits. The numbers are the ones `--sig-field #N` accepts. See Signing an existing signature field.
`-eng, --engine <engine>`	Selects the signing engine for this invocation, overriding the `engine` key in `advanced.properties`. Use `-le` to list available engines. See Signing engines.
`-lp, --load-properties`	Loads properties from the default file (created by the GUI application).
`-lpf, --load-properties-file <file>`	Loads properties from the given file. The file can be created by copying the default `config.properties` from the Configuration directory.
`-o, --option <key=value>`	Sets a single `advanced.properties` key for this invocation, overriding the value from the configuration file (and the bundled default). Repeatable. The value may contain `=` (only the first one is the separator). Useful for the `engine.dss.*` trust knobs — see PAdES & the DSS engine. Note: values are visible in the process list and shell history, so prefer the configuration file for secrets such as truststore passwords.
`--enable-stdin-passwords`	Allow reading password values from standard input. When set, a password option value of `-` means "read one line from stdin" (or prompt, on an interactive console). See Reading passwords from standard input.

Keystore and key options

See Keystore selection and Key alias & key password for details.

Option	Description
`-kst, --keystore-type <type>`	Sets the keystore type. Use `-lkt` to list available types.
`-ksf, --keystore-file <file>`	Path to the keystore file (`.p12`, `.pfx`, `.jks`, ...). Not needed for keystores without files (e.g. `WINDOWS-MY`).
`-ksp, --keystore-password <password>`	Password for the keystore. Use `-` to read from stdin (requires `--enable-stdin-passwords`).
`-ka, --key-alias <alias>`	Name (alias) of the key to use for signing. If not given, the first key in the keystore is used. Use `-lk` to list available aliases.
`-ki, --key-index <index>`	Zero-based index of the key to use. Lower priority than `-ka`.
`-kp, --key-password <password>`	Password for the individual key (if different from the keystore password). Use `-` to read from stdin (requires `--enable-stdin-passwords`).

Signature options

See Signing properties for details.

Option	Description
`-a, --append`	Append the signature to existing ones (incremental update). This is the default since JSignPdf 3.1, so the flag is kept only for backward compatibility and is now a no-op.
`--overwrite`	Replace existing signatures by rewriting the document (non-incremental). Opt-out of the default append behaviour. Rejected by the DSS (PAdES) engine, which always signs incrementally — see Signing engines.
`-ha, --hash-algorithm <algorithm>`	Hash algorithm for the signature. Default: `SHA256`. Values: `SHA1`, `SHA256`, `SHA384`, `SHA512`, `RIPEMD160`. (SHA-1 remains selectable for the OpenPDF engine; the DSS engine accepts only `SHA256`/`SHA384`/`SHA512`.)
`-cl, --certification -level <level>`	Certification level. Default: `NOT_CERTIFIED`. Values: `NOT_CERTIFIED`, `CERTIFIED_NO_CHANGES_ALLOWED`, `CERTIFIED_FORM_FILLING`, `CERTIFIED_FORM_FILLING_AND_ANNOTATIONS`.
`-pl, --pades-level <level>`	PAdES baseline level for the PAdES (`dss`) signing engine. Values (case-insensitive): `B`, `T`, `LT`, `LTA`. Honoured only by engines that support PAdES; the default OpenPDF engine rejects it (use `-eng dss`). `LT`/`LTA` require revocation data — see PAdES & the DSS engine. See Signing engines.
`-sn, --signer-name <signer>`	Signer name. Defaults to the common name (CN) of the chosen certificate.
`-r, --reason <reason>`	Reason for the signature. Empty by default.
`-l, --location <location>`	Location of the signature (e.g. Washington DC). Empty by default.
`-c, --contact <contact>`	Signer's contact details.

Output options

Option	Description
`-d, --out-directory <path>`	Folder for signed documents. Default: current folder.
`-op, --out-prefix <prefix>`	Prefix for the signed filename. Default: empty.
`-os, --out-suffix <suffix>`	Suffix for the signed filename, used exactly as given. Default: the `output.suffix` key in `advanced.properties` (bundled default `_signed`), so `mydocument.pdf` becomes `mydocument_signed.pdf`. With `--timestamp-only` the default comes from `output.suffix.timestamp` (bundled default `_timestamped`) instead.

Visible signature options

See Visible signature options for details.

Option	Description
`-V, --visible -signature`	Enables a visible signature. Implied by `-sf`.
`-sf, --sig-field <field>`	Signs into an existing empty signature field instead of creating a new one: a field name, `#N` for the N-th signature field in document order (see `-lsf`) or `auto` for the first empty field. The field's own rectangle and page are used, so the position options below are ignored. See Signing an existing signature field.
`-pg, --page <pageNumber>`	Page for the visible signature. Default: `1`. If out of bounds, the last page is used. Ignored when `-sf` is used.
`-llx <position>`	Lower left corner X-axis position (negative value is subtracted from page width).
`-lly <position>`	Lower left corner Y-axis position (negative value is subtracted from page height).
`-urx <position>`	Upper right corner X-axis position (negative value is subtracted from page width).
`-ury <position>`	Upper right corner Y-axis position (negative value is subtracted from page height).
`--render-mode <mode>`	Render mode. Default: `DESCRIPTION_ONLY`. Values: `DESCRIPTION_ONLY`, `GRAPHIC_AND_DESCRIPTION`, `SIGNAME_AND_DESCRIPTION`.
`--l2-text <text>`	Signature text. Supports placeholders: `${signer}`, `${certificate}`, `${timestamp}`, `${location}`, `${reason}`, `${contact}`. The timestamp can optionally use a custom display format, for example `${timestamp:yyyy.MM.dd}` or `${timestamp:dd/MM/yyyy HH:mm}`.
`--l4-text <text>`	Status text.
`-fs, --font-size <size>`	Font size for signature text. Default: `10.0`.
`--img-path <file>`	Image path for the visible signature.
`--bg-path <file>`	Background image path for the visible signature.
`--bg-scale <scale>`	Background image scale. Positive: multiply size. Zero: stretch to fill. Negative: best-fit resize.
`--disable-acrobat6 -layer-mode`	Disable Acrobat 6 layer mode (creates all signature layers instead of just n2 and n4).

Encryption and rights options

See Encryption for details.

Option	Description
`-pe, --encryption <mode>`	Encryption mode. Default: `NONE`. Values: `NONE`, `PASSWORD`, `CERTIFICATE`. Use together with `-upwd` and `-opwd` for `PASSWORD` mode, or `-ec` for `CERTIFICATE` mode.
`-e, --encrypted`	*Deprecated.* Use `-pe PASSWORD` instead.
`-opwd, --owner -password <password>`	Owner password for encrypted documents. Use `-` to read from stdin (requires `--enable-stdin-passwords`).
`-upwd, --user-password <password>`	User password for encrypted documents. Use `-` to read from stdin (requires `--enable-stdin-passwords`).
`-ec, --encryption -certificate <file>`	Path to certificate file for `CERTIFICATE` encryption.
`-pr, --print-right <right>`	Printing rights. Default: `ALLOW_PRINTING`. Values: `DISALLOW_PRINTING`, `ALLOW_DEGRADED_PRINTING`, `ALLOW_PRINTING`.
`--disable-copy`	Deny copy in encrypted documents.
`--disable-assembly`	Deny assembly in encrypted documents.
`--disable-fill`	Deny fill in encrypted documents.
`--disable-modify -annotations`	Deny modifying annotations in encrypted documents.
`--disable-modify -content`	Deny modifying content in encrypted documents.
`--disable-screen -readers`	Deny screen readers in encrypted documents.

