Say Hello to Snug



Build it. Brand it. Share it.

A practical guide to packaging, customising, and sharing applications with Snug.

Revised 24 October 2026 · Windows x64

You've built an application.
Now you need to get it onto someone else's computer.

Welcome to Snug.

This book was written for people who are new to packaging and distributing Java applications, although it also contains deeper technical sections for more experienced or curious readers.

Read online: https://synapticloop.github.io/snug/
PDF/eBook: https://github.com/synapticloop/snug/releases/latest/

Your First Snug Application

This first exercise is for anyone who wants to see Snug working before learning about its configuration options. You can follow it even if you have never built a Java application before.

You may have written a Java application yourself, created one with an AI assistant, or arrived here without an application of your own. We will use the existing Snug JavaFX demonstration application so that everyone starts with the same files. The goal is to create and run a Windows executable first; the explanations of JAR files, runtimes, and packaging will make more sense once you have seen the result.

Your first Snug build

For this first build, you will download Snug and the prebuilt Snug JavaFX demonstration JAR, create a Windows executable using the Snug Quick Build helper, and then run it. You do not need to compile Java code or open a terminal:

1. Open the latest Snug release page and select the Windows x64 ZIP file from its Assets list:

   https://github.com/synapticloop/snug/releases/latest/

2. Extract the ZIP file to a folder.

The unzipped folder will have the following icons:

 `snug.exe`	Builds Windows executables from Java applications
 `Snug Quick Build.exe`	Lets you build an executable by dragging a JAR file onto it
 `snug_preview.exe`	Lets you preview Snug's launcher dialogs.

For this exercise, we will use Snug Quick Build. The other tools are explained later in the book.

3. Download the existing Snug JavaFX demonstration JAR and place it in the extracted Snug folder. A JAR is the file containing the Java application; for this exercise, you do not need to know how it was made. Download the demonstration file from:

   https://github.com/synapticloop/snug/blob/main/assets/snug-javafx-demo-windows.jar

On the GitHub file page, click “Download raw file” to save the JAR.



Image: Click the Download raw file icon to download the demonstration JAR.

Your folder should now look like the following:



***Image:** The extracted Snug folder, with the Snug Quick Build helper (`Snug Quick Build.exe`), the Snug builder (`snug.exe`), Snug Preview (`snug_preview.exe`), and the demonstration JAR (`snug-javafx-demo-windows.jar`).*

4. Drag the JAR file onto `Snug Quick Build.exe`.
5. To drag a file, point at the JAR file, hold down the left mouse button, move the pointer over `Snug Quick Build.exe`, and release the mouse button.

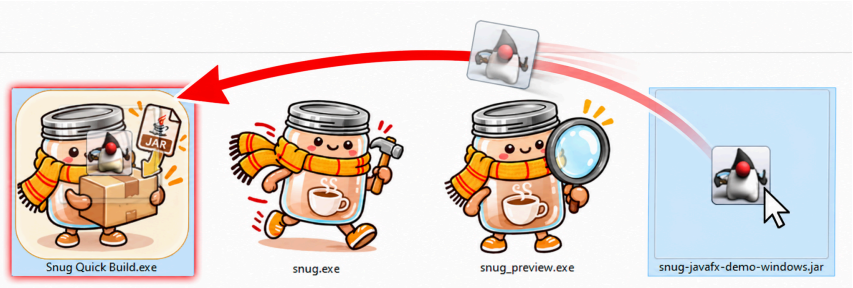


Image: The demonstration JAR being dragged onto Snug Quick `Build.exe`***.***

6. Wait for Snug Quick Build to finish creating the Windows executable. A new file named `Example Application Name.exe` should appear in the same folder. **Note:** This is the default filename used by Snug Quick Build. If no EXE appears, open snug-build.log in the same folder in Notepad and check the Quick Build error before continuing.
7. Double-click `Example Application Name.exe` to run it. If a compatible Java runtime is not already available, Snug may ask permission to download one. Follow the on-screen instructions. Once a suitable runtime is available, the demonstration application should open.



Image: *The new application "*`Example Application Name.exe`*" (outlined in green) from the Snug Quick Build helper.*

What just happened?

A JAR contains the Java application you started with. The Snug Quick Build helper uses the normal Snug builder to turn it into a Windows executable (`.exe`).

For now, getting your first executable to run is enough.

Where to go next

You now have a working Windows executable. If you want to give your application its own name, icon, and other settings, continue to **Section 2: Building with Snug**, where we introduce the normal Snug builder.

~ ~ ~ * ~ ~ ~

If you simply wanted to try Snug, you can stop here.

~ ~ ~ * ~ ~ ~

TABLE OF CONTENTS

Your First Snug Application **2**
Where to go next 5
SECTION 1:
ABOUT THIS BOOK **10**
About This Book **11**
Why Snug? 11
What Snug Offers 12
What Snug Does Not Do 13
SECTION 2:
BUILDING WITH SNUG **15**
Building an Application with Snug **16**
The Builder and Launcher 16
Before You Begin 16
Quick-Start: Hello Snug JavaFX 16
That's It 20
Packaging Your Own Application 21
SECTION 3:
CUSTOMISING THE SNUG USER EXPERIENCE **23**
The Snug Experience **24**
What Happens When Someone Double-Clicks? 24
Application Identity 25
Introducing the snug.options File 26
The Application Icon 27
The Splash Screen 29
SECTION 4:
CONFIGURATION OPTIONS **31**
How Snug Configuration Works **32**
Input and Output 33
Application Identity 36
Changing the Cache Location 37
Application Metadata 37
Java Runtime 39
Application Instance Behaviour 41
Splash Screen Configuration 41

Runtime Download Behaviour....42
Launcher Messages and Localisation....44
Creating Localisation Files....45
Windows Manifest....46
Build and Diagnostic Tools....46
Using a Different Options File....48
A Practical Configuration Workflow....50
Putting It Together....51
SECTION 5: CUSTOMISING LAUNCHER MESSAGES....53
Your First Message Change and Preview....54
Understanding Launcher Dialogs....55
Customising the Messages....56
Common Localisation Key Suffixes....60
The Runtime Installation Dialogs....60
When Snug Cannot Launch....62
Message Key Reference....62
Replacement Tokens and the Key Reference....68
Keys Without Replacement Tokens....73
Previewing Your Changes....75
Other Languages....76
Before You Build: Final Checks....77
SECTION 6: MANAGING YOUR SNUG PROJECT....78
Managing Your Snug Project....79
What Should Go in the snug Directory?....79
What Should Be Saved with the Project?....80
Using Git....80
Should snug.exe Be Committed?....80
A Practical Starting Point....81
Reproducible Builds....81
Automating the Build....82
SECTION 7: TECHNICAL DETAILS....83
Snug Technical Details....84
What Snug Actually Builds....84
The Builder and the Launcher....85
Anatomy of the Windows Executable....86
The Java Application Remains a JAR....88

What Happens When the Executable Starts ... 88
Application Identity and the Cache ... 89
Finding a Java Runtime ... 92
Starting Java ... 93
When the Java Application Exits ... 94
Application Arguments ... 94
The Windows Manifest ... 95
Logging and Diagnostics ... 96
Troubleshooting a Launch ... 97
Code Signing Is Separate from Packaging ... 97
Why Snug Does Not Bundle Everything ... 98
Keeping the Layers Separate ... 98
Technical Details Will Evolve ... 100

SECTION 8: APPENDICES ... 101

Command-Line Options Reference ... 102
Quick Overview: Common Builder Options ... 102
Command-Line Practical Reference ... 105
Snug Builder (snug.exe) Options Reference ... 108
Snug Preview (snug_preview.exe) Options Reference ... 125

Building Snug from Source ... 127
Why Build Snug Yourself? ... 127
Before You Begin ... 127
Obtaining the Source Code ... 128
Understanding the Project Layout ... 128
Building the Complete Windows Release ... 129
Building Individual Components with Cargo ... 130
Running Tests and Checks ... 131
Checking the Result ... 131
Common Build Problems ... 132
Making Changes and Contributing ... 132
A Note About Versions ... 132

Definitions ... 133

Copyright, Licensing, and Third-Party Notices ... 140
Copyright Notices ... 140
Copyright and Licensing ... 140
Snug ... 141
Third-Party Software ... 141

About Snug **142**
Why Is It Called Snug? 142
Afterword: On Documentation **143**
End Plate **145**

SECTION 1:

ABOUT THIS BOOK

This section introduces the book, explains some of the terminology used, and shows how information is formatted.

About This Book

Why Snug?

Building an application and sharing an application are two different things.

Imagine that you have just finished your first Java application.

- Perhaps you wrote it yourself while learning Java, or
- Perhaps an AI assistant helped you turn an idea into something that works, or
- Perhaps you are a professional developer and just need a way to give it to other less technical people

You can launch it on your computer, use it, and see the results. Naturally, you want to give it to somebody else.

That is when a new set of problems can appear. Does the other person have Java installed? Is it the right version? Will they know what to do with a JAR file? What if your application needs additional libraries? Even when the application works perfectly, running it may not feel like opening an ordinary Windows program.

These are not problems with what your application does. These are problems with preparing it for somebody else to use. Snug was created to bridge that gap. It turns a working Java application into a Windows executable with a familiar name and icon, and automatically helps find or download the Java runtime needed to launch it. Instead of explaining Java or JAR files to the person receiving your application, you can give them something they can double-click.

Snug is for anyone who wants a straightforward way to share a Java desktop application, whether they built it themselves, used AI assistance, or develop software professionally. It is particularly useful for small, practical applications shared with colleagues or used within a business.

Snug was developed to accompany a separate book about building Java applications with AI assistance. This book covers the next step: sharing those applications and making them easy for someone else to run.

An Application, Not an Installer

Snug is not an application installer. It creates a Windows executable (`.exe`) that you can double-click to run your Java application. There is no installation wizard, no automatic creation of desktop shortcuts or Start menu entries, and no uninstaller.

~ ~ ~ * ~ ~ ~

The executable is the application you share.

~ ~ ~ * ~ ~ ~

You can copy it to another folder, store it on a USB drive, or give it to someone else. To remove the application, simply delete the executable.

Snug may retain supporting files, such as cached application components and a Java runtime downloaded when the application first runs. These are stored separately and are not automatically removed when you delete the executable.

The goal is simple: build it, brand it, share it, and double-click it to run.

What Snug Offers

Snug is designed to make Java applications easier to package, share, and run.

- **Make applications easy to run.** Turn your Java application into a familiar Windows executable that users can double-click without needing to understand Java or JAR files.
- **Manage Java runtime requirements.** Snug can find or download a suitable Java runtime, so users do not need to install and configure Java themselves.
- **Give your application its own identity.** Add an application name, icon, splash screen, version information, and other metadata to create a more polished Windows application.
- **Support local desktop applications.** Package applications that run on the user's own computer without requiring cloud infrastructure, making Snug particularly useful for internal business tools.
- **Control how your application starts.** Configure Java requirements, application arguments, startup behaviour, and launcher messages to suit your application.

What Snug Does Not Do

Snug concentrates on packaging Java applications and providing a familiar Windows launch experience. It deliberately leaves some responsibilities to other tools or to the person distributing the application.

- **It is not an application store or publishing platform.** Snug creates the executable but does not host applications, provide download pages, or distribute them automatically.
- **It does not update your application automatically.** When you release a new version, you decide how to make it available. Snug can include a link to an update page in its launcher error dialogs, but it does not download or install application updates.
- **It is not an installer or uninstaller.** Snug does not automatically create desktop shortcuts, Start menu entries, or an uninstaller.

Once Snug has created the executable, you decide where it is stored, who can access it, and how updates are distributed. Your application can be shared privately within a business or made publicly available using whatever distribution method you prefer.

~ ~ ~ * ~ ~ ~

Snug deliberately does less.

It creates a polished, self-managing
launch experience for your
Java application, then gets out of the way.

~ ~ ~ * ~ ~ ~

A Few Terms Before We Start

Here are some terms that will be useful.

Application

> *A computer program created to perform a particular task. In this book, it is the Java program you want Snug to package and run as a Windows application.*

JAR

Short for Java Archive. A JAR file contains compiled Java code and other files needed by an application. Snug packages one or more JAR files into a Windows executable.

Executable

A file that the operating system can run as a program. On Windows, executable files normally use the `.exe` extension. Snug creates a Windows executable that can be launched like a normal desktop application.

Runtime

The software environment required for another program to run. Java applications need a suitable Java runtime, which Snug can locate or download when needed.

Launcher

A small program that prepares and starts another application. The Snug launcher is built into the executable and starts the Java application.

Artefact

A file produced by a build or packaging process. In this book, a Snug artefact usually means the Windows executable created from your Java application.

How to Read This Book

Start with the practical building chapters, then use the configuration chapters and appendices when you need them. **Section 7: Technical Details** provides optional background and can be skipped without affecting the beginner exercises.

Sidebars highlight important information, useful notes, tips, and optional technical detail. Commands and examples appear in formatted blocks:



Note: Do not type line numbers or the ↩ symbol: it marks a printed line wrap, so join those lines into one command. A PowerShell backtick (`) at the end of a command line is different: type it exactly as shown to continue onto the next line.

SECTION 2:

BUILDING WITH SNUG

This section moves from the drag-and-drop demonstration to building directly with `snug.exe`.

Building an Application with Snug

The Builder and Launcher

In the first exercise, Snug Quick Build used example settings. Here you will use `snug.exe` directly, choosing the application's name and company. Snug creates the executable, and its built-in launcher starts the Java application when someone runs it.

- **Snug Builder (`snug.exe`)**
  This is the tool *you* use to create the finished Windows executable.

- **Snug launcher**
  The builder embeds this launcher in the executable you create. It prepares the Java runtime and starts your application when someone runs that executable. The person running your application uses it indirectly, but does not normally know it is there.

Before You Begin

If you completed **Your First Snug Application**, you already have the files you need. Otherwise, download and extract the Windows x64 Snug release from

https://github.com/synapticloop/snug/releases/latest/

and place `snug-javafx-demo-windows.jar` from

https://github.com/synapticloop/snug/blob/main/assets/snug-javafx-demo-windows.jar

in the same folder.

On the GitHub file page, click the “Download raw file” icon to save the JAR.

Quick-Start: Hello Snug JavaFX

We will now package the same demonstration application using the command-line builder.

Use the example `--name` and `--company` below for this exercise. For your own application, choose these carefully because Snug uses them as part of its identity.

Open the folder containing `snug.exe` in File Explorer. Click the address bar, type `powershell`, and press Enter. A PowerShell window will open in that folder.

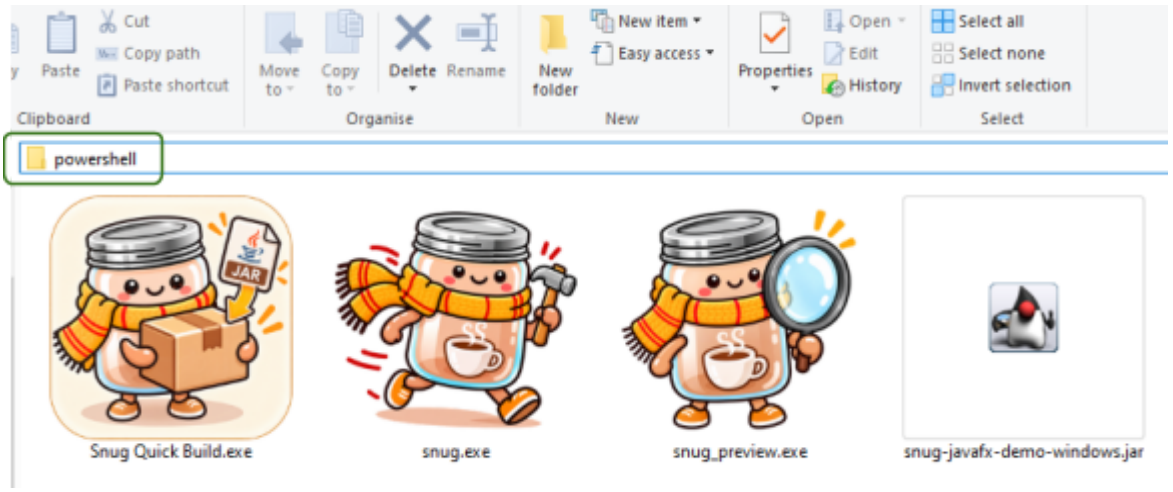


Image: *Opening PowerShell in the Snug folder using File Explorer's address bar.*

Now enter the following command:

Command(s)

```
.\snug.exe --name "My App" --company "My Company" snug-javafx-demo-windows.jar
```



Note: This command packages the demonstration JAR with the application display name "`My App`" and company "`My Company`".

Because no output filename is specified, Snug creates `snug-javafx-demo-windows.exe` beside the input JAR. The `--name` option sets the application's display identity; use `--output` to choose a different executable filename.

You should see output similar to this authentic example. The byte counts may differ with the Snug version and the JAR being packaged:

```
snug: wrote snug-javafx-demo-windows.exe (11524608 bytes; ↩
    payload 9982054 as RT_RCDATA)
snug: built snug-javafx-demo-windows.exe
```

You can now run the Windows executable by double-clicking `snug-javafx-demo-windows.exe` or, from the same terminal window, by running:

Command(s)

```
.\snug-javafx-demo-windows.exe
```

When you launch the executable, Snug checks for a suitable Java runtime. If none is available, it offers to download and install the required runtime components. (If you completed Your First Snug Application at the beginning of the book, Snug will *reuse the runtime already available on your computer.)*

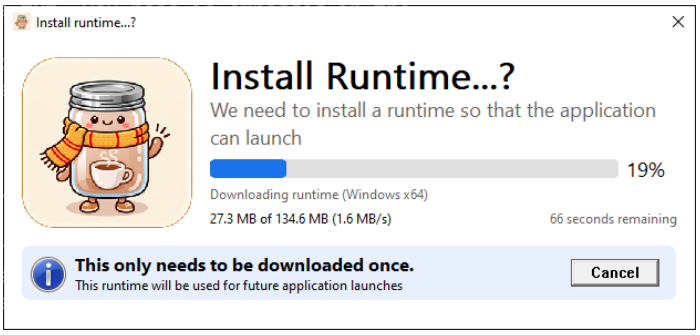


Image: *Snug's runtime installation dialog, shown when a suitable Java runtime is not available.*

Once the required runtime is available, Snug starts the demonstration application. If a suitable runtime is already available, Snug can skip the download step. The screenshot of the running application shows the result of a successful launch.



Image: *The Snug JavaFX demonstration application after a successful launch.*

Snug can reuse a downloaded runtime on future launches while it remains available and meets the application's requirements.



Note: The screens shown may differ when new versions of Snug or `snug-javafx-demo-windows.exe` are released.

Once the demonstration application opens, the build and launch have succeeded.

The final directory should contain the following files:



Image: *The Snug tools, the newly built Windows executable, and the original demonstration JAR in File Explorer.*

Above (from left to right) you can see the icons for the Snug Quick Build helper (`Snug Quick Build.exe`), Snug Builder (`snug.exe`), Snug Preview (`snug_preview.exe`), the ready-to-run Windows executable (`snug-javafx-demo-windows.exe`), and the original Java application (`snug-javafx-demo-windows.jar`).



Tip: If you change nothing else, give your application its own icon. The icon is used for the Windows executable and also appears in Snug's launcher dialogs. A simple, recognisable icon immediately makes the finished application feel more complete and easier to identify.



IMPORTANT: Choose the `--name` and `--company` values carefully. The Snug launcher uses their combination as part of the application's local identity. Reusing the same identity for unrelated applications can cause their local application data to collide.

That's It

You have now used `snug.exe` directly to turn the demonstration JAR into a Windows executable, with your own application name and company.

The rest of this book explains how to refine that result, from icons and splash screens to Java runtime settings and launcher messages.

~ ~ ~ * ~ ~ ~

"You have packaged a Java application into a

Windows executable that someone else

can simply double-click and run."

~ ~ ~ * ~ ~ ~

Packaging Your Own Application

Once the demonstration works, try the same process with your own Java application. The generic My Application example below represents your own project; replace its filenames and identity values with yours.

Before You Begin

Start with a known-working application. For a single-JAR build, use a runnable fat JAR. “Runnable” means the JAR identifies where the application starts; “fat” means it bundles the application’s compiled code and required Java libraries into one file. If someone or a build tool created the JAR for you, ask for a runnable fat JAR. An ordinary JAR may depend on additional files that Snug cannot obtain automatically.

If your application uses JavaFX or other native libraries, those dependencies must also support Windows x64. Downloading a Java runtime does not supply your application’s dependencies.

Build the Executable

Place your JAR beside `snug.exe` and open PowerShell in that folder. Replace `my-app.jar` and the example identity values below with your own: In this PowerShell command, type the backtick (`) at the end of each continued line, then press Enter.

Command(s)

```
.\snug.exe --name "My Application" `
  --company "My Company" --output "My Application.exe" `
  my-app.jar
```

The `--name` value is the application's display name. The `--output` value chooses the executable's filename.

Run and Share

Launch `My Application.exe` and check that your application works as expected.

Then copy the executable to another Windows x64 computer and test it there. Ideally, use a clean computer without your development tools, project files, or a Java runtime you installed for development. This helps reveal anything the application relies on from your own computer. The recipient does not need `snug.exe` or your original JAR. A compatible Java runtime must be available, or Snug must be able to download one.

If your application depends on external configuration, data files, or native libraries outside the packaged JAR, arrange those separately and test their locations on the receiving computer.

Applications with Several JARs

If your application and its dependencies are separate JARs, place them together in one directory and use that directory as the input. Snug includes JARs directly inside it; it does not search subdirectories.

For these applications, you may need `--main-class` to identify the entry point. See **Section 4: Configuration Options**.

If It Does Not Start

Confirm that the Java application works outside Snug, then check the launcher log. Record the Snug version, command used, and complete error message when investigating the problem. **Section 7: Technical Details** explains where to find the logs.

SECTION 3:

CUSTOMISING THE SNUG USER EXPERIENCE

This section looks at your application from the user's point of view: its identity, application icon, startup splash screen, and launch behaviour.

The Snug Experience

Snug should let people double-click your application and get on with their work, without needing to understand JAR files or Java runtimes.

When the launcher does need to communicate, its prompts, icon, and splash screen should feel like part of your application.

What Happens When Someone Double-Clicks?

When someone double-clicks the Windows executable created by Snug, they start the Snug launcher built into it, rather than the Java application directly.

The launcher checks what the application needs and prepares its files before starting Java. For the person using your application, there are two experiences worth considering: the first launch and later launches.

The Initial Launch

On the first launch, Snug checks for a suitable Java runtime before starting the application.

If a suitable runtime is unavailable, Snug can prompt the user to download the required components, as shown in **Section 2: Building with Snug**. The prompt should clearly belong to your application.



Note: Snug's runtime download behaviour can be configured. See **Section 4: Configuration Options** for the available settings.

The Normal Launch

On later launches, Snug still checks that the required runtime and application files are available. When they are, it reuses them and starts the application without repeating the download. A configured splash screen can appear during startup.

This is the experience Snug aims to provide:

~ ~ ~ * ~ ~ ~

"Double-click the application.
The application opens."

~ ~ ~ * ~ ~ ~

The technical details required to make that happen belong behind the scenes.

Application Identity

Before choosing an icon or splash screen, decide how your application should be identified.

Two settings are particularly important:

- **Application name** - The name people will recognise when they use your application.
- **Company name** - The organisation or person responsible for the application.

Snug also uses these values internally to identify the application and manage its local files. Choose them carefully, particularly if you intend to distribute the application to other people.

Other details help give your application a consistent appearance: version information, an application icon, a startup splash screen, and launcher messages.

Together, these details help make your executable recognisable as your application, rather than simply another file someone has received.

Windows can also display information about your application when someone hovers over the executable in File Explorer or opens its Properties window.

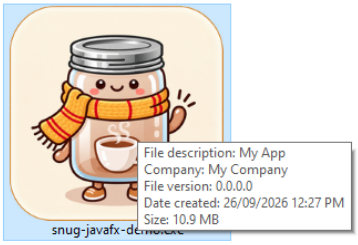


Image: *File Explorer tooltip showing application details.*

The fields shown may vary with the Windows version and File Explorer settings.

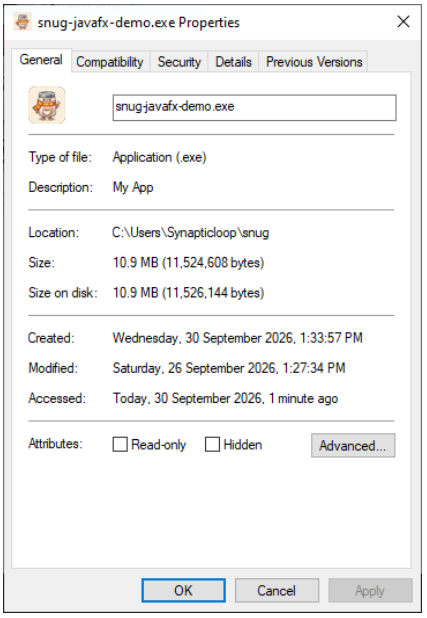


Image: *Windows Properties showing executable details.*

Before customising these details, let's bring your Snug settings together.

Introducing the snug.options File

Use `snug.options` to keep your build settings together as you customise your application.

This plain-text file stores the options Snug uses when building your executable.

The easiest way to get started is to let Snug create an example options file. Open PowerShell in the folder where you are working and run:

```
Command(s)
.\snug.exe --init-options
```

Snug creates `snug.options` in the current folder, with settings and example values you can edit. Open it in Notepad or another plain-text editor and adjust the values for your application. If `snug.options` already exists, Snug will not overwrite it by default.

You can also create the file yourself in a plain-text editor. Make sure it is named `snug.options`, not `snug.options.txt`.

To practise branding the Snug JavaFX demonstration application, start with these three settings in `snug.options`. My Application and My Company are example labels; the input remains the demonstration JAR:

```
--input "snug-javafx-demo-windows.jar"
--name "My Application"
--company "My Company"
```

Save `snug.options`, then run `.\snug.exe` in PowerShell from the same folder to build the executable. Double-click the resulting executable to test it.

Add a custom icon or splash screen using the examples below when you are ready.

Section 4: Configuration Options explains how Snug locates and reads this file, combines it with command-line options, and lists the available settings.

The Application Icon

An application icon helps people recognise your application in File Explorer, on the desktop, and in Windows dialogs. Every executable built with Snug has an icon.



IMPORTANT: Windows may cache an executable's icon using its filename. If File Explorer still shows the old icon after a rebuild, try a different output filename to check the new icon.

The Default Icon

If you do not provide your own icon, Snug uses its built-in default. That is fine while you are testing your application.



Image: Snug's default application icon.

Using Your Own Icon

To use your own icon, add a PNG or ICO image with the `--icon` option:

```
--icon <PNG|ICO>
```

For example, add the following line to `snug.options`:

```
--icon "snug\icon.png"
```

Snug accepts PNG and ICO files. Choose a simple image that stays recognisable at small sizes.

The same icon also appears in Snug's launcher dialogs, including those shown when runtime components are required or an error occurs. This helps the messages feel like part of your application.

The Splash Screen

If an application takes a few seconds to open, a splash screen shows that it is starting. This is usually an image containing the application's name, logo, or other artwork.

Snug can display a splash screen almost immediately after the user double-clicks the Windows executable. For example, `snug-javafx-demo-windows.exe` displays the Snug demonstration splash screen while the application is being prepared and started.



Image: *The Snug demonstration splash screen.*

Should Snug or Your Java Application Display the Splash Screen?

A splash screen does not have to be provided by Snug; your Java application can display its own. This gives you a choice:

1. **Let Snug display the splash screen.**

   Because Snug is already running before the Java runtime starts, its splash screen can appear very quickly after the user double-clicks the executable. This is useful when you want to give the user immediate feedback that the application is starting.

2. **Let the Java application display the splash screen.**

   In this case, the splash screen is controlled by the Java application itself. Exactly how it looks and behaves is up to the developer who created the application.

~ ~ ~ * ~ ~ ~

You would not normally need both.

For a longer startup, however, Snug could show an immediate splash screen before your Java application replaces it with an animated screen or progress indicator.

~ ~ ~ * ~ ~ ~

Do You Need a Splash Screen?

Not necessarily.

If your application opens almost immediately, a splash screen may add little value and can make startup feel slower. If there is a noticeable delay, a splash screen can reassure the user that something is happening. Choose one because it improves the experience, not simply because Snug supports it.

Size and Duration

A splash screen should generally be large enough to be clearly visible without taking over the user's screen.

Choose an image that is easy to recognise and a display time that does not make startup feel unnecessarily slow. **Section 4: Configuration Options** covers the defaults and size-warning settings.

Configuring the Splash Screen

Add your image and a minimum display time to `snug.options`:

```
--splash "snug\splash.png"
--splash-ms 2500
```

Save the file and rebuild with `.\snug.exe`. This example displays your splash image for at least `2.5` seconds. See **Section 4: Configuration Options** for the full option reference.

For now, remember that the splash screen is part of **your application's experience**. It should look like it belongs to your application rather than advertising the tool that was used to package it.

SECTION 4:

CONFIGURATION OPTIONS

In Section 3, we looked at how to customise the experience of using your application. This section explains how to configure Snug to produce that experience.

We will look at the available options, the values they accept, and how to combine them in `snug.options`.

How Snug Configuration Works

Snug takes its build settings as command-line options. For a small build, you can enter them directly on the command line. For example:

Command(s)

```
.\snug.exe --name "Invoice Viewer" --company "Example Pty Ltd" invoice-viewer.jar
```

Invoice Viewer is a fictional application used for the configuration examples in this section. Replace its JAR and identity values with your own. As the number of options grows, keep them in an options file rather than typing them on every build.

By default, Snug looks for a file named in the directory from where the `snug.exe` was run:

`snug.options`

Using an options file is the recommended way to configure a real application. It keeps settings together and makes builds easier to repeat.

You can generate a starter file by running:

Command(s)

```
.\snug.exe --init-options
```

Snug creates an example `snug.options` file in the current working directory. Open it in a text editor and adjust the settings for your application. If a file already exists, Snug will not overwrite it by default. The file-generation options are described later in this section.

For most Windows x64 builds, use a single `snug.options` file. An optional `snug.windows.options` file can hold Windows-specific settings. For single-value options, precedence is highest first:

Priority	Configuration source	What it does
1	Command line	Overrides values from either options file
2	`snug.windows.options`	Overrides the common options file for Windows builds
3	`snug.options`	Recommended general configuration

Both default options files are read from the current working directory, not from beside `snug.exe`. Using `--options <PATH>` instead selects one specified file and bypasses both default files.

Example: Assume `snug.options` contains:

```
--name "Invoice Viewer"
--company "Example Pty Ltd"
--version "1.0.0"
```

If you then run:

Command(s)

```
.\snug.exe --version "1.1.0" invoice-viewer.jar
```

In this example, the command-line version overrides the value in `snug.options`.

Repeatable options behave differently: `--jvm-arg` and `--localisation` entries accumulate from `snug.options`, then `snug.windows.options`, then the command line. Supplying one on the command line does not remove entries from the options files.

The resulting executable uses the name "`Invoice Viewer`", company "`Example Pty Ltd`", and version "`1.1.0`".



Tip: Start with the command line while you are experimenting. Once your configuration begins to grow, move it into `snug.options`.

Input and Output

Snug needs to know which Java application should be packaged and, optionally, where the finished Windows executable should be written.

Specify the JAR file or directory with either of these forms:

```
--input <JAR|DIR>
```

Or use a positional input:

```
[JAR|DIR]
```

Using --input

For example:

Command(s)

```
.\snug.exe --name "Invoice Viewer" --company "Example Pty Ltd" ↩
  --input invoice-viewer.jar
```

For repeatable builds, put the input in `snug.options`:

```
--input invoice-viewer.jar
```

Using the Positional Input

The equivalent positional command is:

Command(s)

```
.\snug.exe --name "Invoice Viewer" --company "Example Pty Ltd" invoice-viewer.jar
```

The positional input does not have to appear last on the command line.

The positional form is also accepted in an options file. For clarity, this book recommends using `--input` in `snug.options`.

Do not supply both `--input` and positional input on the same command line. A positional input supplied on the command line overrides `--input` from an options file.

Input Source

If the input is a single JAR file, Snug packages that JAR as the application.

If you supply a directory instead, Snug includes the JAR files found directly inside that folder, sorted by filename. It does not search subfolders. This is useful when an application consists of several JAR files rather than a single JAR.

For a multi-JAR application, use `--main-class` to specify the startup class if the JAR manifests do not provide the appropriate entry point.

If you are unsure which class should be used as the application's entry point, Snug can inspect the input JAR or JARs and report classes that declare a conventional Java main method.

Use:

Command(s)
`.\snug.exe --find-main <JAR|DIR>`

This is a diagnostic check only. It does not build the application. The `--find-main` option is covered in more detail later in Build and Diagnostic Tools. An abbreviated output when running this on the `snug-javafx-demo-windows.jar` demonstration JAR is shown below:

```
...

  com.sun.javafx.css.parser.Css2Bin
  synapticloop.snugjavafxdemo.Launcher  <-- --main-class
2 main class(es) found.
```

You will notice in the above output that two main classes were found in the `snug-javafx-demo-windows.jar`, so it is important to ensure that the launcher uses the correct Main-Class when running your Java application. Snug will indicate which jar is configured to be run with the `<-- --main-class` designator.

Output File

The output option is:

```
-o, --output <EXE>
```

This controls the path and filename of the Windows executable that Snug creates.

For a single JAR input, Snug uses the JAR's filename to name the executable unless you specify an output file. For example:

```
invoice-viewer.jar → invoice-viewer.exe
```



IMPORTANT: If the output executable already exists, Snug will replace it with the newly built executable. Snug does not create a backup of the existing file. If you need to keep the previous executable, copy or rename it before building again.

Application Identity

Two options are particularly important because Snug uses them to identify the application:

```
--name <NAME>
--company <COMPANY>
```

The `--name` option sets your application's display name. If omitted, Snug uses the input JAR filename without its .jar extension, or the input directory name.

The `--company` option identifies the organisation or person responsible for the application. If omitted, Snug uses the resolved application name.

Although both options have defaults, we recommend setting them explicitly before distributing your application. Together, they form part of its local identity and determine where Snug stores its cached application files.

On Windows, these values determine the application's local cache path:

```
%LOCALAPPDATA%\snug\<Company>\<Application>\
```

See **Section 7: Technical Details** for cache hashing, payload updates, and cleanup behaviour.



IMPORTANT: Choose a distinct combination of `--name` and `--company` for each application. Reusing the same combination for unrelated applications can cause their cached files to interfere with one another.

The application name also becomes part of the Windows executable metadata, including `ProductName` and `FileDescription`. The company value becomes the Windows `CompanyName` metadata.

Changing the Cache Location

Snug normally chooses a per-user cache directory automatically. Most applications should use this default. The advanced `--cache-dir <ABSOLUTE_DIR>` option sets a different location at build time. It requires an absolute path, because a relative path may not resolve reliably on another person's computer.



IMPORTANT: Avoid changing the cache directory unless you have a specific deployment requirement. A folder that exists on your computer may not exist, or may not be writable, on someone else's. If you override the default, test the location on every target machine. **Section 7: Technical Details** explains how Snug manages the cached files.

Application Metadata

Once you have chosen the application's name and company, you can add details that Windows displays about the executable, along with an icon and an optional link to check for newer versions.

The main options covered here are:

```
--version <VERSION>
--description <TEXT>
--copyright <TEXT>
--update-url <URL>
--icon <PNG/ICO>
```

Version, description, and copyright are stored as Windows executable metadata. The icon provides a visual identity, while the update URL adds a link to the launcher's error dialog. It does not change how application updates are delivered.

Version

The `--version` option sets the Windows `ProductVersion` and `FileVersion` values and accepts between one and four dot-separated numeric components, for example:

1
1.2
1.2.3
1.2.3.4

Windows stores four version components. If fewer than four are supplied, the missing trailing components are set to zero. For example:

1.2.3 → 1.2.3.0

Each component must be between 0 and 65535. Invalid values are rejected before the build starts.

Description

The `--description` option sets the Windows `FileDescription` field. People can see it in places such as `Properties` > `Details` without opening the application.