TSA and certificate validation options

See TSA — timestamps and Certificate revocation checking for details.

Option	Description
`-ts, --tsa-server-url <URL>`	Address of the timestamping server (TSA). Enables timestamping.
`-ta, --tsa -authentication <method>`	TSA authentication method. Default: `NONE`. Values: `NONE`, `PASSWORD`, `CERTIFICATE`.
`-tsu, --tsa-user <username>`	TSA username (for `PASSWORD` authentication).
`-tsp, --tsa-password <password>`	TSA password (for `PASSWORD` authentication). Use `-` to read from stdin (requires `--enable-stdin-passwords`).
`-tscf, --tsa-cert-file <file>`	Path to keystore file with private key for TSA `CERTIFICATE` authentication.
`-tscp, --tsa-cert -password <password>`	Password for the TSA certificate keystore. Use `-` to read from stdin (requires `--enable-stdin-passwords`).
`-tsct, --tsa-cert-file -type <ks-type>`	Keystore type for TSA certificate authentication. Default: `PKCS12`.

Option	Description
`-tso, --timestamp-only`	Append a document timestamp (ETSI.RFC3161) instead of signing — no key and no signature. Requires `-ts` and the DSS engine (`-eng dss`). See Adding a document timestamp.
`-tsh, --tsa-hash -algorithm <algorithm>`	Hash algorithm for TSA queries. Default: `SHA-1`.
`--tsa-policy-oid <policyOID>`	TSA policy OID to include in the timestamp request.
`--ocsp`	Enable OCSP certificate validation.
`--ocsp-server-url <responderUrl>`	Default OCSP server URL (used when the signing certificate doesn't contain one).
`--crl`	Enable CRL certificate validation.

Proxy options

See Proxy settings for details.

Option	Description
`--proxy-type <type>`	Proxy type. Default: `DIRECT`. Values: `DIRECT`, `HTTP`, `SOCKS`.
`--proxy-host <hostname>`	Hostname or IP address of the proxy server.
`--proxy-port <port>`	Port of the proxy server. Default: `80`.

Program exit codes

Code	Meaning
0	Program finished without errors.
1	Command line is in a wrong format.
2	No operation requested — e.g. no file for signing provided.
3	Signing of some, but not all, files failed.
4	Signing of all files failed.
5	Input file cannot be read.
6	Common error.

Examples

Sign a PDF using the Windows certificate store:

```
$ jsignpdf -kst WINDOWS-MY mydocument.pdf
```

Sign and encrypt a PDF, disabling printing:

```
$ jsignpdf -kst PKCS12 -ksf my_certificate.pfx \
  -ksp myPrivateKeystorePassword -ka cert23 \
  -pe PASSWORD -opwd xxx123 -upwd 123xxx \
  -pr DISALLOW_PRINTING mydocument.pdf
```

List available keystore types:

```
$ jsignpdf -lkt
```

List key aliases in a keystore (quiet mode):

```
$ jsignpdf -kst PKCS12 -ksf my_certificate.pfx \
  -ksp myVeryPrivatePassword -lk -q
```

Reading passwords from standard input

Passing a password on the command line exposes it to every local process that can read `/proc/<pid>/cmdline` (and to shell history). To avoid that, JSignPdf can read password values from standard input (or, in an interactive terminal, from the console) instead.

Opt in with `--enable-stdin-passwords` and use `-` as the value of any password option. Any of the six password options can use the sentinel: `-ksp`, `-kp`, `-opwd`, `-upwd`, `-tscp`, `-tsp`.

Single password, piped from the environment

```
$ printf '%s\n' "$KS_PASSWORD" \
  | jsignpdf --enable-stdin-passwords \
      -kst PKCS12 -ksf keystore.p12 -ksp - -ka mykey \
      -d out/ input.pdf
```

When more than one option uses `-`, the values are consumed from stdin in a **fixed canonical order** regardless of the order in which they appear on the command line:

1. `--keystore-password` (`-ksp`)
2. `--key-password` (`-kp`)
3. `--owner-password` (`-opwd`)
4. `--user-password` (`-upwd`)
5. `--tsa-cert-password` (`-tscp`)
6. `--tsa-password` (`-tsp`)

Keystore password first, then TSA password — even though the CLI lists them in the other order

```
$ { printf '%s\n' "$KS_PASSWORD"; printf '%s\n' "$TSA_PASSWORD"; } \
  | jsignpdf --enable-stdin-passwords \
      -kst PKCS12 -ksf keystore.p12 -ksp - -ka mykey \
      -ts https://tsa.example/ -tsu tsauser -tsp - \
      -d out/ input.pdf
```

Before blocking on each read, JSignPdf prints a progress line such as `[jsignpdf] Reading password for --keystore-password (1/2) from stdin…` to *stderr* so you can verify the order matches what your pipe is feeding. Use `-q` / `--quiet` to suppress it.

When a console is attached (an interactive terminal), `-` switches to a prompted, no-echo read via `java.io.Console.readPassword`:

```
$ jsignpdf --enable-stdin-passwords \
      -kst PKCS12 -ksf keystore.p12 -ksp - -ka mykey input.pdf
Enter password for --keystore-password: ****
```



Without `--enable-stdin-passwords`, a literal `-` is still accepted as a password value (backwards compatible). JSignPdf will emit a one-time warning naming the flag, so a typo does not silently turn into a `-` password. The warning is suppressed by `-q`.



Only exactly `-` is the stdin sentinel — values such as `-abc` or `abc-` are treated as literal passwords. Note that Apache Commons CLI intercepts a bare `--` as the end-of-options marker before JSignPdf sees it, so `--` itself cannot be used as a literal password via `-ksp --`; this is a parser limitation.

Signing options

This chapter describes all signing options in detail. The options are the same regardless of which interface you use (JavaFX GUI, Swing GUI, or command line). In the JavaFX interface the options are organized into sidebar panels; the corresponding panel name is noted in each section header.

Keystore selection

JavaFX panel: Certificate

The Keystore means the location where the private keys are located.

By default, JSignPdf displays keystore types provided by Java Runtime itself and the Bouncy Castle cryptographic provider.

JSignPdf has been extended to support remote/external keystore types. The first entry is "CloudFoxy" (https://gitlab.com/cloudfoxy), which is a REST API for physical smart cards, initially developed to support eIDAS signatures.

If you use PKCS#12 or Java keystore types (JKS, JCEKS), you have to select the file where the keys are stored and provide the password of this file. Path to the keystore file can be inserted directly by typing or you can use the browse button to navigate through the file system with Open File Dialog.

Supported key types

Both bundled engines sign with RSA and EC (ECDSA) keys, and derive the signature algorithm from the key itself — there is nothing to configure.

RSASSA-PSS needs the `dss` engine. A PSS-only certificate (one whose public key is marked `id-RSASSA-PSS` rather than `rsaEncryption`, increasingly common for eIDAS qualified certificates) produces a conformant PSS signature there, including when the key lives on a PKCS#11 token that reports it as a plain RSA key. The `openpdf` engine cannot encode the PSS parameters a signature needs, so sign with `-eng dss` for such certificates.

For EC keys the two engines differ in one detail: the `dss` engine records the signature algorithm as `ecdsa-with-SHA256`, while `openpdf` records `id-ecPublicKey`, which some strict validators may not accept. Prefer `-eng dss` for EC signing.

Key alias & key password

JavaFX panel: Certificate

When you have more private keys stored in the keystore, you can select which one will be used to sign the PDF file by filling the *Key alias* field. Either you can type alias name directly (combo box is editable) or you can load all names by pressing the *Load keys* button and then select one from the drop-down list.[1]

If you don't fill the *Key alias* field the first alias read from keystore will be used.