For example:

```
--description "Internal invoice review application"
```

A short description is usually more useful than repeating the application name.

Copyright

The `--copyright` option supplies the Windows `LegalCopyright` metadata.

For example:

```
--copyright "Copyright 2026 Example Pty Ltd"
```

This is informational metadata stored in the executable. It does not provide code signing, licence enforcement, or any other form of technical protection.

Update URL

The `--update-url` option adds a clickable "Check for a newer version" link to the launcher's error dialog. Set it to a page where people can find a newer release, such as your application's GitHub Releases page. This link appears in the error dialog; it is not an automatic update system. Snug does not download or install new versions of your application.

Application Icon

The `--icon` option accepts a PNG or ICO file to replace Snug's default icon. **Section 3: Customising the Snug User Experience** explains how this affects the user experience, including Windows icon caching.

For PNG artwork, we recommend starting with a high-resolution square image, such as 1024x1024, so Snug has enough detail when producing the Windows icon sizes used by the executable.

If you do not provide an icon, Snug uses its built-in default. For distribution, choose a simple design that stays recognisable at small sizes.

Java Runtime

These options tell Snug which Java version your application requires, which class to launch, and whether the Java Virtual Machine (JVM) needs any special settings. They do not determine how Snug downloads a runtime; that is covered later under Runtime Download Behaviour.

```
--min-java <N>
--main-class <CLASS>
--jvm-arg <OPTION>
```

Minimum Java Version

The `--min-java` option sets the minimum Java major version your application requires. Snug uses this requirement when checking whether a runtime is suitable.

The current default is:

```
25
```

For example, if you set `--min-java 25`, a Java 21 runtime is too old for this build. Whether Snug can offer to download a suitable runtime depends on the separate `--download-jdk` setting, explained later in this section.

A Java 25 runtime meets a `--min-java 21` requirement, but the minimum version alone does not determine which runtime Snug selects. Discovery and the `--download-jdk` mode also affect that choice.

Normally, set this to the oldest Java major version that your application actually requires. Setting a higher value than necessary can cause otherwise suitable installed runtimes to be rejected.

Main Class

Java applications normally identify their startup class through the JAR manifest, but the `--main-class` option can override that value and tell Snug which Java class should be launched.

This is particularly useful for applications containing more than one possible entry point, for multi-JAR builds where the appropriate main class needs to be specified explicitly, or when the JAR manifest does not contain the entry point you intend Snug to launch.

If your JAR already contains the correct `Main-Class` entry in its manifest, you normally do not need to set `--main-class`. Use it when you deliberately want to override that value, or when Snug cannot determine the correct entry point from the packaged JARs.

If you are unsure which class should be used, run `--find-main` before building. It can report classes that declare a conventional `public static void main(String[])` method and can help you decide which value, if any, should be supplied to `--main-class`.

JVM Arguments

The `--jvm-arg` option passes an option directly to the Java Virtual Machine when Snug creates it.

The option can be repeated. For example:

```
--jvm-arg=-Xms256m
--jvm-arg=-Xmx2g
--jvm-arg=-Dfile.encoding=UTF-8
```

These are JVM settings, not arguments passed to your application's own main method.

Use `--jvm-arg` only when your application needs a particular JVM setting. The examples above set the initial and maximum heap sizes and a Java system property.

Be careful when adding JVM arguments. Snug passes them directly to the JVM, so an invalid or unsupported option can prevent the Java runtime from starting. If your application does not require any special JVM settings, you can leave this option out.

Application Instance Behaviour

By default, Snug permits one running instance of an application. Starting the executable again activates the existing application instead of launching another copy.

To allow more than one instance to run at the same time, use:

```
--allow-multiple-instances
```

Add this option to `snug.options`, or supply it on the command line when building.

When `--allow-multiple-instances` is present, each launch can start another copy. Omit it to retain the single-instance default.

Splash Screen Configuration

The Snug splash screen is controlled by three related options:

```
--splash <PNG>
--splash-ms <MS>
--splash-max <WxH|off>
```

Splash Image

The `--splash` option supplies the PNG image that Snug displays while the Java application is being started.

The PNG is embedded in the executable and rendered at its native pixel size, centred on the primary monitor. Snug does not rescale the splash image.

A splash image around 480x270 to 640x360 pixels is a reasonable starting point.

Minimum Display Time

The `--splash-ms` option sets the minimum time, in milliseconds, for which Snug displays the splash screen.

The current default is:

```
1500 milliseconds. That is 1.5 seconds.
```

Snug dismisses the splash screen only when the minimum duration has elapsed and the JVM has signalled readiness. If startup takes longer, the splash remains visible.

Recommended Maximum Size

The `--splash-max` option controls the build-time warning threshold for splash-screen dimensions.

The current default is:

```
640x360
```

This setting is advisory only. It does not resize the image.

If the image exceeds the configured dimensions, Snug issues a build-time warning. The image is still displayed at its original size.

Set a different threshold using the option below, or disable the warning with `off`, `none`, or `unlimited`:

```
--splash-max <WxH>
```

Runtime Download Behaviour

Snug can download runtime components when a suitable Java runtime is not already available.

The runtime download option is:

```
--download-jdk <off|auto|force>
```

Choose the download mode according to your deployment needs:

`off`
Use an available compatible runtime without offering a download. If none is found, the launcher shows an error and exits.

`auto` (default)
Search for a compatible runtime first. If none is found, retrieve download information and ask the user to confirm installation. If accepted, download, verify, and extract the runtime; the user can cancel.

If you include the option, you must supply a mode: `off`, `auto`, or `force`. For example:

```
--download-jdk auto
```

Using `--download-jdk` without a value produces an error.

`force`
Snug skips normal Java discovery initially and uses a suitable cached Snug-managed runtime, or starts the installation flow if none is available. If the installation is cancelled or fails, Snug may still fall back to a compatible locally installed runtime.

Use auto for most applications, `off` where downloads are not permitted, or `force` to prefer a Snug-managed runtime. Force does not guarantee exclusive use of a managed runtime: the fallback described above still applies.

When Snug installs a runtime, it downloads it, verifies the downloaded archive using its SHA-256 checksum, and extracts the runtime beneath:

```
%LOCALAPPDATA%\snug\jdk\<version>\
```

Snug reuses that runtime for future launches.

Runtime Updates

When Snug downloads a Java runtime, it stores it on the computer for reuse. On later launches, Snug checks whether a suitable runtime is already available before attempting another download.

A newer Java release becoming available does not automatically cause Snug to download or install it. If a compatible cached runtime is found, Snug normally continues to use that runtime.

Consequently, a Snug-managed runtime may continue to be used after newer maintenance or security releases become available.

The current launcher does not provide an automatic runtime-update schedule or a runtime-retirement process that tracks running applications and removes older Java versions when they are no longer in use.

Anyone responsible for distributing an application should consider how its Java runtime will receive maintenance and security updates. Application updates and Java runtime updates are separate concerns, and Snug's runtime-update behaviour may evolve in future releases.

Launcher Messages and Localisation

Snug can embed customised launcher messages in different languages. This section explains how to include the files in a build; **Section 5: Customising Launcher Messages** shows how to edit and preview the messages.

The localisation option uses the British spelling:

```
--localisation <TXT|DIR>
```

This option is repeatable.

A single localisation file uses the filename convention:

```
snug-localisations.<tag>.txt
```

where `<tag>` is a BCP 47 language tag such as `en`, `en-US`, `pt-BR`, `de`, or `ja`. The locale is taken from the filename.

You can provide one localisation file, repeat `--localisation` for multiple files, or provide a directory containing them.

When a directory is supplied, Snug scans the top level only. Every file in that directory must match the expected `snug-localisations.<tag>.txt` pattern.



IMPORTANT: A stray file, backup file, or incorrectly named file causes the build to fail so that localisation mistakes are not silently ignored.

Snug always embeds its built-in English baseline. At runtime, the launcher attempts the most specific matching language first, then the primary language, and finally falls back to built-in English.

Snug also warns during the build if a supplied localisation bundle is missing keys that exist in the built-in English baseline.

For example, a project might keep translated launcher messages in:

```
snug\localisation\snug-localisations.de.txt
snug\localisation\snug-localisations.fr.txt
```

and configure Snug with:

```
--localisation snug\localisation
```

Using a directory is convenient when your application supports several languages. **Section 5: Customising Launcher Messages** explains how to edit the messages themselves.

Use `snug_preview.exe` to inspect launcher dialogs without rebuilding the application. It can load a single localisation file or a directory of files, with the available languages selectable from its language dropdown.

Creating Localisation Files

To create a localisations directory containing an English template, run:

Command(s)

```
.\snug.exe --init-localisations
```

To generate a template for another language, run:

Command(s)

```
.\snug.exe --init-localisations --init-localisations-tag de
```

(replace `de` with the appropriate language tag). These commands create templates; they do not translate the text. Edit the values, preview them as described in **Section 5: Customising Launcher Messages**, and include the finished file or directory using `--localisation <TXT|DIR>`.

The optional `--init-localisations-force` and `--init-localisations-stdout` switches control overwriting and output. See the Command-Line Options Reference for their details.

Windows Manifest

Snug also provides a manifest option:

```
--manifest <XML>
```

The manifest is embedded into the Windows executable as an `RT_MANIFEST` resource.

Typical reasons for supplying a manifest include declaring DPI-awareness behaviour, requesting a particular Windows execution level, or supplying application identity information needed by a Windows-specific requirement. Snug does not require you to create a manifest for an ordinary application.

This is an advanced option. If you do not already know that your application requires a custom Windows manifest, you can normally leave it alone.



IMPORTANT: Snug embeds the manifest you provide. Snug checks that the supplied manifest is well-formed XML. You remain responsible for its Windows settings; test the finished executable on your target systems.

Build and Diagnostic Tools

Several options are useful while testing or diagnosing a Snug build.

Dry Run

```
--dry-run
```

Validates the input and configuration and prints what Snug would build, but does not write an executable.

For example:

Command(s)
`.\snug.exe --dry-run invoice-viewer.jar`

Use this before a real build when you want to check the settings without writing or replacing an executable.

Find Main Class

`--find-main`

Scans the input JAR or JARs and lists classes that declare:

`public static void main(String[])`

Snug also indicates whether the main class it would use appears in that list.

This is useful when a fat JAR contains several possible entry points or when you are deciding what value to supply to `--main-class`.

The check is diagnostic only. It does not write files and does not cause a build to fail if the configured main class is not found. This is important because a JavaFX Application subclass can be launched without declaring a conventional main method.

For example, run `--find-main` against `snug-javafx-demo-windows.jar`. An abbreviated result might be:

```
...

  com.sun.javafx.css.parser.Css2Bin
  synapticloop.snugjavafxdemo.Launcher  <-- --main-class
2 main class(es) found.
```

The marked entry:

> The `<-- --main-class` marker identifies the configured entry point. Output varies with the JAR and configuration; classes beyond Snug's scan window may not be reported.

Emit Payload

`--emit-payload`

Writes Snug's encoded embedded payload to standard output instead of writing an executable.

This is primarily an advanced inspection or tooling option and is unlikely to be needed for a normal application build.



Technical: This is intended for advanced tooling and inspection, not ordinary builds. It outputs the encoded payload rather than verifying the final executable by itself.

This is used by the build and verification process for the Snug release pipeline.

Using a Different Options File

To select an alternative options file, use:

`--options <PATH>`

Lines beginning with `#` are comments. Use quotation marks for values containing spaces, such as `--name "Invoice Viewer"`. Backslashes in paths are literal characters; shell-style backslash escaping is not supported.



IMPORTANT: The precedence table and repeatable-option caveat under How Snug Configuration Works also apply when selecting an alternative options file.

Supply `--options` on the command line. It selects one file in place of both defaults; placing it inside an options file does not load another file.

Creating an Example Options File

Snug includes tooling options for creating an example options file:

```
--init-options [<PATH>]
--init-options-force
--init-options-stdout
```

`--init-options` writes the embedded example configuration and exits without building. With no path, it writes `snug.options` in the current working directory; an optional path selects another destination.

Existing files are protected unless you add `--init-options-force`.

- Add `--init-options-stdout` to print the example to the console instead of writing a file.



Tip: Generate the starter file once, then maintain your application's settings in `snug.options`.

Getting Help from the Command Line

You do not need to remember every Snug option because, at any time, you can display Snug's built-in help:

Command(s)

```
.\snug.exe --help
```

Or, for shorter command-line help output:

Command(s)

```
.\snug.exe -h
```

The help output describes the options supported by the exact version of Snug you are running.

Because Snug continues to evolve, this built-in help should be treated as the definitive reference for the installed version.

Other Useful Snug Tooling

Snug Version

`--snug-version`

Prints Snug's own version and exits.

This is different from `--version <VERSION>`, which sets the version metadata for the application being packaged.

A Practical Configuration Workflow

For a new application, a sensible progression is:

1. Generate a starter `snug.options` file with

Command(s)
`.\snug.exe --init-options`

   then set `--input`, `--name`, and `--company`.

2. Run the following command to build the executable:

Command(s)
`.\snug.exe`

   Launch the result to check that your Java application starts correctly.

3. Add version, description, copyright, and icon information.
4. Add a splash screen if the application benefits from one.
5. Decide whether you need to change runtime-download behaviour.
6. Customise launcher messages or localisation if required.

7. Rebuild and test the application from the user's point of view.

This keeps the configuration process incremental, so you do not need to decide everything before your first build.

A Recommended Starting Configuration

Start with the identity, metadata, and icon your application needs:

```
--input build\libs\invoice-viewer.jar
--name "Invoice Viewer"
--company "Example Pty Ltd"
--version "1.0.0"
--description "Internal invoice review application"
--icon snug\icon.png
```

This supplies identity, Windows metadata, and an icon while leaving Java requirements, runtime downloads, and instance behaviour at their defaults.

Add a splash screen, translated launcher messages, a custom manifest, or JVM settings only when needed, then rebuild and test.

Putting It Together

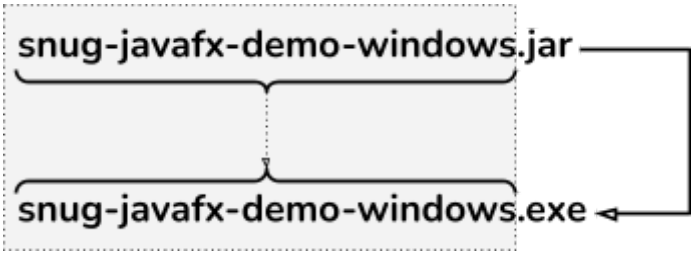
For comparison, this command supplies an explicit identity and JAR path:

Command(s)
`.\snug.exe --name "Invoice Viewer" --company "Example Pty Ltd" invoice-viewer.jar`

The fuller example below adds optional branding, Java requirements, and localisation. Include only the settings your application needs.

```
--input build\libs\invoice-viewer.jar
--name "Invoice Viewer"
--company "Example Pty Ltd"
--version "1.4.0"
--description "Internal invoice review application"
--copyright "Copyright 2026 Example Pty Ltd"
--icon snug\icon.png
--splash snug\splash.png
--min-java 25
--download-jdk auto
--localisation snug\localisation
```

Without `--output`, a single JAR produces an executable beside it: `invoice-viewer.jar` becomes `invoice-viewer.exe`. A directory named `app-jars` produces `app-jars.exe` beside that directory.



After editing `snug.options`, run

Command(s)
`.\snug.exe`

to build the application. Run the resulting executable on another Windows computer to check the launch experience and any runtime-download behaviour.

The Appendix contains the compact command-line reference. This section explains which settings matter and when to use them.

SECTION 5:

CUSTOMISING LAUNCHER MESSAGES

Snug includes English launcher messages, so customisation is optional. Start with the short tutorial below to change and preview one message. The dialog guide explains where messages appear; the Message Key Reference and complete token tables are provided for reference.

Your First Message Change and Preview

First, generate a starter localisation file in PowerShell:

Command(s)
`.\snug.exe --init-localisations`

Open `localisations\snug-localisations.en.txt` in a plain-text editor. Find `runtime.prompt.heading` and change only its value to:

```
runtime.prompt.heading = Install Java to start this application
```

Keep the key exactly as written, then save the file. This heading does not accept replacement tokens.

Next, open Snug Preview to see how your edited message appears. This is a separate program that displays launcher dialogs without requiring a complete application rebuild.

To open the preview, run:

Command(s)
`.\snug_preview.exe --localisation ".\localisations"`

Select English (`en`), then open the runtime installation confirmation dialog. Check that its main heading reads "`Install Java to start this application`".

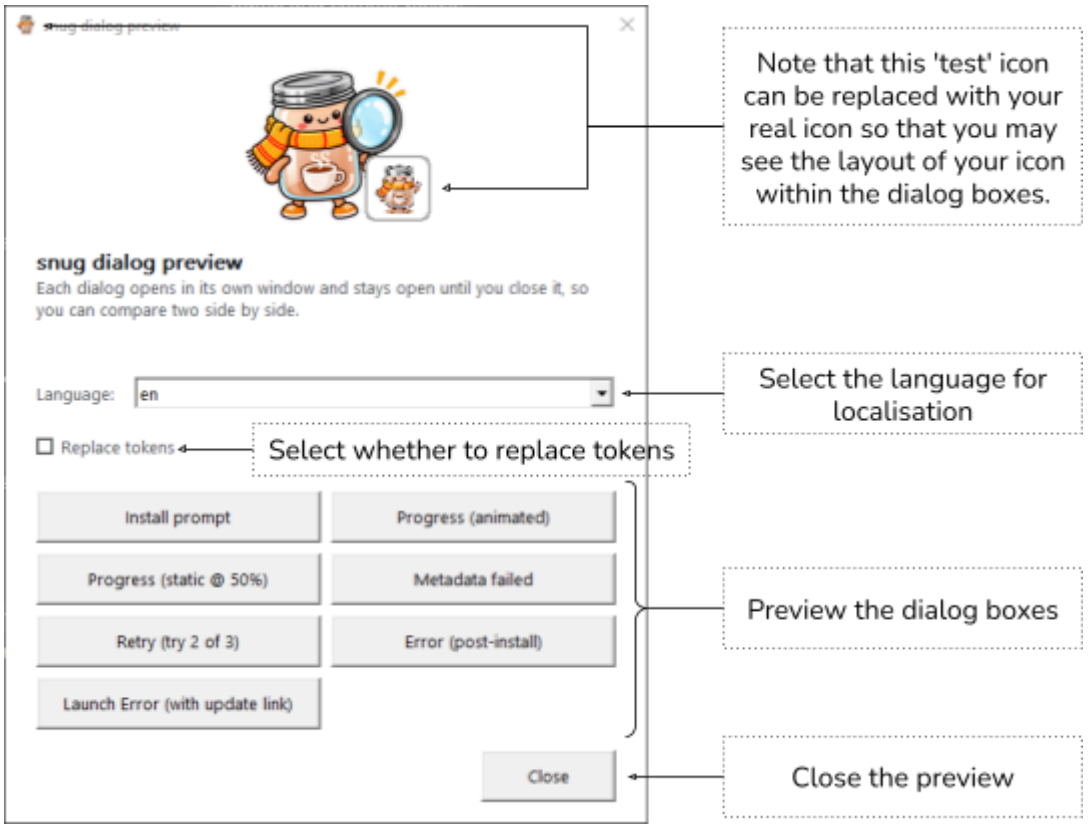


Image: *The running Snug Preview, which allows you to select the dialog that you wish to preview.*

The screenshot uses an optional application icon. To preview your own branding, use the `--icon` option described under **Previewing Your Changes**.

Command(s)

```
.\snug_preview.exe --localisation .\test\localisations\ --icon .\test\test-icon.png
```

This example uses the Snug project's test files; replace these paths with your localisation directory and icon.

Understanding Launcher Dialogs

Most launches are uneventful: the person using your application double-clicks its executable, and the Java application starts. When Snug needs to communicate, its dialogs fall into two main groups.

Runtime Installation Dialogs

These dialogs appear when Snug needs to obtain a compatible Java runtime. Snug first asks the user to confirm installation. If the user agrees, a progress window shows the download, verification, and extraction stages.

If installation encounters a problem, Snug may display a retry or failure dialog.

Cannot Launch Dialogs

The Snug launch-error window uses `launcher.error.*` keys for its wording and the relevant `err.*` key for the underlying error. Built-in English is available when custom localisation is unavailable. Windows failures before the Snug launcher has initialised are separate: these localisation keys do not control Windows' own messages.

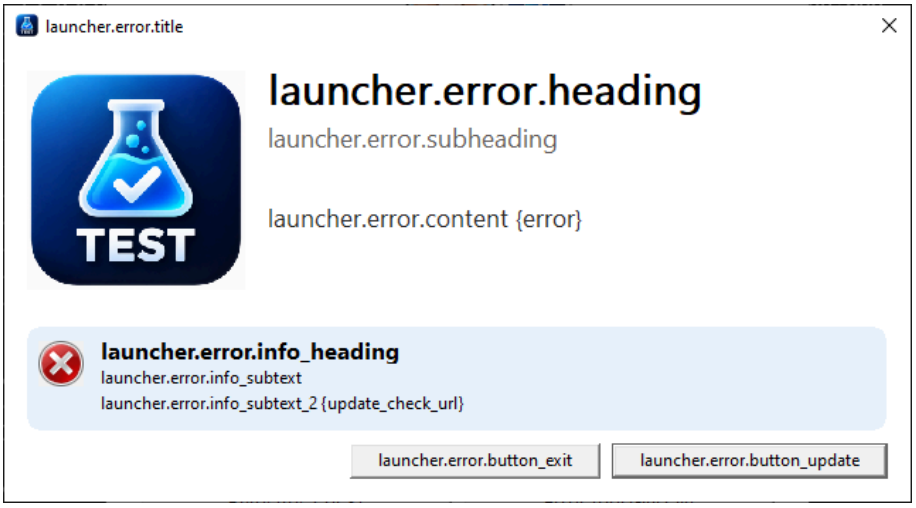


Image: The Cannot Launch dialog appears when the Snug-packaged application cannot be launched.

Customising the Messages

You can leave the default English messages alone, replace their wording with language that suits your application, or supply translations. The messages are stored as named entries in a localisation file. Each entry has a key that Snug recognises and a value containing the wording shown to the user. Some values also contain replacement tokens, such as `{version}`, which Snug fills in when the dialog appears.

For example, the structure of a message entry is:

```
runtime.progress.pct_label = {pct}%
```

Here, `runtime.progress.pct_label` is the message key, and `{pct}` is a replacement token. Snug supplies the current percentage when showing the progress dialog. When editing a localisation file, leave both the key and supported replacement tokens intact. Change the surrounding words to suit the people using your application. This is a teaching example of a custom value, not a quotation of Snug's default English wording.

A Simple Editing Workflow

1. Start with a copy of a Snug localisation file for the language you want to customise. Keep the original keys unchanged.
2. Open the copy in a plain-text editor and change the values for the messages you want to personalise. You do not need to rewrite every message.
3. Save the file, then use `snug_preview.exe` to inspect the changed wording and buttons.
4. Add the localisation file or directory to `snug.options`, build your executable, and check the result.



IMPORTANT: If a translation is missing a message, Snug uses the available fallback wording for that message, ultimately falling back to built-in English. Other translated messages remain available.

For example, to use a directory of language files, include an option like this in `snug.options`:

```
--localisation "snug\localisation"
```

Section 4: Configuration Options gives the accepted values and full option reference.

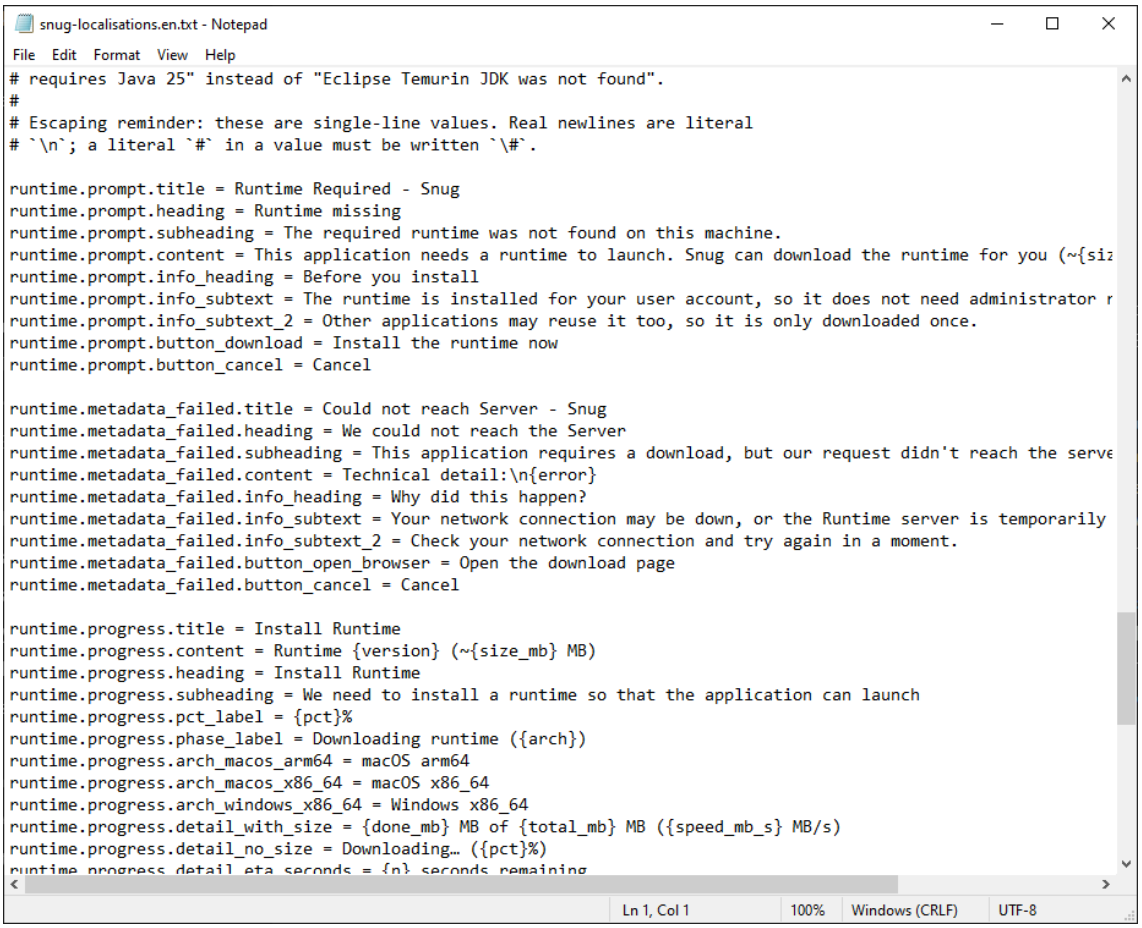


Image: Edit the text after the equals sign, not the key.

Which Messages Can I Change?

The messages cover runtime installation and errors that prevent an application from launching. You can customise the wording of headings, explanations, and supported button labels. Changing a message does not change what a button does or how Snug handles an error.



IMPORTANT: Localisation changes the wording of Snug's dialogs, including supported button labels. It does not change button actions or error handling.

Use the following dialog guide to identify the message you want to change. For exact key names and token support, use the reference tables later in this section.

Understanding Dialog Layouts

Most Snug runtime dialogs share a common layout. Rather than learning a completely different set of localisation keys for every window, you can recognise the same parts by their key suffixes. The dialog's prefix tells Snug which window the text belongs to; the suffix identifies where the text appears.

For example, `runtime.progress.heading` identifies the main heading in the installation progress window.

Before downloading a required runtime, Snug displays a confirmation dialog where the user can choose to install it or cancel. The wording in this dialog uses the `runtime.prompt.*` localisation keys.

If the user agrees, Snug opens the progress window and begins downloading, verifying, and extracting the runtime.

These screenshots show examples of the localisation keys available for each dialog and where their messages appear. In normal use, Snug displays the message text associated with each key.

The annotated example below shows the common parts of a runtime dialog. Not every window uses every field: the progress dialog also shows download information, while the Snug launch-error window uses its own `launcher.error.*` message keys.



Image: *The anatomy of a runtime dialog window showing the key suffixes and where they are placed.*

Common Localisation Key Suffixes

`.title` - The title displayed in the dialog window.

`.heading` - The main heading, which tells the user what is happening.

`.subheading` - Supporting text immediately below the main heading.

`.content` - The main explanation or instructions.

`.info_heading` - A heading for additional information.

`.info_subtext` - The first line of additional information.

`.info_subtext_2` - The second line of additional information.

`.button_<name>` - The label for an action button, such as `.button_cancel` or `.button_download`. Buttons vary according to the dialog.

For example, `runtime.progress.content` identifies the main explanation in the installation progress window. The key determines where the text appears; a replacement token such as `{version}` determines which information Snug can insert when the dialog is displayed.

Keep the complete key, including its prefix and suffix, unchanged when editing a localisation file. Change only the text after the equals sign. The message tables later in this section list the actual keys available for each dialog and any replacement tokens they support.

The Runtime Installation Dialogs

Runtime installation is the most interactive part of Snug's launcher. Normally the user only encounters it when a compatible runtime is unavailable. Once the runtime has been obtained and cached, later launches can reuse it without repeating the installation.

The normal installation path is:

→ Runtime required
→ Retrieve runtime download information
→ Ask the user to confirm installation
→ Download, verify, and extract the runtime
→ Start the application.

If Snug cannot retrieve the download information, it can offer to open the runtime download page. If installation fails, a retry dialog can offer another attempt or cancellation.

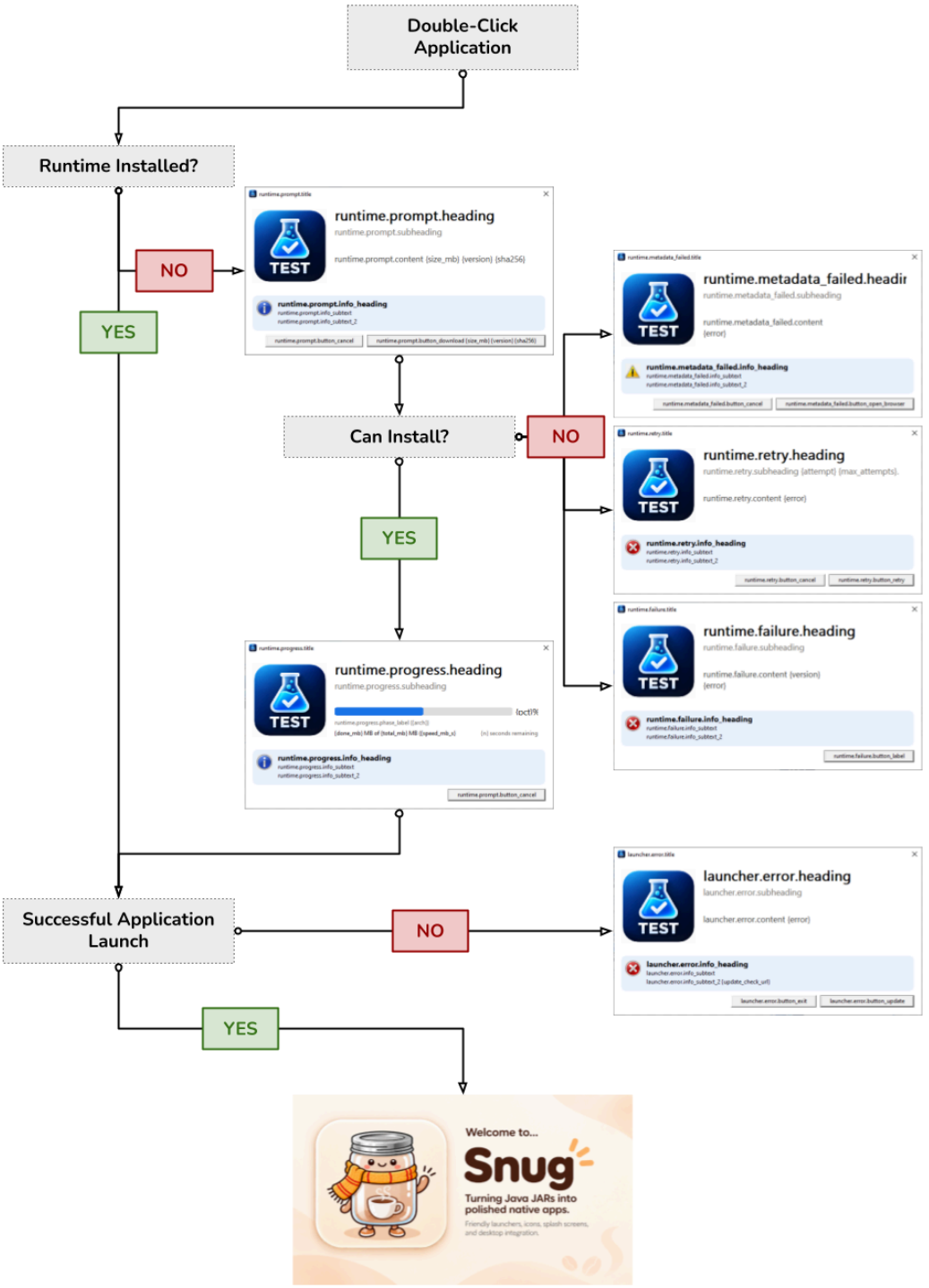


Image: *Where Snug may need to ask the user for help.*

When Snug Cannot Launch

Snug's launch-error window reports failures handled by the launcher, such as a missing application file or a Java application that cannot start. A failure reported by Windows before launcher initialisation is outside this dialog reference.

You can customise the launcher error window using `launcher.error.*` keys. Snug fills `launcher.error.content` with the underlying error message, which can itself be translated using the appropriate `err.*` key. If the embedded localisation files cannot be read, the window falls back to built-in English.

Message Key Reference

The reference begins with selected keys for each dialog, followed by token meanings, the complete token-bearing key table, and all no-token keys. `REPLACEMENT_TOKENS.md` in the Snug repository records the machine-checked token contract. For another Snug version, check its reference:

https://github.com/synapticloop/snug/blob/main/REPLACEMENT_TOKENS.md

Or you can run the command

Command(s)
`.\snug.exe --list-tokens`

This prints the replacement-token reference for the version of Snug you are using.

Keep every key exactly as shown. “Allowed tokens” lists the tokens a key supports; “Default English tokens” lists only those used by the supplied English wording. “None” means no tokens in that column. You may use any allowed token, even if the default wording omits it.

Runtime Installation Confirmation Keys

Use `runtime.prompt.*` for the installation confirmation dialog.

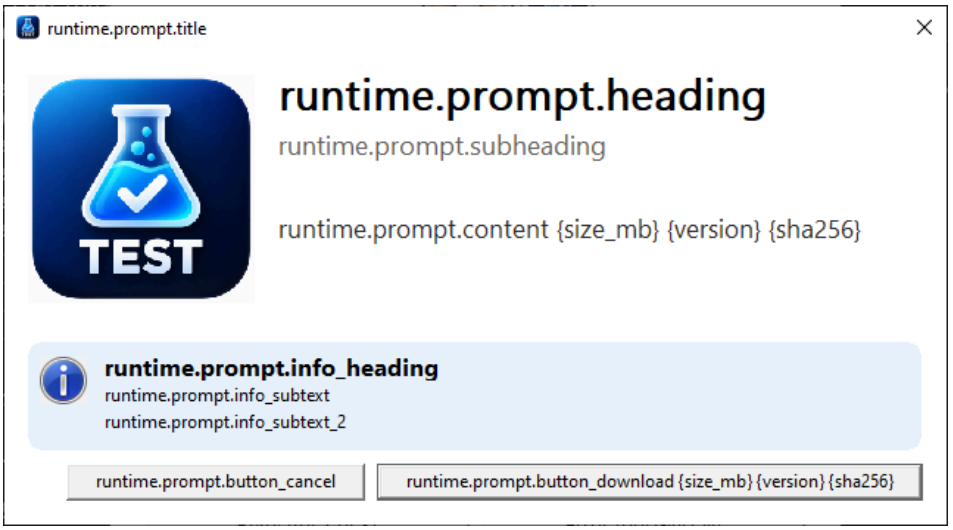


Image: *The anatomy of a runtime dialog window showing the key suffixes and where they are placed.*

The selected keys below cover its explanation and action buttons. The complete reference later includes its headings and information text.