Each key in the keystore can be protected with its password. If this password differs from the password of keystore, fill it in the *Key password* input field.

Signing properties

JavaFX panel: Signature Properties

Input and output PDF files

Input PDF file is the existing PDF file to which a digital signature will be added.

Output PDF file is the name of the resulting PDF file. If not specified, the *Input PDF file* name with the suffix `signed is used automatically (e.g. input _test.pdf` will produce *test_signed.pdf*). The suffix is taken from the `output.suffix` key in `advanced.properties`, so it can be localized (see Advanced application configuration). The Swing UI has no suffix field of its own.

The Input and Output files have to be different!

Reason, location, contact

The reason, location, and contact fields provide additional information about the signature. Filled values will be stored in the result PDF.

Append signature

JSignPdf can work in two signing modes. When *Append signature* is enabled, the new signature is appended and any previously-existing signatures stay unchanged; when it is disabled, the existing signatures are replaced by rewriting the document. Append is the default in both the GUI and — since JSignPdf 3.1 — the CLI; on the command line, pass `--overwrite` to replace instead (the legacy `-a` / `--append` flag is now a no-op, kept for backward compatibility). The DSS (PAdES) engine always signs incrementally and rejects `--overwrite`. **This option is disabled for encrypted documents.**

Certification level

The JSignPdf application can add a certificate to the signed PDF. There are four levels of certification:

- NOT_CERTIFIED
- CERTIFIED_NO_CHANGES_ALLOWED
- CERTIFIED_FORM_FILLING
- CERTIFIED_FORM_FILLING_AND_ANNOTATIONS

Hash algorithms

You can choose which hash function will be used for the signature. Available algorithms include SHA-256, SHA-384, SHA-512, and others.

Remember passwords

JSignPdf stores filled information when you are exiting the application, so it's present when you run it the next time. Passwords are stored (encrypted) when the *Remember passwords* option (labelled *Store passwords* in the JavaFX GUI) is enabled. The option is enabled by default; uncheck it if you prefer to re-enter passwords on every launch.

Even if the password is stored in the encrypted form, we do not recommend storing passwords if your computer is used by more users!

Encryption

JavaFX panel: Encryption & Rights

The *Encryption* option enables additional fields for support of PDF security. By using this you can either sign secured PDFs (and change the rights and user password) or you can add encryption to unencrypted PDF during the signing.

Encryption: Passwords

Fill owner and user passwords to set it in secured result PDF. If the input PDF is encrypted, the *Owner password* field has to match to owner password of the input PDF.

Encryption: Certificate

Fill the path to a certificate file (*.cer, *.crt, ...) which should be used for the PDF encryption. Only the user who has the private key for the certificate will be able to open the file.

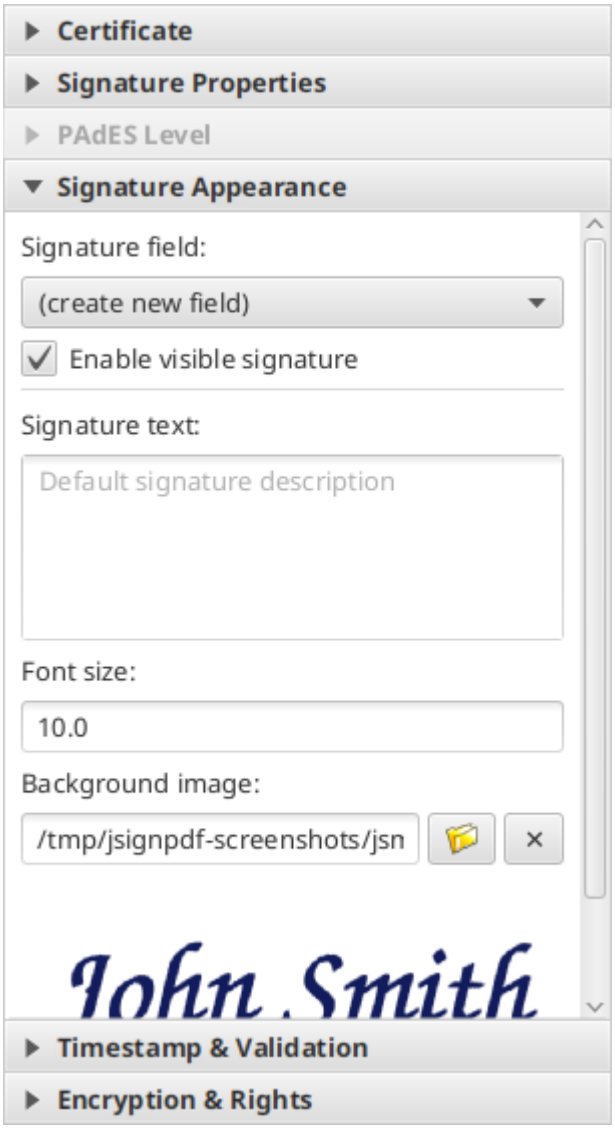
Rights

You can set allowed actions in encrypted result PDF by configuring the document rights. Normal rights are represented by checkboxes. Printing right has 3 levels, so the combo box is used for it.

Visible signature options

JavaFX panel: Signature Appearance

Enabling the *Visible signature* option allows you to create a visible field with signature directly in the signed PDF. You can configure parameters (position/texts/images) of the visible signature.



Page

Page number (counted from 1) to which the signature will be added.

Signature corners

The four inputs *Lower Left (X, Y)* and *Upper Right (X, Y)* define the position of the signature on the page. You can fill in float numbers (with decimal places) as input. Negative values are subtracted from the page width (for X) or height (for Y), allowing you to position the signature relative to the right or top edge.

The zero ([0,0]) position on the page is in the left bottom corner.

In the JavaFX interface you can create the signature rectangle visually by holding **Shift** and dragging on the document preview.

After the rectangle has been created, it can be edited directly with the mouse. Drag inside the rectangle to move it, or drag one of its corner handles to resize it. The signature coordinates are updated automatically while the rectangle is moved or resized.

Display

In the *Display* / *Render mode* option you can set which fields will be generated to visible signature.

Acrobat 6 layers

The *Acrobat 6 layer mode* option (enabled by default) allows you to control which signature layers will be added to the signed document. Acrobat 6.0 and higher recommends that only layers n2 and n4 be present. If the option is disabled then all layers will be created.

Texts and Images

Signature Text, *Status Text*, *Image*, and *Background Image* inputs define the content of fields in a visible signature. *Font Size* is used for setting the size of *Signature Text*, it should contain a positive decimal number.

The *Signature Text* supports placeholders that are expanded when the document is signed:

Placeholder	Expands to
`${signer}`	Common name (CN) of the signing certificate.
`${timestamp}`	Signing date and time using the default display format.
`${timestamp:<pattern>}`	Signing date and time using a custom Java date/time pattern, for example `${timestamp:yyyy.MM.dd}`.
`${certificate}`	Description of the signing certificate.
`${location}`	The *Location* field value.
`${reason}`	The *Reason* field value.
`${contact}`	The *Contact* field value.

For example, a *Signature Text* of `Signed by ${signer}` + `on ${timestamp}` renders the signer name and the signing date/time.

A custom timestamp format can be specified after a colon. Examples:

```
${timestamp:yyyy.MM.dd}
${timestamp:dd.MM.yyyy}
${timestamp:yyyy-MM-dd}
${timestamp:dd/MM/yyyy HH:mm}
${timestamp:yyyy-MM-dd HH:mm:ss}
```

Common pattern letters include `yyyy` for the four-digit year, `MM` for month, `dd` for day, `HH` for hour in 24-hour format, `mm` for minutes, and `ss` for seconds. Note that `MM` means month while `mm` means minutes.