Localisation Key	Description and allowed tokens
`runtime.prompt.content`	the explanation; may use `{version}`, `{size_mb}`, and `{sha256}`.
`runtime.prompt.button_cancel`	the cancel button (no replacement tokens).
`runtime.prompt.button_download`	the download button; may use `{version}`, `{size_mb}`, and `{sha256}`

Downloading and Preparing the Runtime

Use `runtime.progress.*` for the download, verification, and extraction progress window.

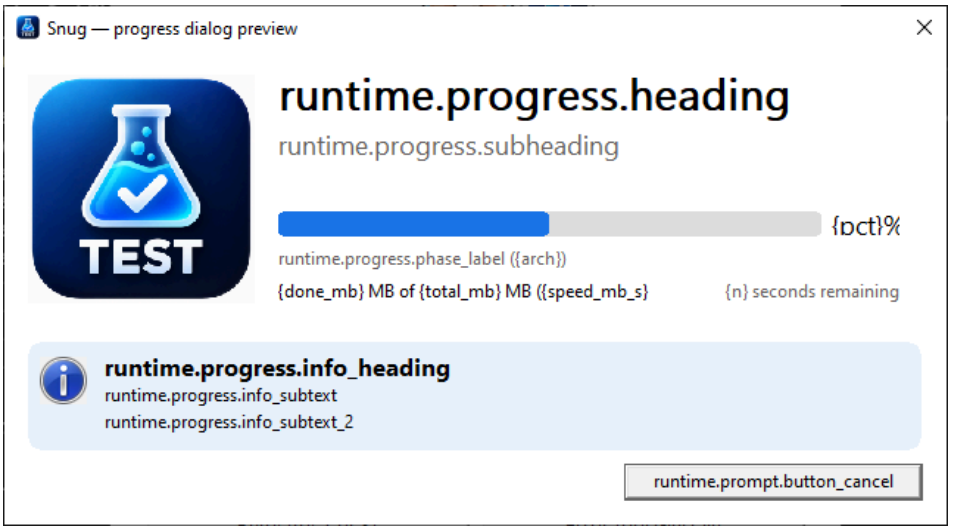


Image: *The runtime installation progress window, showing the localisation keys used for its messages and progress information.*

The progress dialog also supports title, heading, subheading, and informational text keys. These fields do not accept replacement tokens.

Localisation Key	Description and allowed tokens
`runtime.progress.content`	the main progress message; may use `{version}` and `{size_mb}`.
`runtime.progress.phase_label`	the current download phase; may use `{arch}`.
`runtime.progress.pct_label`	the percentage display; may use `{pct}`.
`runtime.progress.detail_with_size`	downloaded amount and speed; may use `{done_mb}`, `{total_mb}`, and `{speed_mb_s}`.
`runtime.progress.detail_no_size`	progress when a total size is unavailable; may use `{pct}`.
`runtime.progress.detail_eta_seconds`	remaining seconds; may use `{n}`.
`runtime.progress.detail_eta_done`	the message shown when the download is almost finished (no replacement tokens).

When the Download Service Cannot Provide the Runtime

This dialog may appear when Snug cannot retrieve runtime release information, for example, because the network is unavailable or the download service cannot be reached.

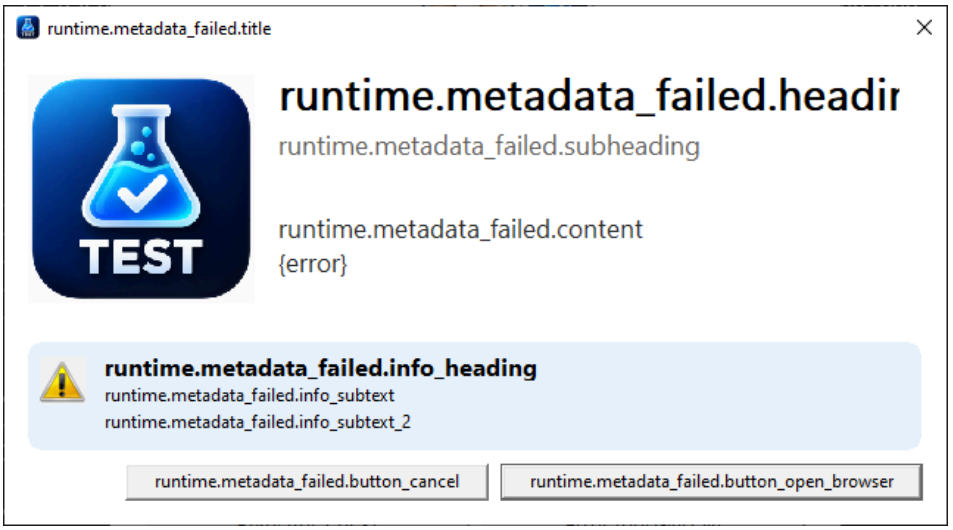


Image: *The anatomy of a runtime dialog window showing the key suffixes and where they are placed.*

Use `runtime.metadata_failed.*` when runtime release information cannot be retrieved. The subheading allows `{major}`; the content allows `{major}` and `{error}`. The other keys in this group accept no tokens.

The default English content is `Technical detail:\n{error}`. It uses `{error}` only; a custom value may also use `{major}`.

Localisation Key	Description and allowed tokens
`runtime.metadata_failed.subheading`	`{major}`
`runtime.metadata_failed.content`	Allowed tokens: `{major}`, `{error}`. Default English tokens: `{error}`.
`runtime.metadata_failed.button_cancel`	None
`runtime.metadata_failed.button_open_browser`	None

Retrying a Runtime Download

Use `runtime.retry.*` for the retry dialog following a download, verification, or extraction error. Its token contract is listed below.



Image: *The anatomy of a runtime retry dialog window showing the key suffixes and where they are placed.*

Localisation Key	Description and allowed tokens
`runtime.retry.subheading`	`{version}`, `{attempt}`, `{max_attempts}`, `{error}`
`runtime.retry.content`	`{version}`, `{attempt}`, `{max_attempts}`, `{error}`
`runtime.retry.button_retry`	None
`runtime.retry.button_cancel`	None

When Runtime Installation Cannot Complete

Use `runtime.failure.*` for the final installation failure dialog. Metadata retrieval and retry dialogs have separate message groups.

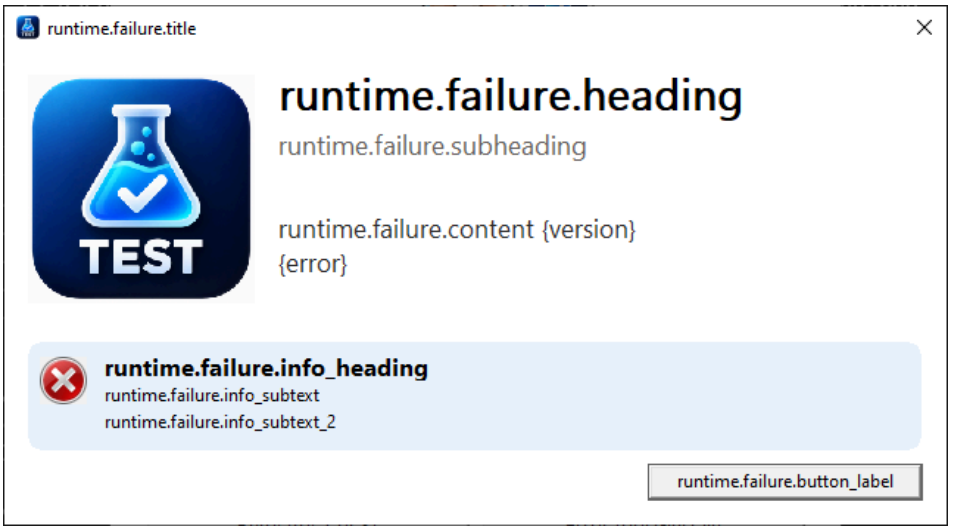


Image: The runtime installation failure dialog, showing the localisation keys used for its messages and button.

`runtime.failure.content` allows `{version}` and `{error}`. The retry entries below are repeated for convenience; their contracts are unchanged.

Localisation Key	Description and allowed tokens
`runtime.failure.content`	the failure explanation; may use `{version}` and `{error}`.
`runtime.retry.subheading`	the retry message; may use `{version}`, `{attempt}`, `{max_attempts}`, and `{error}`.
`runtime.retry.content`	additional retry information; may use `{version}`, `{attempt}`, `{max_attempts}`, and `{error}`.
`runtime.retry.button_retry`	the retry button (no replacement tokens).
`runtime.retry.button_cancel`	the cancel button (no replacement tokens).

Snug Launcher Error Window

Use `launcher.error.*` for Snug's launch-error window. These keys do not customise failures reported by Windows before launcher initialisation.

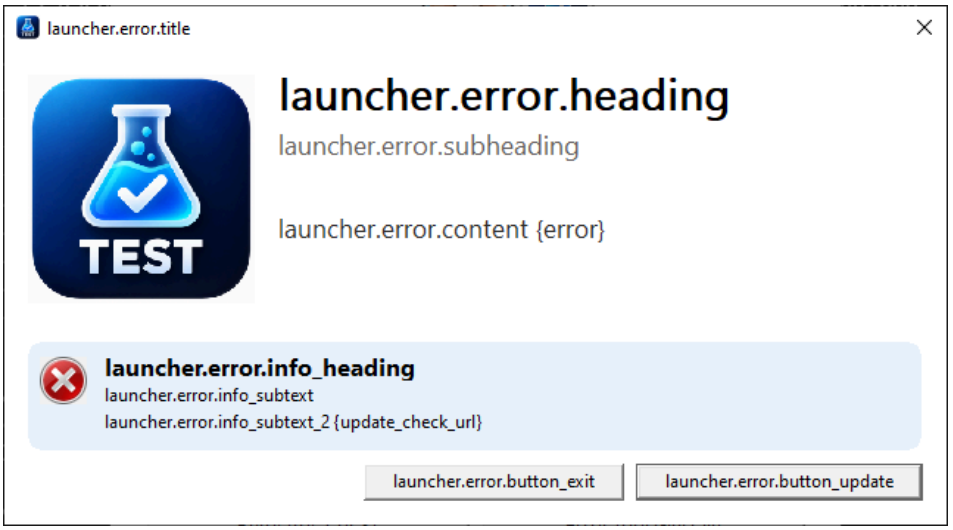


Image: *The launcher error window, showing the localisation keys used for its messages and buttons.*

Localisation Key	Description and allowed tokens
`launcher.error.content`	the explanation; may use `{error}`.
`launcher.error.info_subtext_2`	additional information; may use `{update_check_url}`.
`launcher.error.button_exit`	the exit button (no replacement tokens).
`launcher.error.button_update`	the update button (no replacement tokens).

The selected dialog keys above are a starting point. The complete reference below also covers Java, file, splash-screen, and platform errors handled by Snug.

Replacement Tokens and the Key Reference

Replacement Token	What Snug puts in its place
`{0}`	positional slot - the single argument this key is handed, usually an error message
`{actual}`	byte length the pixel buffer turned out to have

Replacement Token	What Snug puts in its place
`{arch}`	target architecture label, e.g. "Windows x64"
`{attempt}`	which attempt this is, counting from 1
`{computed}`	the SHA-256 the launcher calculated from the downloaded file
`{declared}`	the SHA-256 the publisher advertised
`{done_mb}`	megabytes downloaded so far
`{error}`	the underlying error, already localised when it is a launcher error
`{expected}`	byte length the buffer's dimensions require
`{found}`	the Java major version actually found on the machine
`{library}`	the library that failed to load
`{major}`	the Java major version the runtime metadata lookup was for
`{max_attempts}`	total number of attempts configured
`{message}`	the OS-level error text explaining the failed load
`{min_java}`	the Java major version the application requires
`{n}`	whole seconds of estimated time remaining
`{name}`	name of the metadata field that was malformed
`{path}`	a filesystem path that was located, or searched
`{pct}`	completion percentage as a whole number
`{sha256}`	the SHA-256 digest the download is verified against
`{size_mb}`	approximate download size in megabytes
`{speed_mb_s}`	current download speed in MB/s
`{symbol}`	the exported symbol that was missing from it
`{total_mb}`	total megabytes to download
`{update_check_url}`	the URL supplied through the builder's --update-url option, used to check for a newer release

Replacement Token	What Snug puts in its place
`{value}`	the value the launcher received for that field
`{version}`	the JDK release being downloaded or installed, e.g. "25.0.1+9"



IMPORTANT: A replacement token is only available where Snug supplies its value. For example, `{size_mb}` is supported by the download-confirmation and current progress messages, but not by `runtime.retry.subheading`. If you put an unsupported or misspelt token into a message, Snug leaves it visible literally instead of replacing it. Keep the braces, spelling, and case exactly as documented.

Keys and Their Allowed Tokens

This complete table lists every token-bearing key. “Allowed tokens” defines what a custom value may use; “Default English tokens” describes the supplied English value. None means no tokens in that column. Tokens are optional, but any token you use must be allowed for that exact key.

Localisation File Key	Allowed tokens	Default English tokens
`err.format`	`{0}`	`{0}`
`err.invalid_state`	`{0}`	`{0}`
`err.io`	`{0}`	`{0}`
`err.java_exception`	`{0}`	`{0}`
`err.jni_attach`	`{0}`	`{0}`
`err.jni_create`	`{0}`	`{0}`
`err.jni_init`	`{0}`	`{0}`
`err.jni_invoke`	`{0}`	`{0}`
`err.main_class_not_found`	`{0}`	`{0}`
`err.no_main_method`	`{0}`	`{0}`
`err.self_path`	`{0}`	`{0}`

Localisation File Key	Allowed tokens	Default English tokens
err.zip	{0}	{0}
err.jvm_not_found	{min_java}	{min_java}
err.jvm_too_old	{path} {found} {min_java}	{path} {found} {min_java}
err.library_load	{library} {message}	{library} {message}
err.symbol_not_found	{symbol} {library}	{library} {symbol}
jdk.err.bad_metadata_shape	{0}	{0}
jdk.err.dialog	{0}	{0}
jdk.err.download	{0}	{0}
jdk.err.extract	{0}	{0}
jdk.err.io	{0}	{0}
jdk.err.metadata_fetch	{0}	{0}
jdk.err.no_java_exe	{0}	{0}
jdk.err.no_metadata_for_version	{0}	{0}
jdk.err.bad_field	{name} {value}	{name} {value}
jdk.err.sha256_mismatch	{declared} {computed}	{computed} {declared}
splash.err.thread_spawn	{0}	{0}
splash.err.buffer_length	{expected} {actual}	{actual} {expected}
launcher.bare_stub.no_payload	{path}	{path}
launcher.error.content	{error}	{error}
launcher.error.info_subtext_2	{update_check_url}	{update_check_url}

Localisation File Key	Allowed tokens	Default English tokens
runtime.prompt.content	{version} {size_mb} {sha256}	{size_mb}
runtime.prompt.button_download	{version} {size_mb} {sha256}	None
runtime.metadata_failed.subheading	{major}	None
runtime.metadata_failed.content	{major} {error}	{error}
runtime.progress.content	{version} {size_mb}	{version} {size_mb}
runtime.progress.phase_label	{arch}	{arch}
runtime.progress.pct_label	{pct}	{pct}
runtime.progress.detail_with_size	{done_mb} {total_mb} {speed_mb_s}	{done_mb} {total_mb} {speed_mb_s}
runtime.progress.detail_no_size	{pct}	{pct}
runtime.progress.detail_eta_seconds	{n}	{n}
runtime.failure.content	{version} {error}	{version} {error}
runtime.retry.subheading	{version} {attempt} {max_attempts} {error}	{attempt} {max_attempts}
runtime.retry.content	{version} {attempt} {max_attempts} {error}	{error}

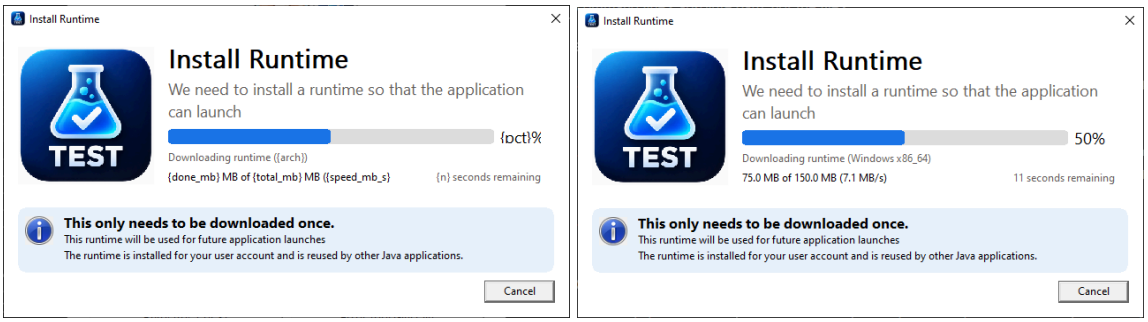


Image: *Message keys identify the text shown in the progress dialog. Replacement tokens insert live values, such as the runtime version, download size, and percentage complete.*



IMPORTANT: Keys not listed in the token-bearing table do not accept replacement tokens. A token placed in one of those keys is displayed literally. The remaining keys are listed by group below so that this chapter can be used as a complete reference.

Keys Without Replacement Tokens

The keys below are also available in the English localisation catalogue. They do not accept replacement tokens. Keep the key names unchanged, and edit only their values.

The table groups no-token keys by dialog or purpose, with one key per line.

Dialog Window	Localisation Key

Launcher and Java errors	`err.no_main_class`
	`err.unsupported_platform`
	`launcher.bare_stub.hint`
	`launcher.bare_stub.command`

Snug launcher error window	`launcher.error.title`
	`launcher.error.heading`
	`launcher.error.subheading`
	`launcher.error.info_heading`
	`launcher.error.info_subtext`
	`launcher.error.button_exit`
	`launcher.error.button_update`

Dialog Window	Localisation Key

Runtime installation progress	`runtime.progress.title`
	`runtime.progress.heading`
	`runtime.progress.subheading`
	`runtime.progress.arch_macos_arm64`
	`runtime.progress.arch_macos_x86_64`
	`runtime.progress.arch_windows_x86_64`
	`runtime.progress.detail_eta_second`
	`runtime.progress.detail_eta_done`
	`runtime.progress.info_heading`
	`runtime.progress.info_subtext`
	`runtime.progress.info_subtext_2`

Runtime metadata, retry, and failure dialogs	`runtime.metadata_failed.title`
	`runtime.metadata_failed.heading`
	`runtime.metadata_failed.info_heading`
	`runtime.metadata_failed.info_subtext`
	`runtime.metadata_failed.info_subtext_2`
	`runtime.metadata_failed.button_open_browser`
	`runtime.metadata_failed.button_cancel`
	`runtime.retry.title`
	`runtime.retry.heading`
	`runtime.retry.info_heading`
	`runtime.retry.info_subtext`
	`runtime.retry.info_subtext_2`
	`runtime.retry.button_retry`
	`runtime.retry.button_cancel`
	`runtime.failure.title`
	`runtime.failure.heading`
	`runtime.failure.subheading`
	`runtime.failure.info_heading`
	`runtime.failure.info_subtext`
	`runtime.failure.info_subtext_2`
	`runtime.failure.button_label`

Runtime installation confirmation	`runtime.prompt.title`
	`runtime.prompt.heading`
	`runtime.prompt.subheading`
	`runtime.prompt.info_heading`
	`runtime.prompt.info_subtext`
	`runtime.prompt.info_subtext_2`
	`runtime.prompt.button_cancel`

Dialog Window	Localisation Key

Splash, platform, and generic messages	`splash.err.overflow`
	`splash.err.empty`
	`splash.err.bitmap`
	`splash.err.window`
	`splash.err.update_layered`
	`splash.err.create_compatible_dc`
	`splash.err.create_dib_section`
	`splash.err.create_window`
	`splash.err.update_layered_window`
	`splash.title`
	`generic.error_dialog_ok`
	`generic.info_dialog_continue`

Previewing Your Changes

You do not need to rebuild an executable every time you change a sentence. The separate `snug_preview.exe` utility can show launcher dialogs in their intended context, including unusual runtime-error states that are inconvenient to reproduce during ordinary testing.

By default, the preview uses the Snug Preview icon. To see the dialogs with your application icon, start it with:

Command(s)

```
.\snug_preview.exe --icon "my-app.ico"
```

To load a single localisation file or an entire directory, use `--localisation`. The language dropdown always includes English (en). Loading one German (de) file therefore offers de and en; loading a directory makes each available language selectable.

Command(s)

```
.\snug_preview.exe --localisation "snug\localisation" --icon "my-app.ico"
```

Open the preview with your localisation file or directory, select the language, and inspect each changed dialog.

Check that headings explain the situation, buttons describe their actions, and translations fit. Snug Preview displays sample dialog states; also test the packaged application before release.



Note: Close all open preview dialogs and restart `snug_preview.exe` after editing a localisation file to see your latest changes.

Other Languages

Localisation allows the same Windows executable to show launcher messages suited to the language selected by Windows. This is particularly useful when the Java application itself already supports more than one language.

Keep each language's messages in its own localisation file, or provide a directory containing the files. The built-in English messages remain available as a fallback when Snug cannot select a suitable translation. You can preview each language before building.

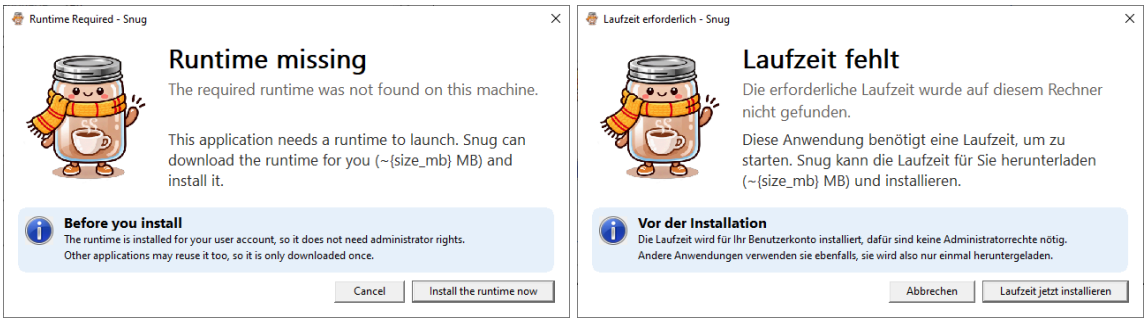


Image: *The same runtime dialog shown in English and German. The German wording is an illustrative translation for testing.*

Before You Build: Final Checks

Before building, check:

- Keep every localisation key unchanged; edit only its value.
- Use only tokens allowed for that exact key, with braces, spelling, and case intact.
- Save each language file and restart Snug Preview to load your edits.
- Preview every changed dialog and language; check headings, line wrapping, button labels, and unexpected literal tokens.
- Have translations checked by a fluent speaker.
- Supply the intended file or directory through `--localisation` in `snug.options` (see **Section 4: Configuration Options**).
- Build and run the packaged application on Windows; check the messages in the launch and runtime scenarios you can reproduce. Preview alone does not verify runtime behaviour.

SECTION 6:

MANAGING YOUR SNUG PROJECT

This optional section helps you maintain, rebuild, share, and continue developing your application. You do not need it to build or use a Snug application.

Managing Your Snug Project

As your project grows, keep its packaging files together in a predictable place. A simple convention is to create a snug directory at the root of your project:

```
.\snug\
```

A typical project might look like this (the built JAR is generated by your Java build tool and is not shown here):

```
my-application\
  ├── build.gradle / build.gradle.kts / pom.xml
  ├── snug.options
  ├── src\
  ├── ...
  └── snug\
    ├── snug.exe
    ├── icon.png
    ├── splash.png
    └── localisations\
```

Keeping these files together separates your application's source and Snug packaging resources from generated output. The built JAR may be produced in `build\libs\` (Gradle) or `target\` (Maven); point Snug at its actual location using `--input` or a positional JAR argument.

What Should Go in the snug Directory?

Typical contents include:

- `snug.exe` - the Snug builder
- `icon.png` - the application icon, if you use one
- `splash.png` - the Snug splash screen, if you use one
- `localisations\` - translated launcher messages, if you use localisation
- other packaging resources, such as a custom Windows manifest

Keep `snug.options` and any `snug.windows.options` overrides in the project root. Open a terminal there and run `.\snug\snug.exe`: the executable is in the snug subdirectory, but the current working directory remains the project root. Snug looks for its options files and resolves relative paths from that working directory, not from the executable's location. Running Snug elsewhere will not automatically load this project's options files.

The exact filenames are up to you; the important part is consistency.

What Should Be Saved with the Project?

Some files describe how your application should be built and packaged; others are tools or generated output.

Keep the Java source, build instructions, Snug configuration, and project-specific assets with the project. For packaging, this includes `snug.options`, any snug.`windows.options` overrides, icons, splash screens, localisation files, and manifests.

If you are using Git, these files should normally be committed to the repository so that the Snug setup is preserved alongside the application source code. Generated JARs, build folders, and finished Windows executables normally do not need to be committed; distribute the finished executable separately, for example through a GitHub release.

Using Git

Git is a version-control system used to keep a history of changes to a project. You do not need Git in order to use Snug, but if you are already using it as part of a larger application project, a useful rule is to keep the files needed to recreate the project rather than every file it produces.

For Snug, this usually means keeping the configuration and assets in Git while making a deliberate choice about the `snug.exe` binary itself.

Should snug.exe Be Committed?

There are two reasonable approaches, depending on how self-contained you want the project to be.

Keep `snug.exe` Outside Git

Keep `snug.exe` locally in the `snug` directory and exclude it from Git. Add this entry to `.gitignore`:

```
/snug/snug.exe
```

This avoids storing a new copy of the binary in the repository each time Snug is upgraded.

Keep snug.exe in Git

For a small project, committing the exact `snug.exe` makes it more self-contained: someone cloning it can use the same Snug version without a separate download.

The trade-off is repository size: Git retains historical copies of the binary when it changes.

Neither approach is inherently wrong; the important thing is to know which version of Snug the project expects.

A Practical Starting Point

Use the directory layout above as a starting point: keep the options files and packaging assets with the project, and decide whether to commit `snug.exe` or download the recorded version when needed.

This preserves the Snug setup without requiring every project-management decision at once.

Reproducible Builds

As a project becomes more important, it helps to be able to recreate a working executable months later. Keep the Java source, build instructions, Snug configuration, assets, and the versions of your build tools together. Here, reproducibility means being able to repeat the build reliably - not a guarantee that two executables will be byte-for-byte identical.

In a short build-notes file, record the exact Snug release, JDK version, and Gradle or Maven version, plus the commands used to build the JAR and run Snug from the project root. Keep `build.gradle`, `build.gradle.kts`, or `pom.xml`, as appropriate, and any build-tool wrapper files with the project. Record where to obtain the chosen Snug binary if it is excluded from Git. Note the expected JAR path and preserve the options files and assets used for each release.

Automating the Build

Once you can repeat the local build from those instructions, you can optionally automate the same steps. You may encounter the term CI/CD: an automated system runs build or release steps for you. Services such as GitHub Actions can run them when a project changes or a release is created.

Snug does not require CI/CD; a normal local build is perfectly valid.

If you do automate a Snug build, use a specific Snug version rather than automatically downloading whatever happens to be the latest release. This helps ensure that the same source and configuration continue to produce predictable results.

A good automated process might:

- obtain a known version of Snug,
- build the Java application,
- run Snug with the project's saved configuration,
- and keep the resulting executable as a build or release artefact.

The important principle is the same whether the build is manual or automated: keep the instructions for creating the application with the project, and make the versions of the tools deliberate rather than accidental.



Tip: Before you finish, save the source, build instructions, options files, packaging assets, and tool versions. Check that you can rebuild the JAR and run `.\snug\snug.exe` from the project root. Distribute the finished executable separately; Git and automation remain optional.

SECTION 7:

TECHNICAL DETAILS

This optional section explains how Snug packages and launches a Java application.



Tip: You can skip this section and still complete the earlier build and configuration exercises.

Snug Technical Details

The following pages separate the executable's contents from the steps performed when it runs.

What Snug Actually Builds

When you run `snug.exe`, Snug does not convert your Java application into native Windows machine code. Your application remains a Java application; instead, Snug creates a native Windows executable that contains the Snug launcher together with the Java application and the information needed to launch it.

The executable contains the following components; this is a structural outline, not a launch sequence:

- Windows executable (container)

  Contains:
  - Native Snug launcher code

    Alongside:

  - Windows resources, including configuration and application payload

- Payload includes:
  - Java application JAR files

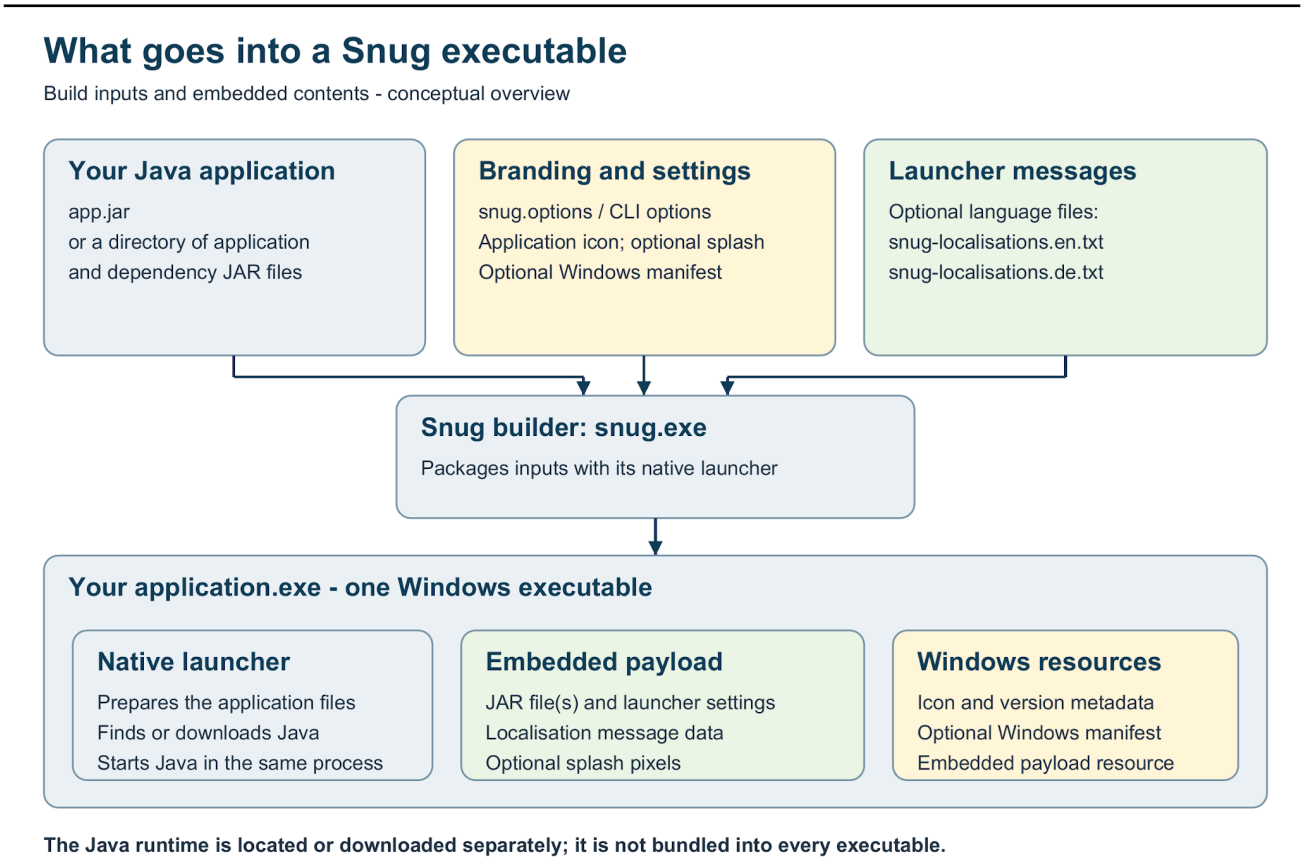


Image: *Executable contents - native launcher code alongside Windows resources and the embedded Java payload.*

The advantage of this design is that the person receiving the application sees a normal Windows executable, while the Java application itself can remain packaged in the familiar JAR format.

The Builder and the Launcher

There are two distinct pieces of Snug involved in this process.

The Builder

`snug.exe` is the builder, and its job is to take your JAR file or files, your Snug configuration, and any additional resources such as an icon, splash screen, or manifest, and produce the finished Windows executable.

Once the executable has been built, the person receiving your application does not need `snug.exe`.

The Launcher

The Snug launcher is the native Windows program that becomes part of the executable created by the builder. Its job begins when the user double-clicks the finished .exe, at which point it reads the embedded configuration, prepares the packaged application, locates or obtains a suitable Java runtime, and starts the Java application.

This separation is important:

- `snug.exe` builds the Windows executable.

- The embedded Snug launcher runs the application.

Anatomy of the Windows Executable

A Snug-built application is a normal Windows Portable Executable, usually referred to simply as a PE executable.

PE sections organise code and data within an executable. Windows resource types classify entries in its resource directory; they are not PE sections.

A Snug executable contains:

- the native launcher code
- Windows executable metadata
- the application icon
- an optional Windows manifest
- an optional Snug splash image
- Snug configuration
- the packaged Java application files

The Java application data is stored inside the executable as Windows resources rather than as loose files alongside the .exe.

As the Quick-Start output indicates, Snug embeds the application payload as a named `RT_RCDATA` resource.



Technical: `RT_RCDATA` identifies application-defined binary resource data; `RT_MANIFEST` identifies a Windows manifest resource. These are resource types, not PE section names.

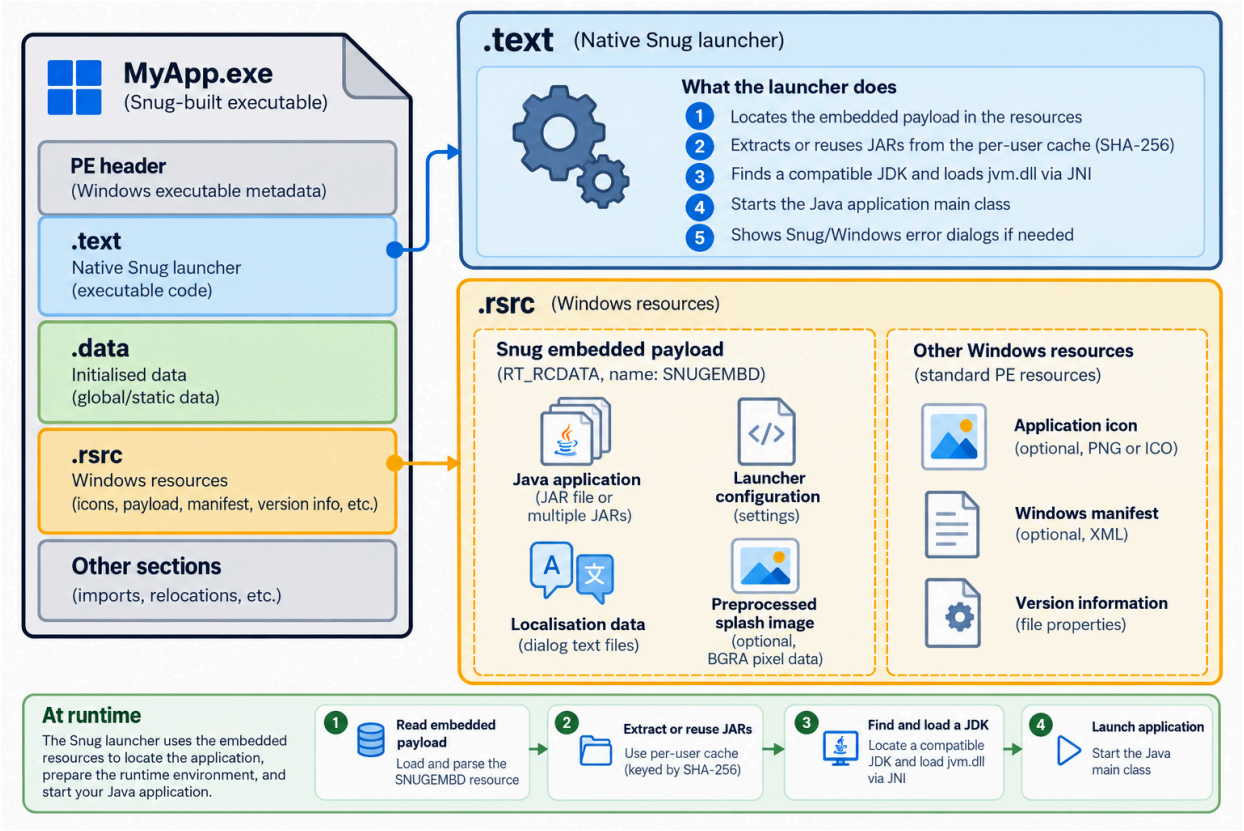


Image: *A Snug-built Windows executable contains the native launcher and Windows resources. Its embedded payload carries the Java application and the configuration needed to launch it.*

Windows Resources

Several of the options described earlier in this book become Windows resources in the finished executable.