Custom timestamp formatting changes only the date/time text shown in the visible signature. It does not change or configure the cryptographic timestamp produced by a timestamp authority (TSA).

When *Signature Text* is left empty, JSignPdf uses a default layout that already includes the signer and the date.

Background image scale defines the size of a background image. Any negative number means the best-fit algorithm will be used. Zero value means to stretch, which fills the whole field — it doesn't keep the image ratio. A positive value means the multiplicator of the original size.

Supported file formats for the *Image* and *Background Image* are GIF, JPEG, JPEG2000, PNG, WMF, BMP, and TIFF.

Signing an existing signature field

Documents that are meant to be signed by several people usually carry the signature boxes already: the form author pre-places an empty signature field for each signer ("Signature #1", "Signature #2"), and every signer fills their own box. JSignPdf can sign into such a field instead of creating a new one. The bundled `demo/service-agreement.pdf` carries one, so you can try this out right away — see Demo files.

Empty signature fields are shaded on the page preview as soon as a document is opened, so you can see where the boxes are before picking one. JSignPdf draws the shading itself, independently of the preview backend in use; fields with a zero-size or hidden widget are not shaded, because they are invisible in any viewer too.

In the JavaFX UI, the *Signature field* combo box at the top of the *Signature Appearance* panel lists the empty signature fields of the opened document. Pick one and the signature goes there; the selected field gets a stronger marker on the page preview and the position controls are switched off, because the field's own rectangle decides where the signature lands. The live appearance preview follows the selected field, so the signature text and image are rendered inside its rectangle just as they are for a rectangle you draw yourself. *Visible signature* is turned on and locked (in the panel, the toolbar and the menu alike) for as long as a field is selected, because the appearance is always drawn into the field — switch back to *(create new field)* if you want an invisible signature, or pick a field whose rectangle has zero size, which gives one anyway. The combo stays on *(create new field)* for documents without empty fields, and the selection is reset whenever you open another document.

On the command line, `-lsf` shows what a document offers:

```
$ jsignpdf -lsf contract.pdf
Signature fields of contract.pdf:
#1   Employee                      page 1    [70.0 700.0 300.0 760.0] blank
#2   Witness                       page 1    [320.0 700.0 500.0 760.0] signed
#3   Manager                       page 3    [70.0 500.0 300.0 560.0] blank
```

and `-sf` picks one of them:

```
$ jsignpdf -kst PKCS12 -ksf key.p12 -ksp secret -sf Manager contract.pdf
$ jsignpdf -kst PKCS12 -ksf key.p12 -ksp secret -sf '#3' contract.pdf     # the same
field, by number
```

```
$ jsignpdf -kst PKCS12 -ksf key.p12 -ksp secret -sf auto contract.pdf     # the first
empty field
```

Things worth knowing:

- **The numbers count all signature fields**, empty and signed alike, in document order (page by page). A number therefore keeps pointing at the same field as the other signers fill theirs.
- **A field name always wins over a selector.** If a document really contains a field named `auto` or `#1`, that field is what you get; `-lsf` marks such a name.
- **`#N` and `auto` are resolved per input file**, so the same command works for a batch of documents built from one template.
- **The field's rectangle and page are used**, so `-pg`, `--llx`, `--lly`, `--urx` and `--ury` are ignored (with a warning) — neither signing engine can move an existing field. The appearance is drawn into the field even without `-V`; a field with a zero-size rectangle gives an invisible signature.
- **A name that does not match an empty field is an error**, and the message lists what the document offers. JSignPdf never creates a field under a name you asked for, so a typo cannot silently produce a stray signature somewhere else on the page.
- **`--overwrite` and PDF encryption are refused** together with `-sf`. Both rewrite the document instead of appending to it, which contradicts signing a document the way its author built it — and a rewrite drops the signatures your co-signers already added. Sign first and encrypt the result afterwards.
- JSignPdf does not evaluate `/Lock` dictionaries or a previous certification signature. Signing a field that an earlier signer locked, or any field of a document certified with "no changes allowed", produces a technically valid signature that PDF readers will nonetheless report as invalidating the earlier one.

TSA — timestamps

JavaFX panel: Timestamp & Validation

To add a timestamp to the signature you will need a timestamping authority (TSA). Fill the server address into the *TSA URL* field and if the server requires authentication, choose the authentication type and fill either the *TSA User* and *TSA Password* fields or the path to the certificate's private key (it has to be a PKCS#12 keystore) and the password. You can also set *TSA Policy OID*, which will be sent to the TSA server in the request, but typically the server uses the right policy by itself.

Adding a document timestamp

A document timestamp (a `DocTimeStamp` with the `ETSI.RFC3161` subfilter) proves that the bytes of a PDF existed at a point in time. It is not a signature: no private key is involved, nothing is claimed about who the document belongs to, and no signature field is filled. JSignPdf appends one with the `--timestamp-only` flag:

```
$ jsignpdf -eng dss --timestamp-only -ts http://tsa.example/tsa contract.pdf
```

Two uses:

- **Timestamping a document that carries none** — signed or not. Any PDF can be given a trusted point in time this way.
- **Archival refresh ("digital continuity")** — a PAdES B-LTA signature stays verifiable only while its archive timestamp is; re-timestamp the document before that timestamp's TSA certificate expires, and the chain of evidence continues. Every refresh covers everything before it, so the document can be kept verifiable indefinitely by repeating this every few years.

The operation needs the **DSS engine** — add `-eng dss`, or set `engine=dss` in `advanced.properties` once. The OpenPDF engine cannot produce document timestamps and refuses the operation with a message naming the missing `DOC_TIMESTAMP` capability. `--tsa-server-url` is mandatory, and all the other TSA options (`--tsa-authentication`, `--tsa-user`, `--tsa-password`, `--tsa-policy-oid`, `--tsa-hash-algorithm`, the TSA certificate options and the proxy options) apply unchanged.

In the JavaFX UI the same operation is the *Timestamp* toolbar button and *Signing* > *Add Timestamp…* (`Ctrl+T` / `⌘T`). Both are enabled once a document is open and the DSS engine is selected; the certificate and appearance panels are not consulted. The *TSA URL* field is required regardless of the TSA toggle, which means "timestamp the signature I am about to make" — a different thing.

Things worth knowing:

- **The output is an incremental revision.** The original bytes are untouched, so existing signatures stay valid. `--overwrite` is therefore refused, as are `--sig-field` (no field is filled) and `--certification-level` (DocMDP is a property of a signature).
- **The output file name uses the `output.suffix.timestamp` suffix** (bundled default `_timestamped`), so `contract.pdf` becomes `contract_timestamped.pdf`. `-os`, `-d` and `-op` work as they do for signing.
- **Validation data for what is already in the document is collected when it can be.** For a document that carries a timestamped (T / LT / LTA) signature, JSignPdf first writes a DSS dictionary with fresh validation data and then appends the new timestamp — that is what makes an archival refresh a complete one. This needs `engine.dss.online.enabled=true` and trust material (see PAdES & the DSS engine).
- **No validity claim is made about the existing signatures.** An expired signer certificate, an untrusted chain or unreachable revocation data is reported as a warning and the timestamp is appended anyway — which is the point, since an archived document being refreshed usually has exactly those properties.
- **The newest timestamp is only covered by validation data at the next refresh.** That is how every long-term validation chain works, not a limitation of JSignPdf.
- **An encrypted input keeps its encryption.** Use `--owner-password` to open it; `--encryption` / `-e` are ignored (with a warning), because the incremental append preserves what the input already has.
- **Several TSAs, several runs.** Each run appends another timestamp, so stacking timestamps from different jurisdictions is just a matter of repeating the command:

```
$ jsignpdf -eng dss --timestamp-only -ts http://tsa-a.example/tsa contract.pdf
$ jsignpdf -eng dss --timestamp-only -ts http://tsa-b.example/tsa -os '' -d out
```

```
contract_timestamped.pdf
```

- Keystore, appearance, PAdES level, reason / location / contact and the OCSP / CRL flags are ignored (with a warning), so a properties file or preset made for signing can be reused as it is.