For example:

- the application icon becomes an executable icon resource
- version, company, and description information become Windows version metadata

- a custom manifest becomes an `RT_MANIFEST` resource
- packaged application data is carried as application-defined resource data
- the Snug splash image is available to the native launcher before Java starts

This is why Windows can display information about the executable before the Java application has run at all.

The Java Application Remains a JAR

As explained in "**What Snug Actually Builds**", your Java application remains in JAR form.

The launcher prepares that JAR for Java and provides the native launch experience.

This also means that normal Java concepts still matter. Your JAR still needs to be a valid runnable Java application, and the runtime used by Snug must be suitable for that application.

What Happens When the Executable Starts

When a user double-clicks a Snug application, Windows starts the native Snug launcher, which performs a sequence of preparation steps before handing control to the Java application.

The following is a conceptual runtime sequence, not a description of the executable's internal layout or a verified call order:

1. Read the embedded Snug configuration.
2. Identify the packaged application.
3. Prepare or reuse the cached application files.
4. Determine whether a suitable Java runtime is available.
5. Obtain runtime components if the configuration allows it and they are required.
6. Start the Java runtime.
7. Launch the configured Java application.
8. Run the Java application within the launcher process.

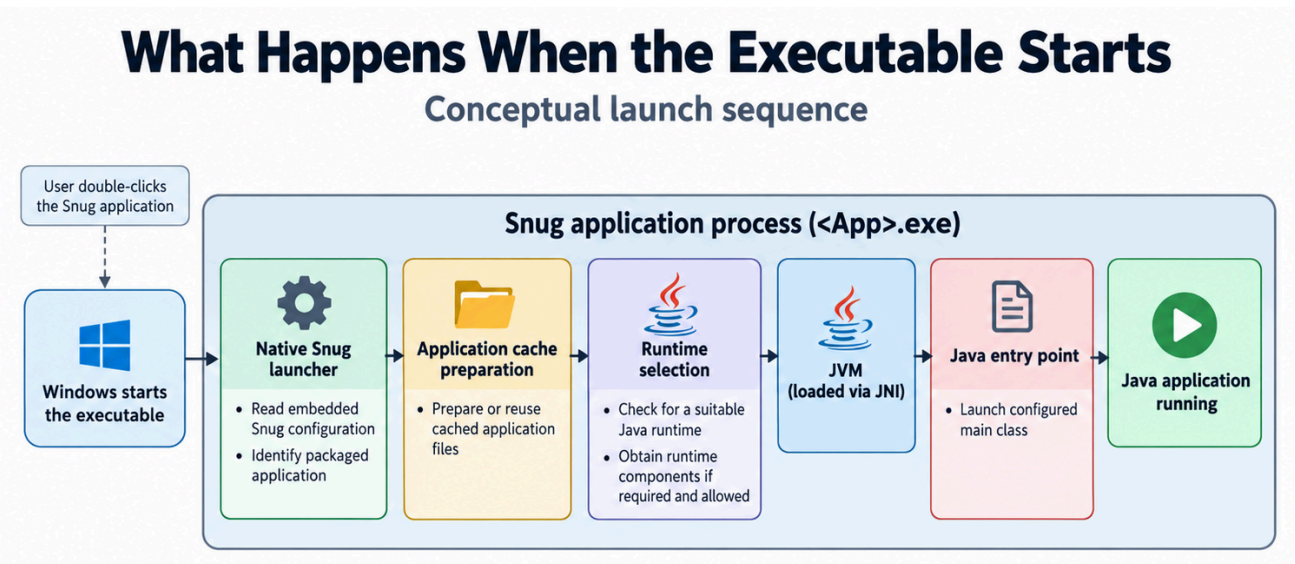


Image: *The Snug launcher prepares the application files and obtains a compatible Java runtime before starting the Java application.*

Application Identity and the Cache

The `--name` and `--company` values are more than cosmetic labels.

They are used by the launcher as part of the identity of the application on the user's computer.

In the current Windows implementation, Snug stores prepared application data beneath the user's local application-data area. A typical path follows this general pattern:

```
%LOCALAPPDATA%\snug\<Company>\<Application>\...
```

Within that application area, Snug can keep prepared application files associated with the packaged JAR.

The embedded JAR hashes distinguish packaged content. The build-time calculation and launch-time cache checks have different purposes, as described below.

How Snug Identifies Cached JARs

During a build, Snug calculates a SHA-256 hash for each JAR and stores that hash alongside the JAR contents in the executable's embedded payload. Each JAR therefore has its own content-based cache identifier. The hash is calculated when Snug builds the executable, not when someone launches it.

At launch, the stored hashes identify cached JARs. The launcher checks that each cached file exists and has the expected size, reusing matching files and extracting replacements otherwise. It does not recompute cached-file SHA-256 on each launch, so these checks do not guarantee content integrity.

For example, a diagnostic log may contain entries such as:

```
app: Synapticloop / Snug JavaFX Demo
primary JAR sha256: <hash>
cached jars: 1
```

Cache Lifecycle and Cleanup

Application-cache cleanup runs automatically in the background, subject to a minimum interval between sweeps. Snug records when each JAR is used and considers recent use, retained entry count, and a per-application storage budget when removing older entries. The sweep protects every JAR belonging to the current build. Cleanup is best-effort; files that cannot be removed may remain until a later sweep.

Application JAR caches and downloaded runtime caches serve different purposes. Runtime maintenance is covered below.

The exact directory names, retention thresholds, and cleanup intervals are implementation details and may change in later versions of Snug.



IMPORTANT: Use distinct `--name` and `--company` values for unrelated applications. These values form part of the application's local identity; confirm the exact directory mapping against the Snug version being used.

Where Snug Stores Its Files

By default, Snug uses the current user's Windows Local AppData area. A configured cache-directory override may change application-cache placement; the paths below illustrate the default layout.

The base directory is:

```
%LOCALAPPDATA%\snug
```

`%LOCALAPPDATA%` is a Windows environment variable that points to the local application-data directory for the currently signed-in user. On a typical Windows installation, the Snug directory will therefore resolve to a location similar to:

```
C:\Users\YourUserName\AppData\Local\snug
```

The following is a schematic default layout, not an exact directory listing:

```
%LOCALAPPDATA%\snug
  │
  ├── jdk
  │     └── ...
  │
  └── <Company Name>
      └── <Application Name>
        ├── jar(s)
        ├── logs (perhaps snug.log)
        └── ...
```

Downloaded Java runtime components are stored in:

```
%LOCALAPPDATA%\snug\jdk
```

This runtime is stored separately from individual applications so that Snug can reuse it for future application launches rather than downloading another copy for every application.

Files belonging to an individual application are stored beneath the company and application names supplied when the executable was built. For example:

```
%LOCALAPPDATA%\snug\SynapticLoop\Snug JavaFX Demo
```

Within an application's directory, Snug keeps the prepared JAR files, launcher logs, and other cached files it needs to start and diagnose the application.

In normal use, you should not need to modify anything inside these directories manually. Knowing where they are stored is useful, however, when troubleshooting an application, examining its launcher logs, or understanding which files Snug has cached.

Why Use a Cache?

Preparing JAR files on disk once and reusing them avoids repeated extraction.

The cache also provides a writable location for launch files and diagnostics when the executable's own directory is unsuitable.

Finding a Java Runtime

A Java application cannot run without a compatible Java runtime, so the launcher needs to determine whether an appropriate runtime is already available or whether runtime components need to be obtained.

The required Java version is part of the packaged launch configuration, and at startup Snug can inspect the available runtime situation and apply the runtime-download behaviour configured when the executable was built.

The user-facing result was described in **Section 3: Customising the Snug User Experience**. Technically, the important point is that runtime selection happens in the native launcher before the Java application itself is started.

Downloaded Runtime Components

Snug deliberately does not have to place a complete Java runtime inside every application executable. When runtime download is enabled, required runtime components can instead be obtained when necessary and reused for future launches, keeping the packaged application separate from the runtime environment that executes it.

For organisations distributing several Java applications, this can also avoid every executable carrying its own full copy of Java.

Known limitation: an installed JVM that is too old can stop runtime discovery before Snug offers an automatic download. If this occurs, check the launcher log and make a compatible Java runtime available, or rebuild with --download-jdk force to prefer the managed-runtime path. Test the chosen approach on the target Windows computers.

The managed download provider is Eclipse Temurin, obtained through the Adoptium API.

The builder defaults to a minimum Java major version of 25. Use `--min-java` to set the minimum required by your application; see **Section 4: Configuration Options**.

Reuse Across Launches

Later launches can reuse prepared JARs and a suitable cached runtime. See "**How Snug Identifies Cached JARs**" for the distinction between build-time hashes and launch-time checks.

The packaged download configuration controls whether Snug may obtain a runtime. After the user agrees, Snug downloads the archive to disk and compares its SHA-256 checksum with the checksum supplied by Adoptium before extraction. A mismatch stops that attempt. Download, verification, and extraction failures can lead to a retry or failure dialog; cancellation stops the attempt and discards its temporary files. This validates the archive against the provider's checksum, rather than guaranteeing that every later launch revalidates the installed runtime.

Changing a JAR changes its build-time hash and identifies new application content; it does not by itself require a new Java runtime.

Runtime Maintenance

Cached runtimes do not receive automatic maintenance or security updates.

Publishers should plan and test runtime updates separately from application updates.

Starting Java

Once the files and runtime are ready, the launcher loads jvm.dll directly and uses JNI (the Java Native Interface) to create a JVM and invoke the Java entry point in the same Windows process. It does not start `java.exe` as a separate application process. JVM creation and invocation errors can be returned to the launcher; after a normal return, the launcher asks the JVM to shut down.

For a JavaFX application, JavaFX initialises and the application's normal windows, stages, and controls are created by the Java application itself.

This is also the point at which an application-managed Java or JavaFX splash screen may appear if the application has been configured to use one.

The Snug splash and a Java application splash therefore belong to different layers:

1. Snug splash (Native launcher)
2. Java/JavaFX splash (Java application startup)

That distinction is why the Snug splash can appear before Java itself has finished starting.

When the Java Application Exits

Normal shutdown is controlled by your Java application and the JVM.

With the in-process JVM model described above, a fatal JVM or native-library crash can terminate the launcher process too. Errors returned during preparation or JVM creation may be reportable; if the process terminates, it cannot display a later failure dialog.

When investigating an unexpected shutdown, check the launcher log for startup information, then consult the application's own logs and any JVM crash report for details about the failure.

Application Arguments

Application arguments are values passed to the Java entry point. They differ from builder options, which configure the executable, and JVM options, which configure the runtime. Whether the application accepts a filename or another argument depends on its own code.



Technical: The standard Snug builder enables argument forwarding by default. The launcher uses Windows command-line parsing, omits the executable name, and passes the remaining arguments to Java in their original order. There is no builder command-line switch for changing this default; custom launcher payloads can disable forwarding.

For example, if your application accepts a filename, you could run:

```
Command(s)
.\MyApplication.exe "C:\Documents\example.txt"
```

With argument forwarding enabled, the filename is passed to the Java application, which decides how to interpret it.

The Windows Manifest

A Windows manifest is metadata that tells Windows about particular behaviours expected by an executable, such as DPI awareness or the requested execution level. When `--manifest` is supplied, Snug embeds it into the finished executable at build time, allowing Windows to inspect and use the manifest before the Java application starts.

If you do not have a specific Windows requirement that needs a custom manifest, there is normally no reason to create one.



IMPORTANT: Snug checks that the supplied manifest is well-formed XML and rejects malformed XML during the build. This check does not confirm that the manifest contains valid or appropriate Windows application settings.
You remain responsible for the manifest's contents. Test the finished executable on the Windows systems you intend to support.

Logging and Diagnostics

The Snug launcher writes diagnostic information that can be useful when an application does not start as expected.

A launcher log can include the following information. The excerpt below is shortened for readability; `[...]` marks omitted text, not literal log output:

- the application and company identity
- the application version
- the cache location
- the SHA-256 hash of the primary JAR
- the number of cached JARs
- the required Java version
- runtime-download behaviour
- whether arguments are forwarded
- the Java main class
- errors encountered during preparation or launch

```
[1790232128] app: SynapticLoop / Snug JavaFX Demo
[1790232128] primary JAR sha256: [...]
[1790232128] cached jars: 1
[1790232128] forward args: true
[1790232129] JDK cache hit: [...] (meets min_java=25)
[1790232129] loading jvm.dll from: [...]\bin\server\jvm.dll
```

For normal users, this information should remain out of the way.

For the person diagnosing a problem, however, the log can answer a much more useful question than "it did not open": it can show how far the launcher got and what it was trying to do when the problem occurred.

Troubleshooting a Launch

If a Snug application builds successfully but does not launch correctly, a useful technical troubleshooting sequence is:

1. Confirm that the same JAR runs correctly outside Snug.
2. Confirm that the Snug application was built with the intended configuration.
3. Collect the launcher log, application logs, and any JVM crash report before changing or deleting caches; cache deletion may remove diagnostic evidence. Record the Snug version and packaged configuration.
4. Look for the detected Java requirement and runtime information.
5. Check whether the packaged application files were prepared successfully.
6. Check the reported main class and application arguments.
7. Rebuild with the current version of Snug if the executable was produced by an older release.

The aim is to separate a Java application problem from a packaging or launcher problem. If the JAR itself does not run correctly, packaging it with Snug will not repair the Java application; if the JAR runs normally but the Snug executable does not, the launcher log is the best place to start.

Code Signing Is Separate from Packaging

Creating a Windows executable and digitally signing one are two different operations. Snug can create the executable and add metadata such as the company name, description, icon, and version, but that metadata does not constitute a digital signature.

A code-signing certificate is used to cryptographically identify a publisher and allow Windows to verify that a signed executable has not been altered since it was signed.

The `--company` option therefore should not be confused with code signing. Setting:

```
--company "My Company"
```

adds application identity used by Snug and Windows metadata, but it does not cause Windows to treat the executable as digitally signed by My Company.

If code signing is required by your organisation or distribution process, it is a separate step performed after the Snug executable has been created.

Why Snug Does Not Bundle Everything

It would be possible to build a Java desktop package that contains an entire Java runtime inside every application, but Snug deliberately takes a different approach. The application executable contains the launcher and application payload, while a suitable runtime can be located or obtained separately. There are trade-offs to both approaches.

Bundling a runtime can make an application more self-contained, but it can also make every application considerably larger. Reusing runtime components keeps the individual application artefact smaller and avoids duplicating the same runtime across multiple applications.

Snug's design is therefore focused on providing a small, familiar Windows entry point while managing the Java requirements behind it.

Keeping the Layers Separate

The easiest way to understand Snug technically is to think in layers:

Windows	Starts the native executable and reads its Windows resources.
Snug launcher	Prepares the packaged application, manages the runtime requirement, and starts Java.
Java runtime	Provides the environment needed to execute the Java application.
Java application	Contains the behaviour and user interface that you actually built.

Each layer has a different responsibility, and this separation is one of the reasons Snug can remain largely invisible to the person using the application.

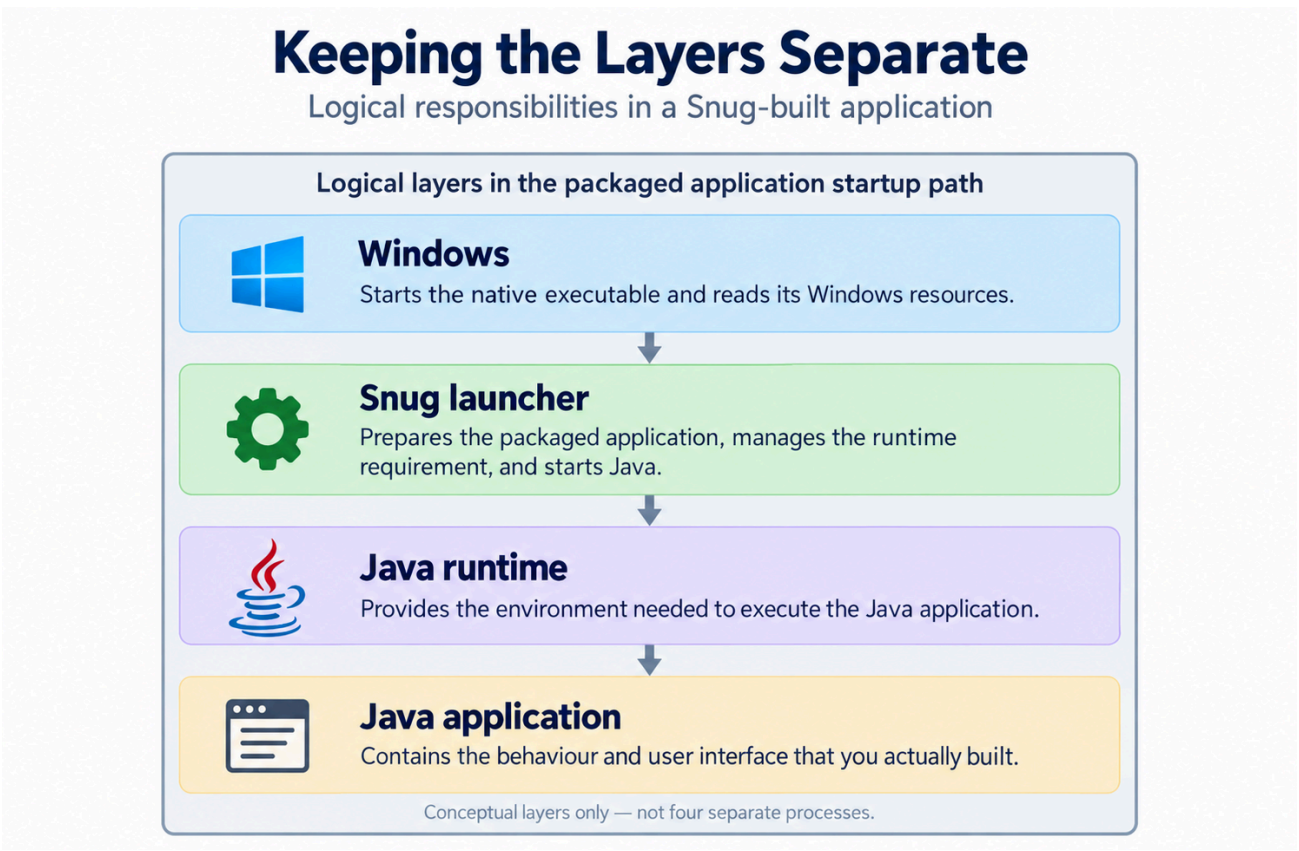


Image: *Windows, the Snug launcher, the Java runtime, and your application each have a different responsibility in starting and running the packaged application.*

Technical Details Will Evolve

Implementation details can change between releases. Use the source revision and release notes that match your Snug binary when checking cache behaviour, resource identifiers, runtime selection, and JVM invocation. The runtime-discovery limitation described under Finding a Java Runtime also applies here.