Certificate revocation checking

JavaFX panel: Timestamp & Validation

JSignPdf supports two standard ways of certificate revocation checking — CRL and OCSP. Most X.509 certificates support CRL, but it has some disadvantages (for instance the size of the list and possibly outdated information). OCSP solves these issues, but not all Certification Authorities (CA) support it.

CRL

A Certificate Revocation List (CRL) is a list of serial numbers for certificates that have been revoked or are no longer valid (RFC 5280). When CRL checking is enabled in JSignPdf, the revocation list is downloaded from the CA and embedded in the signed PDF.

OCSP

The Online Certificate Status Protocol (OCSP) allows real-time verification of a certificate's revocation status (RFC 6960). Unlike CRL, which requires downloading a potentially large list, OCSP queries the CA for the status of a single certificate.

If OCSP is enabled in JSignPdf and the protocol is supported for the certificate, the OCSP request will be created and the response will be stored in the signed PDF. The URL of the OCSP server is retrieved from the certificate. If the OCSP part is not found in the signing certificate, the value from the *default OCSP server URL* field will be used.

Proxy settings

If some "online" feature (TSA, CRL, OCSP) is enabled and JSignPdf runs behind a firewall, you can set the proxy, which will be used for all internet connections. Proxy type DIRECT means no proxy will be used.

[1] Only the private keys, which are valid (at the time of the signing) are displayed in the list. If the certificate supports the Key Usage extension, the private key will only be displayed if it is meant for signing.

Using hardware tokens for signing

Steps to sign documents using hardware tokens:

1. Install PKCS#11 driver for your token. Check the vendor's documentation and install a proper driver for your system;
2. Place a configuration file at `<config-dir>/pkcs11.cfg` (see Configuration directory). It is used to configure a Java SunPKCS11 security provider (see PKCS#11 Reference Guide). The easiest way to create or edit it is via *File* > *Preferences* > *PKCS#11* in the JavaFX UI.
   The content depends on your driver, you can try to start with a simple 2 lines:

```
name=Test
library=/path/to/your/PKCSDriver.so
```

1. Try to run JSignPdf with PKCS11 debug enabled:

```
JAVA_OPTS="-Djava.security.debug=pkcs11keystore -Djava.security.debug=sunpkcs11" \
    bin/jsignpdf.sh
```

If it doesn't work, try to add parameter slot or slotListIndex into pkcs11.cfg file, e.g.:

```
name=Test
library=/path/to/your/PKCSDriver.so
slot=2
```

or

```
name=Test
library=/path/to/your/PKCSDriver.so
slotListIndex=1
```

Value used for slot or slotListIndex depends on how many certificates you have installed.

If the PKCS11 keystore type works properly in the GUI and you can use the certificate on your token, you're ready to use it also in the batch mode.

```
jsignpdf -kst PKCS11 -ksp 123456 document.pdf
```

Signing engines

Starting with JSignPdf 3.1, the signing backend is pluggable. The work of producing the signature is performed by a **signing engine**, selected at runtime. JSignPdf ships with the **OpenPDF** engine (id `openpdf`), which is the default and produces exactly the same signatures as previous releases — so unless you choose otherwise, nothing changes.

To see the engines available in your installation:

```
jsignpdf --list-engines
```

Each line is printed as `<id> - <name>`, with the default engine annotated `(default)`.

The default engine is taken from the `engine` key in `<config-dir>/advanced.properties` (bundled default: `openpdf`). To override it for a single command-line run without changing the configuration, use `-eng` / `--engine`:

```
jsignpdf --engine openpdf -ksf cert.p12 -ksp secret document.pdf
```

In the JavaFX UI, the active engine is chosen from the **Engine** selector in the Preferences; the choice is saved to `advanced.properties` immediately. When an engine does not support a particular option, the corresponding control is disabled (with an explanatory tooltip); on the command line, signing fails fast with a message listing the unsupported options.

Engines are discovered automatically: dropping a third-party engine jar (with its dependencies and a `META-INF/services/net.sf.jsignpdf.engine.SigningEngine` registration) into the `lib/` directory of an installed JSignPdf makes it appear in `--list-engines` and the Preferences engine selector.

PAdES & the DSS engine

JSignPdf 3.1 bundles a second engine, **EU DSS (PAdES)** (id `dss`), built on the European Commission's Digital Signature Service library. It produces **PAdES** signatures (`ETSI.CAdES.detached`) at the four ETSI baseline levels, which the OpenPDF engine cannot create:

Level	What it adds
`B`	Basic signature (the default when `dss` is selected without a level).
`T`	`B` + a signature timestamp from the configured TSA. Setting a TSA while requesting `B` automatically upgrades to `T`.
`LT`	`T` + embedded validation material (certificates and OCSP/CRL responses) for long-term validation.
`LTA`	`LT` + an archive timestamp.

Select it with `-eng dss` on the command line (or the **PAdES Level** sidebar section in the GUI) and pick the level with `-pl` / `--pades-level`:

```
jsignpdf -eng dss -pl LT -ksf cert.p12 -ksp secret -ha SHA256 document.pdf
```

DSS requires a PAdES digest, so the hash algorithm must be `SHA256`, `SHA384` or `SHA512` (`SHA1`/`RIPEMD160` are rejected). Certificate-based PDF encryption and CloudFoxy external signing are not available with `dss`; use the OpenPDF engine for those.

Trust material for LT/LTA. The `B` and `T` levels work fully offline (only `T` needs network access to the TSA). The `LT` and `LTA` levels embed revocation data, so they need reachable OCSP/CRL endpoints and a trust anchor set. These are configured with `engine.dss. keys in advanced.properties (also editable from the *DSS engine` tab of the Preferences dialog):

Key	Meaning
`engine.dss.online.enabled`	Fetch revocation data (OCSP/CRL) and intermediate certificates (AIA) online. Must be `true` to produce `LT`/`LTA`. Default `false` (keeps `B`/`T` offline).
`engine.dss.trust.eu.enabled`	Build trust anchors from the bundled EU List of Trusted Lists (LOTL), validating the LOTL signature against the bundled Official Journal (OJ) keystore. Default `false`.
`engine.dss.trust.eu.lotlUrl`	Relocate the default EU LOTL (default `https://ec.europa.eu/tools/lotl/eu-lotl.xml`). Effective when `engine.dss.trust.eu.enabled=true`.
`engine.dss.trust.eu.ojUrl`	Official Journal scheme-information URL used for the OJ announcement predicate. Effective when `engine.dss.trust.eu.enabled=true`.
`engine.dss.trust.eu.ojKeystoreFile` `engine.dss.trust.eu.ojKeystorePassword`	External OJ keystore (and its password) overriding the bundled one used to validate the EU LOTL signature.
`engine.dss.trust.lotlUrls`	Comma-separated custom LOTL URLs.
`engine.dss.trust.lotlMraSupport`	Enable MRA processing for the custom `lotlUrls` (third-country mutual-recognition LOTLs, e.g. the eIDAS international pilot `mra_lotl.xml`). Default `false`.
`engine.dss.trust.certFiles`	Comma-separated trusted X.509 certificate files.
`engine.dss.trust.certUrls`	Comma-separated trusted X.509 certificate URLs.

`engine.dss.trust.truststoreFile`
`engine.dss.trust.truststoreType`
`engine.dss.trust.truststorePassword`
	A truststore holding trust anchors (type defaults to the JVM default).

Key	Meaning
`engine.dss.trust.systemStore`	`true`/`false` (default `false`). When on, also trust this platform's OS / JVM **root** certificate stores: `cacerts` everywhere, plus `Windows-ROOT` on Windows and `KeychainStore` on macOS. These are public-CA / OS roots, not eIDAS trusted-list anchors: they let `LT`/`LTA` embed validation data for commercially-issued (non-qualified) certificates but do not confer qualified status. The Windows personal store (`Windows-MY`) is intentionally not included — it holds your own end-entity certificates, and trusting the signing certificate itself breaks LT/LTA (see Self-signed and private-PKI certificates).
`engine.dss.trust.allowUntrusted`	`true`/`false` (default `false`). **Advanced / private-PKI or testing only.** Downgrades DSS's untrusted-chain and missing-revocation checks to warnings so `LT`/`LTA` completes for a self-signed or otherwise untrusted signer chain that has no revocation service. The output has the `LT`/`LTA` structure but **not** the revocation data a conformant long-term signature must embed, so strict validators will not accept it as such. See Self-signed and private-PKI certificates.
`engine.dss.contentSize`	Bytes reserved in the PDF `/Contents` for the CMS signature. `0` (default) sizes it automatically from the certificate chain and options; set a positive value to force a fixed size.
`engine.dss.retryOnUndersize`	Re-sign with a larger reservation if the reserved `/Contents` turns out too small. Default `true`; each retry repeats signing (and refetches the TSA timestamp for level `T` and above).
`engine.dss.tsa.nonce`	`true`/`false` (default `true`). Send a random nonce in the RFC 3161 timestamp request and require the TSA to echo it back, so a replayed or substituted timestamp response is rejected. Turn it off only for a TSA that rejects or mishandles nonced requests. The `openpdf` engine always sends a nonce and ignores this key.
`engine.dss.relaxFieldOverlap`	`true`/`false` (default `false`). By default DSS refuses to sign when the visible signature rectangle overlaps an existing annotation, because the overlapped content may be what the signer believes they are signing. Set to `true` to log a warning and place the signature on top instead — useful when signing documents that already carry stamps or form widgets in that area.

If `LT`/`LTA` is requested but revocation data cannot be reached (for example, with online fetching disabled), the signing fails with a logged error rather than silently emitting a weaker level. The log also records which trust anchors were loaded and every AIA / CRL / OCSP call the engine made,

which is usually enough to tell a network problem from a missing anchor — see Reading the signing diagnostics.

On the command line these keys can be set for a single run with `-o` (see General options), without editing `advanced.properties`:

```
jsignpdf doc.pdf -eng dss -pl LT -ts <tsa-url> \
    -o engine.dss.online.enabled=true \
    -o engine.dss.trust.certFiles=/path/to/ca.pem
```

Self-signed and private-PKI certificates

`LT`/`LTA` are defined around trusted CAs that publish revocation data, so a **plain self-signed** certificate cannot produce a conformant long-term signature: it is its own root (no revocation service), and if you trust that certificate as an anchor DSS aborts with `Signing-certificate token was not found!` because the signing certificate must not be a trust anchor. You have three options:

- **Use level `B` or `T`.** Level `T` embeds a qualified timestamp and needs no trust anchor or revocation, so it works with a self-signed certificate. This is the recommended level for self-signed / test keys.
- **Run a proper private PKI (real `LT`/`LTA`).** Issue your signing certificates from your own **root (or intermediate) CA**, and give each leaf a reachable **CRL Distribution Point** and/or **OCSP (AIA)** endpoint. Then trust the **CA** — not the leaf — as an anchor, and enable online fetching:

  ```
  jsignpdf doc.pdf -eng dss -pl LTA -ts <tsa-url> \
      -o engine.dss.online.enabled=true \
      -o engine.dss.trust.certFiles=/path/to/your-root-ca.pem
  ```

  DSS skips revocation for trust anchors, so the CA needs none; the leaf's chain is anchored through the CA and its revocation is fetched from the endpoint in the certificate. (For a TSA whose root is likewise outside the public trust lists, add that root to `certFiles`/`truststoreFile` too.)
- **Force it with permissive trust (non-conformant).** Setting `engine.dss.trust.allowUntrusted=true` lets `LT`/`LTA` complete for a self-signed or untrusted chain by downgrading the trust and revocation checks to warnings. The file then has the `LT`/`LTA` **structure** but no real revocation material, so it is **not** valid for eIDAS long-term validation — use it only for private-PKI or testing.

Certification level and encryption with LT/LTA

`LT` and `LTA` extend a signature by appending validation material (the DSS dictionary) and, for `LTA`, an archive timestamp **after** the signature, as PDF incremental updates. This is required by the levels and is explicitly permitted by ISO 32000-2 and ETSI EN 319 142.

A **certification** signature (certification level other than `NOT_CERTIFIED` — i.e.

`CERTIFIED_NO_CHANGES_ALLOWED`, `CERTIFIED_FORM_FILLING`, or `CERTIFIED_FORM_FILLING_AND_ANNOTATIONS`) records a `DocMDP` restriction on what may change after signing. Some strict PDF readers (for example Foxit) treat the `LT`/`LTA` additions as violating that restriction and report the document as **modified or corrupted since it was signed**, even though the signature validates on the official eIDAS/EU validators and in Adobe. To keep `LT`/`LTA` signatures compatible with the widest range of readers, leave the certification level at the default `NOT_CERTIFIED`.

Password-based PDF encryption combined with signing is likewise not honoured uniformly across readers. If a signed document must validate in third-party viewers, prefer an unencrypted document.

Advanced application configuration

Application-wide tweaks beyond the per-document signing options live in `<config-dir>/advanced.properties` and — for hardware tokens — `<config-dir>/pkcs11.cfg`. Both are plain text files; the easiest way to edit them is the JavaFX Preferences dialog, which writes them on *OK*. Power users can also hand-edit the files directly while JSignPdf is closed; the next launch picks up the changes.

`advanced.properties` covers the same topics as the Preferences dialog (the signing engine, verbose `debug` output, visible-signature font, certificate-validation toggles, relaxed SSL, PDF preview backend order, default TSA hash algorithm) plus a few file-editable knobs. When a key is missing from `advanced.properties`, JSignPdf falls back to defaults bundled inside the application jar, so a fresh install needs no config file at all.

Set `debug=true` (bundled default `false`, the **Debug output** checkbox on the *General* tab of the Preferences dialog) to log the verbose signing diagnostics — the certificate chain, the loaded trust anchors, and every timestamp / OCSP / CRL / AIA network call. See Reading the signing diagnostics.

Set `output.suffix` (bundled default `signed, the Output file suffix field on the _General` tab of the Preferences dialog) to the text appended to the input file name when no output file is given — for example `output.suffix=firmado produces mydocument_firmado.pdf. It is used exactly as configured, so it has to include the separator. It is the starting value for the _Output suffix` field in the *Signature Properties* panel and the default for the CLI `-os` option.

Set `output.suffix.timestamp` (bundled default `_timestamped`, no Preferences control) to the suffix used by the append-only document timestamp operation (`--timestamp-only`), which adds no signature — `_signed` would be a lie there. It is used exactly as configured and only applies when no suffix is set explicitly. See Adding a document timestamp.

Set `ui.language` to a BCP-47 language tag (e.g. `de`, `fr`, `pt`, `zh-CN`) to pick the interface language explicitly instead of following the operating-system locale; the empty bundled default keeps **System default**. The value is read once at startup, so restart JSignPdf for a change to take effect. It is exposed as the **Language** selector on the *General* tab of the Preferences dialog (which lists the bundled translations and writes this key), and on the command line as `-o ui.language=<tag>` — for example `jsignpdf --help -o ui.language=de` prints the help in German. The choice affects the interface text only; number and date formatting (and therefore the signed output) keep following the OS locale.

Set `preview.render.dpi` (bundled default `300`, no Preferences control) to change the resolution at which document pages are rasterized for the on-screen preview. A higher value gives a sharper preview at the cost of memory; the value is clamped to the 72—600 range, and pages large enough to exceed a safe pixel budget (A0 or CAD drawings, for example) are automatically rendered at a lower resolution. The setting affects the preview only — it never changes the signed document. The order of the preview backends themselves is `pdf2image.libraries`, which the *PDF Render* tab of the Preferences dialog writes.

JSignPdf stamps `config.version` into the file to record which one-time upgrades have already been applied to it. Leave it alone; deleting it only makes an upgrade run again. Version 1 covers the 3.2.0 change of the preview backend order: a file that still held the previous default

`jpedal,pdfbox,openpdf` is moved to the new default, while an order you set yourself is kept. Once the file is stamped the order is never second-guessed again, so selecting `jpedal,pdfbox,openpdf` on purpose afterwards is preserved.

Set `buffering.mode=temp` (bundled default `memory`, no Preferences control) to stage the document being signed in temporary files instead of on the Java heap, so its size no longer has to fit in `-Xmx`; `buffering.tempDir` chooses where those files go. Both keys are described under Signing very large documents.

In batch mode, any of these keys can be overridden for a single run with the `-o key=value` command-line option (see General options), without changing the file. CLI overrides take precedence over `advanced.properties` and the bundled defaults, and are not persisted.

Migration from earlier versions

In JSignPdf 2.x the same options lived in `<install dir>/conf/conf.properties` and the PKCS#11 provider was pointed to by a `pkcs11config.path` key in that file. Starting with 3.0.0 those files move under your per-user configuration directory:

- `<install dir>/conf/conf.properties` → `<config-dir>/advanced.properties`
- `<install dir>/conf/pkcs11.cfg` → `<config-dir>/pkcs11.cfg` (a fixed well-known location; the `pkcs11config.path` key has been removed)

The 3.0.0 distribution no longer ships an `<install dir>/conf/` directory and there is no automatic migration of pre-3.0.0 install-dir tweaks. If you previously customized `conf.properties` or maintained a `pkcs11.cfg`, re-enter the values via the Preferences dialog — or copy your old files into `<config-dir>/advanced.properties` and `<config-dir>/pkcs11.cfg` by hand before launching 3.0.0.

Configuration directory

Starting with JSignPdf 3.0.0, per-user state is stored in a platform-native configuration directory instead of a single file in your home folder. The directory holds the main config file (`config.properties`, which persists the last-used signing settings), a `presets/` subdirectory (see Using presets), and — once you save changes from the Preferences dialog — the advanced-config files `advanced.properties` and `pkcs11.cfg` (see Advanced application configuration).

The default location depends on your operating system:

Platform	Default location
Linux / BSD	`$XDG_CONFIG_HOME/jsignpdf/`, falling back to `$HOME/.config/jsignpdf/`
Windows	`%APPDATA%\JSignPdf\` (typically `C:\Users\<you>\AppData\Roaming\JSignPdf\`)
macOS	`~/Library/Application Support/JSignPdf/`

You can override the default by setting the `JSIGNPDF_CONFIG_DIR` environment variable to any directory path — useful for portable installs, scripted tests, or keeping multiple profiles side-by-side.

Layout

```
<config-dir>/
  config.properties          # main configuration (last-used signing settings)
  advanced.properties        # advanced preferences (created on first save)
  pkcs11.cfg                 # SunPKCS11 provider config (optional)
  presets/
    preset-1745270400123.properties
    preset-1745270500456.properties
    ...
```

Preset filenames are opaque (`preset-<epoch-millis>.properties`); the user-visible name lives inside the file as `preset.displayName`. You can read, copy, or hand-edit any of these files — they are plain Java properties.

Migration from earlier versions

Previous versions of JSignPdf stored the main settings in `~/.JSignPdf` (a single file in your home directory). On the first run of JSignPdf 3.0.0 or newer, if the new configuration directory does not yet exist, JSignPdf copies `~/.JSignPdf` to `<config-dir>/config.properties` and creates an empty `presets/` subdirectory. **The legacy file is left in place**, so if you downgrade to an older JSignPdf release, it continues to work with its old settings. Pre-3.0.0 install-dir tweaks (`conf/conf.properties` and `conf/pkcs11.cfg`) are **not** migrated automatically — see Advanced application configuration for what to do.

Once the new directory exists, the legacy locations are no longer consulted or updated.

Java VM options for the native installer launchers

If you need to change JVM properties for the four bundled launchers (for example to increase the maximum heap), edit the corresponding `.cfg` file under the `app` subfolder of the installation. jpackage creates one `.cfg` per launcher:

Launcher	Configuration file
`JSignPdf`	`app/JSignPdf.cfg`
`JSignPdf-swing`	`app/JSignPdf-swing.cfg`
`JSignPdfC`	`app/JSignPdfC.cfg`
`InstallCert`	`app/InstallCert.cfg`

The `app/` folder lives under the install root: `%ProgramFiles%\JSignPdf\app\` on Windows, `/opt/jsignpdf/lib/app/` on Linux, `JSignPdf.app/Contents/app/` on macOS.

Each `.cfg` file contains a `[JavaOptions]` section with one `java-options=` entry per option. Add additional options as further `java-options=` lines, for example:

```
[JavaOptions]
```

```
java-options=--add-exports=java.base/sun.security.util=ALL-UNNAMED
java-options=-Xmx1g
java-options=-Dswing.aatext=true
```

The same `.cfg` files are present inside the per-platform portable ZIPs (`jsignpdf-${VERSION}-windows-x64.zip`, `jsignpdf-${VERSION}-linux-x64.zip`, ...) under `JSignPdf/app/` (Windows / Linux) or `JSignPdf.app/Contents/app/` (macOS).

Troubleshooting

Out of memory error

If you see an OutOfMemoryError in the program console, the document is bigger than the heap Java was given.

Signing very large documents

Try this first. By default both engines build the whole signed document on the Java heap, so a large PDF needs a large heap. Setting `buffering.mode=temp` in `advanced.properties` stages the intermediates in temporary files instead, and document size stops driving heap usage:

```
buffering.mode=temp

# Optional: stage on a specific disk. Must already exist and be writable.
buffering.tempDir=/var/tmp/jsignpdf
```

Or for a single run, without changing the configuration file:

```
$ bin/jsignpdf.sh -o buffering.mode=temp ...
```

The signed output is the same either way; only the staging location changes. The cost is disk I/O and free space: roughly the size of the document for the `openpdf` engine, two to three times that for `dss` (it stages a copy while computing the digest and another while assembling the signature). Point `buffering.tempDir` at a fast local disk when signing very large files.



`buffering.tempDir` does not cover **everything** the `dss` engine writes. The underlying DSS library has no setting for the location of its own PDF scratch files, so those always land in the JVM's temporary directory. On systems where `/tmp` is a RAM-backed `tmpfs` (common on Fedora, Arch and openSUSE) that puts part of the spill straight back into memory, which defeats the purpose. Redirect the JVM's temporary directory as well in that case:

```
$ JAVA_OPTS="-Djava.io.tmpdir=/var/tmp/jsignpdf" bin/jsignpdf.sh -o
buffering.mode=temp ...
```

The `openpdf` engine has no such gap — it honours `buffering.tempDir` for everything it stages.



`buffering.mode=temp` also lifts a limit that `-Xmx` cannot. In the default mode the `openpdf` engine stages the document in a single Java array, which caps the **output** at 2 GB no matter how much heap is available. Its PDF reader still cannot open an **input** larger than 2 GB, though — use the `dss` engine for those.

Raising the heap

If the document is not unusually large, or `temp` mode is not enough, allow Java to use more memory.

When running from a cross-platform ZIP, set the `-Xmx<size>` flag via `JAVA_OPTS` before invoking the launcher. The following example allows Java to use 512 MB of heap:

```
$ JAVA_OPTS=-Xmx512m bin/jsignpdf.sh
```

When running from a native installer (MSI / DEB / RPM / DMG / Flatpak) or the per-platform portable ZIP, add a `java-options=-Xmx512m` line to the `.cfg` file of the affected launcher (see Java VM options for the native installer launchers).

SSL/TLS certificate errors

When JSignPdf connects to a server through HTTPS (e.g. to a TSA server for timestamping), it may fail with a `SSLHandshakeException`. This happens because Java ships with a keystore of trusted root certificates (`cacerts`), and if the server's certificate is not signed by one of these trusted authorities, the connection is refused.

If you trust the server, you can add its certificate to the Java `cacerts` keystore using the bundled InstallCert tool (see InstallCert Tool).

Certificate not found in keystore

If JSignPdf doesn't list your certificate when you press *Load keys*:

- Verify the keystore file path and password are correct.
- Check that the certificate is still valid (not expired).
- If the certificate has a Key Usage extension, make sure it includes digital signature capability.

Timestamp server unreachable

If timestamping fails:

- Verify the TSA URL is correct and the server is accessible.
- If you are behind a proxy, configure the proxy settings (see Proxy settings).
- Check if the TSA server requires authentication and configure it accordingly.
- If you get an `SSLHandshakeException`, see SSL/TLS certificate errors.

Reading the signing diagnostics

With **debug output** enabled, JSignPdf logs the certificate material it uses and, for the DSS engine, every network call it makes while collecting validation data. This is what to look at when a signature fails for a reason that is not obvious from the error message itself.

Turn it on by ticking **Debug output** on the *General* tab of the Preferences dialog, or by setting `debug=true` in `advanced.properties` (it can also be set for a single CLI run with `-o debug=true`). It is off by default so normal runs stay quiet. Everything below is emitted only while it is on.

Certificate chain. Right after the key is loaded, the whole signing chain is dumped — one block per certificate, leaf first:

```
Signing certificate chain (2 certificates)
  [0] subject=O=Acme,CN=Jane Doe
      issuer=O=Acme,CN=Acme Issuing CA
      serial=2a1f, validity=2026-01-01 00:00:00 UTC .. 2027-01-01 00:00:00 UTC
      id=C-958E99061FA999181718D8FE3E6D70ECC4EF46DD78FC76E76AD348214E7AD6EE
      keyUsage=digitalSignature,nonRepudiation
      qcStatements=QcCompliance,QcSSCD
      AIA caIssuers=http://pki.acme.example/ca.crt
      AIA OCSP=http://ocsp.acme.example
      CRL DP=http://crl.acme.example/issuing.crl
```

The `id=C-⋯` value is the identifier the DSS engine uses in its own error messages, so a `No revocation data found for C-⋯` failure can be matched to a real certificate without inspecting it with `openssl`. A certificate outside its validity window is flagged with `(EXPIRED)` / `(NOT YET VALID)` next to the dates. The timestamp token's own chain is dumped the same way once the TSA has answered.

Trust anchors (DSS engine). Each configured `engine.dss.trust.*` source is logged with the number of anchors it contributed, followed by the total. A total of `0` means nothing is trusted at all — a configuration problem — as opposed to a specific CA simply not being covered:

```
Trust source trust.certFiles=/etc/pki/acme-root.pem: 1 anchor(s)
Trust anchors loaded: 1 from 1 configured source(s)
```

Revocation and issuer downloads (DSS engine). For `LT`/`LTA`, every AIA, CRL and OCSP call is reported twice: once for the HTTP request (URL, response size, elapsed time, or the failure) and once for what the answer meant, naming the certificate it was for:

```
CRL GET http://crl.acme.example/issuing.crl: 503 bytes, elapsed=18ms
CRL lookup for Jane Doe (C-958E...): status=GOOD,
source=http://crl.acme.example/issuing.crl,
    thisUpdate=2026-07-24 06:24:29 UTC, nextUpdate=2027-07-24 07:24:29 UTC,
elapsed=27ms
OCSP lookup for Jane Doe (C-958E...): no OCSP data (no access point on the
certificate, or none usable), elapsed=0ms
```

A `FAILED` line names the responder that could not be reached and why — the usual cause of an `LT`/`LTA` failure behind a proxy or a firewall (see Proxy settings). A `no CRL data` / `no OCSP data` line means the certificate itself carries no matching distribution point, which no amount of trust configuration can fix.

These messages are logged at the `FINE` level, which `debug=true` enables. The `-q` / `--quiet` flag silences all output (including the normal progress messages) regardless of the debug setting; for full control over what is logged, point Java at your own configuration with `JAVA_OPTS=-Djava.util.logging.config.file=<file>`.

Other command line tools

InstallCert Tool

JSignPdf comes with a command line utility for adding server certificates to the Java `cacerts` keystore. This is useful when connecting to HTTPS servers whose certificates are not signed by a well-known CA (see SSL/TLS certificate errors).

When installed via a native installer (MSI / DEB / RPM / DMG / Flatpak) it is exposed as the `InstallCert` launcher:

```
$ InstallCert

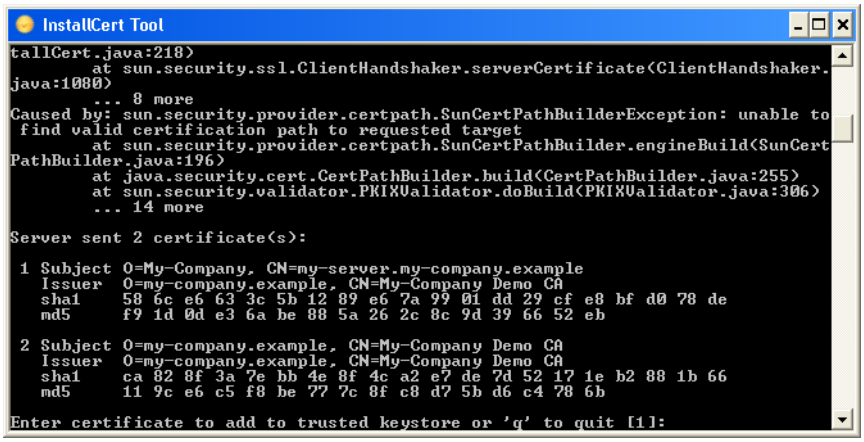
or

$ InstallCert hostname[:port] [cacertPwd]
```

From a cross-platform ZIP the tool is reachable via `java -cp` against the bundled jars:

```
$ java -cp "lib/*" net.sf.jsignpdf.InstallCert hostname[:port] [cacertPwd]
```

If you don't provide a hostname argument, you will be asked for it.



The certificate chain will be displayed and you can choose which one will be imported.