For a particular build, consult:

- the `--help` output from the Snug version being used
- the launcher log produced by that application
- the release notes and documentation for that version

For implementation details, consult the matching source-code revision. The executable contains the native launcher and Windows resources; at runtime, the launcher prepares the application and starts Java.

Snug packages the application and manages its launch; Java runs the application.

SECTION 8:

APPENDICES

Command-Line Options Reference

Use this reference to look up builder and preview options. Generic names such as My Application stand for your own project; Invoice Viewer continues the worked configuration example from **Section 4: Configuration Options**. The installed tool's `--help` output is the definitive reference for its version.

The overview highlights common builder options; the detailed entries that follow cover the complete command-line reference. For worked examples, see **Section 4: Configuration Options**.

Quick Overview: Common Builder Options

Option(s)	Notes
Minimum Input Required for a Normal Build	

`--input <JAR|DIR>`
 	
Or use positional input
 	
`[JAR|DIR]`
	Input source: either a single JAR or a directory containing JAR files.
	
	Positional alternative to `--input`. The two forms are mutually exclusive.

Application Identity	
`--name <NAME>`	Application display name. Used in Windows metadata and Snug's per-user application identity.
`--company <COMPANY>`	Company or vendor name. Used in Windows metadata and Snug's per-user application identity.
Output	
`-o`, `--output <EXE>`	Output Windows executable path. If omitted for a single-JAR build, Snug uses the JAR filename as the basis for the `.exe` name.

Option(s)	Notes
`Application Metadata` `--version <VERSION>`	Application version. Accepts one to four numeric components, each from 0 to 65535. Missing trailing components default to zero.
`--description <TEXT>`	Windows `FileDescription` metadata.
`--copyright <TEXT>`	Windows `LegalCopyright` metadata.
`--update-url <URL>`	Adds a clickable "Check for a newer version" link to launcher error dialogs.
`--icon <PNG/ICO>`	PNG or ICO file used as the Windows executable icon.

Java Runtime

`--min-java <N>`	Minimum required Java major version. Default: `25`.
`--main-class <CLASS>`	Overrides the Main-Class read from the JAR manifest.
`--jvm-arg <OPTION>`	Extra JVM option passed when the JVM is created. Repeatable.

Windows Resources

`--manifest <XML>`	Windows application manifest embedded as `RT_MANIFEST`.
`--splash <PNG>`	PNG splash image shown by the native launcher before the JVM starts.
`--splash-ms <MS>`	Minimum splash duration in milliseconds. Default: `1500`.
`--splash-max <WxH|off>`	Recommended maximum splash dimensions used for a build-time warning. Default: `640x360`. The setting does not resize the image.

Runtime Behaviour and Localisation

Option(s)	Notes
`--download-jdk <off|auto|force>`	Controls Snug's managed Java-runtime download behaviour. Default: `auto`. Use off only when a compatible runtime will already be available on the target computer.
`--localisation <TXT|DIR>`	Embeds launcher localisation bundles. May be repeated for individual files or supplied with a directory containing `snug-localisations.<tag>.txt` files.
`--allow-multiple-instances`	Allows more than one copy of the packaged application to run at the same time. By default, Snug applications are single-instance.

Build and Diagnostic Tools

`--dry-run`	Validates inputs and reports what would be built without writing an executable.
`--find-main`	Lists classes in the input JAR or JARs that declare `public static void main(String[])` and identifies the main class Snug would use when applicable.
`--emit-payload`	Writes the encoded embedded payload to standard output instead of building an executable.
`--list-tokens`	Lists all tokens that can be used in the localisation files for replacement.

Snug Tooling Options

`--snug-version`	Prints Snug's own version and exits.
`--options <PATH>`	Selects one options file instead of automatic lookup. See the detailed `--options` entry for lookup order and precedence.
`--init-options [<PATH>]`	Writes Snug's embedded example options file and exits. If no path is supplied, the default is `./snug.options`.

Option(s)	Notes
`--init-options-force`	Allows `--init-options` to overwrite an existing destination file.
`--init-options-stdout`	Prints the example options configuration to standard output instead of writing a file.
`-h`, `--help`	Prints Snug's command-line help.

Command-Line Practical Reference

The detailed reference below explains each option's syntax, purpose, default behaviour, and recommended use.

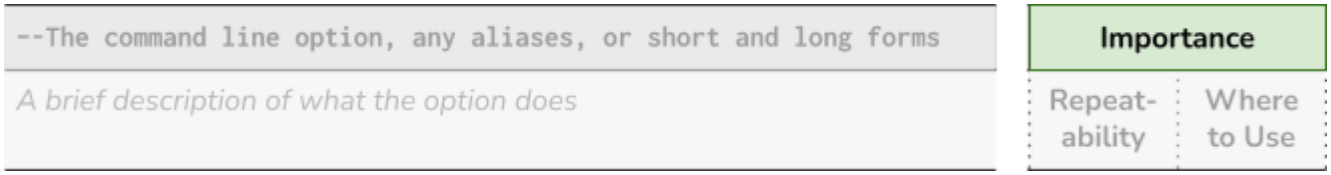
Reading the Reference

Each entry uses the same fields: option name, importance, purpose, repeatability, where to use, syntax, when to use, behaviour if omitted, example, and important note.

`Option name, aliases, and short or long forms`	**Importance**	
A brief description of what the option does	**Repeatability**	**Where to Use**

Syntax	`The option followed by its value, if required. Angle brackets mark a value to supply; square brackets mark an optional value.`
When to use	A description of when to use this option
If omitted	What happens when the option is absent from both the command line and the loaded options files, including any default or build requirement.
Example	`An example of the option and any parameter values`
Important note	Constraints, interactions, or cautions to consider when using the option.

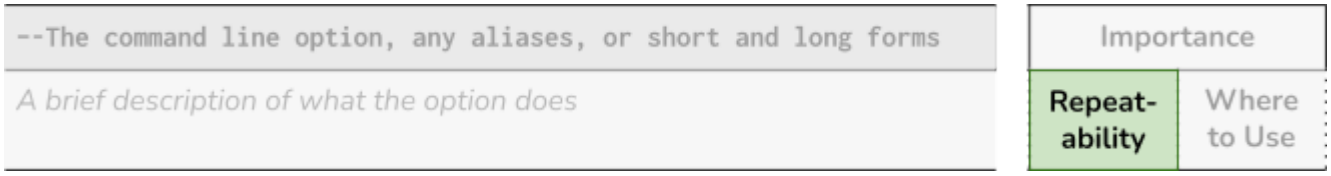
Understanding Importance



Importance describes the option's role in a normal application build:

- **Required** - a normal build needs this information. Input may be supplied by `--input` or the positional `[JAR|DIR]` form; information and scaffolding commands do not require application input.
- **Recommended** - set this deliberately for a distributed application, even though Snug can build without it.
- **Optional** - use this when the application or workflow needs it. Optional settings can still affect build success or runtime behaviour.

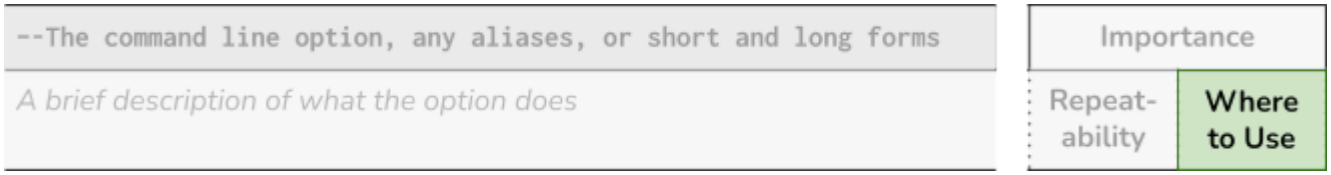
Understanding Repeatability



Repeatability describes how many values an option is intended to receive. It does not mean that duplicate single-value options are always rejected.

- **Single** - supply one effective value or flag. Avoid duplicate occurrences.
- **Multiple** - repeat the option to supply additional values. Builder `--jvm-arg` and `--localisation` values accumulate across loaded options files and the command line.

Understanding Where to Use



Where to Use distinguishes saved build settings from commands best entered directly:

- **Options** - reserved for settings that belong only in `snug.options`. No entry in this reference currently uses this classification.
- **CLI only** - recommended placement for tooling, scaffolding, information, and diagnostics. This is a usage recommendation, not a blanket parser restriction: some diagnostic flags also work in options files, while file-selection and init flags receive special handling. Snug Preview reads its flags directly from the command line.
- **Both** - use the setting in a loaded options file or on the builder's command line. Command-line single-value settings take precedence; repeatable `--jvm-arg` and `--localisation` values accumulate.

Snug Builder (`snug.exe`) Options Reference

`--input`	Required input	
Selects the JAR file or directory of JAR files that Snug should package.	**Single**	**Both**

Syntax	`--input <JAR|DIR>`
When to use	Use it when you want the input source to be explicit, especially in `snug.options`.
If omitted	**Packaging fails if neither `--input` nor positional input supplies a JAR file or directory.**
Example	`--input build\libs\my-app.jar`
Important note	The positional [JAR|DIR] form is also accepted in an options file. For clarity, this book recommends using `--input` in `snug.options`.

`--name`	Recommended	
Sets the application's display name, used in Windows metadata and Snug's local application identity.	**Single**	**Both**

Syntax	`--name <NAME>`
When to use	Always set this to the real application name for a normal build.
If omitted	Snug uses the input JAR filename without its .jar extension, or the input directory name.
Example	`--name "My Application Name"`
Important note	Together with `--company`, this value identifies the application in Windows metadata and Snug's application cache.

`--company`	Recommended	
Sets the company or vendor name for Windows application identity.	**Single**	**Both**

Syntax	`--company <COMPANY>`
When to use	Always set this to the real company or vendor name for a normal build.
If omitted	Snug uses the resolved application name.
Example	`--company "My Company"`
Important note	Together with `--name`, this value identifies the application in Windows metadata and Snug's application cache.

`-o, --output`	Recommended	
Sets the path and filename of the Windows executable that Snug creates.	**Single**	**Both**

Syntax	`--output <EXE>`
When to use	Use it when you want to control the output filename or location.
If omitted	Snug chooses the default output name for the build which is based on the input JAR file(s) or directory name.
Example	`--output dist\invoice-viewer.exe`
Important note	If the destination executable already exists, Snug overwrites it.

`--version`	Recommended	
Sets the Windows version number for the application identity.	**Single**	**Both**

Syntax	`--version <VERSION>`
When to use	Use it to give distributed builds a clear version number.
If omitted	Snug uses `0.0.0.0` as the application version.
Example	`--version 1.2.0`
Important note	The version may contain one to four numeric components, each from 0 to 65535.

`--description`	Recommended	
Sets the Windows FileDescription metadata.	**Single**	**Both**

Syntax	`--description <TEXT>`
When to use	Use it to give the executable a meaningful description in Windows properties.
If omitted	Snug uses the application name as the Windows `FileDescription`.
Example	`--description "Invoice Viewer"`
Important note	This affects metadata only; it does not change the application's display name.

`--copyright`	Optional	
Sets the Windows LegalCopyright metadata for the application identity.	**Single**	**Both**

Syntax	`--copyright <TEXT>`
When to use	Use it when you want copyright information embedded in the executable.
If omitted	No custom `LegalCopyright` value is supplied.
Example	`--copyright "Copyright 2026 Example Pty Ltd"`
Important note	This is informational metadata and does not provide legal protection by itself.

`--update-url`	Optional	
Adds a clickable "Check for a newer version" link to launcher error dialogs.	**Single**	**Both**

Syntax	`--update-url <URL>`
When to use	Use it when users should have an easy path to a download or update page.
If omitted	Launcher dialogs do not include an update link.
Example	`--update-url https://example.com/downloads`
Important note	Use a stable HTTPS page where users can find an updated version of your application. Snug does not check whether the page exists or provides a download. This option adds a link to the launcher's error dialog; it does not download or install application updates.

`--icon`	Optional	
Sets the icon embedded in the application executable.	**Single**	**Both**

Syntax	`--icon <PNG/ICO>`
When to use	Use it for a branded application icon.
If omitted	The default Snug application icon is used.
Example	`--icon snug\icon.png`
Important note	This option affects the built application, unlike `snug_preview.exe --icon`, which affects preview dialogs only.

`--manifest`	Optional	
Embeds a Windows application manifest as `RT_MANIFEST`.	**Single**	**Both**

Syntax	`--manifest <XML>`
When to use	Use it only when the application needs a custom Windows manifest.
If omitted	No custom manifest file is embedded.
Example	`--manifest snug\app.manifest`
Important note	Snug checks that the supplied manifest is well-formed XML and rejects malformed XML during the build. This check does not confirm that the manifest contains valid or appropriate Windows application settings. You remain responsible for the manifest's contents. Test the finished executable on the Windows systems you intend to support.

`--main-class`	Optional	
Overrides the Main-Class that Snug would otherwise use in the manifest.	**Single**	**Both**

Syntax	`--main-class <CLASS>`
When to use	Use it when discovery is ambiguous or you intentionally want a different entry point.
If omitted	Snug uses its normal main-class detection behaviour.
Example	`--main-class com.example.Launcher`
Important note	The class must be a valid application entry point available in the packaged JARs. At packaging time, use `--find-main` with the input JAR or directory to inspect candidate entry points.

`--find-main`	Optional	
Lists classes that declare `public static void main(String[])` *and identifies the main class Snug would use when applicable.*	**Single**	**CLI only**

Syntax	`--find-main`
When to use	Use it when you need to inspect or troubleshoot application entry points.
If omitted	Snug proceeds with its normal main-class handling.
Example	`--find-main`
Important note	This is a diagnostic command and does not replace `--main-class` when you intentionally need an override.

`--min-java`	Optional	
Sets the minimum Java major version required by the application.	**Single**	**Both**

Syntax	`--min-java <N>`
When to use	Use it when the application requires a Java version different from the default.
If omitted	Snug will set the minimum Java version to Java `25`.
Example	`--min-java 25`
Important note	Choose the lowest Java major version that the application genuinely requires.

`--jvm-arg`	Optional	
Passes an option directly to the JVM when Snug creates it.	**Multiple**	**Both**

Syntax	`--jvm-arg <OPTION>`
When to use	Use it only when the Java application requires a specific JVM option, such as a memory limit or system property.
If omitted	The JVM starts without any additional options supplied by Snug.
Example	`--jvm-arg=-Xmx2g`
Important note	Invalid or unsupported JVM arguments can prevent the Java runtime from starting.

`--download-jdk`	Optional	
Controls Snug's managed Java-runtime download behaviour.	**Single**	**Both**

Syntax	`--download-jdk <off|auto|force>`
When to use	Use it to control runtime downloads. Normally keep auto. Use force to exercise the managed-runtime path; a compatible Snug-managed cached runtime may still be reused.
If omitted	Snug uses automatic runtime discovery and download behaviour - i.e. the value is set to `auto`.
Example	`--download-jdk=force`
Important note	Using `off` can leave the application unable to start on a machine without a compatible Java runtime.

`--cache-dir`	Optional	
Overrides the application's default per-user cache location.	**Single**	**Both**

Syntax	`--cache-dir <ABSOLUTE_DIR>`
When to use	Only when a deployment has a compelling storage requirement.
If omitted	On Windows, Snug normally stores application JARs under `%LOCALAPPDATA%\snug\<company>\<app>\`.
Example	`--cache-dir "D:\ApplicationCache"`
Important note	Use an absolute path. The directory need not exist at build time, but it must be usable on the target computer. This setting changes the application-JAR cache, not the separate managed-JDK cache.

`--splash`	Optional	
Sets the PNG splash image shown by the launcher before the JVM starts.	**Single**	**Both**

Syntax	`--splash <PNG>`
When to use	Use it when you want a branded launcher splash while Java starts.
If omitted	The launcher starts without the custom splash image.
Example	`--splash snug\splash.png`
Important note	This is the Snug launcher splash, separate from JVM or JavaFX splash mechanisms.

`--splash-ms`	Optional	
Sets the minimum time the Snug splash remains visible.	**Single**	**Both**

Syntax	`--splash-ms <MS>`
When to use	Use it when the splash should remain visible for at least a specific duration.
If omitted	Snug uses the default minimum duration of `1500` ms - i.e. 1.5 seconds.
Example	`--splash-ms 2000`
Important note	This is a minimum duration, not a guarantee that the startup will finish at that time.

`--splash-max`	Optional	
Sets the recommended maximum splash dimensions used for a build-time warning.	**Single**	**Both**

Syntax	`--splash-max <WxH|off>`
When to use	Use it to change or disable the splash-size warning threshold.
If omitted	Snug uses the default `640x360` recommendation.
Example	`--splash-max 800x450`
Important note	This option does not resize the image. If the image exceeds the configured dimensions, Snug displays a build-time warning and continues packaging the application.

`--allow-multiple-instances`	Optional	
Allows more than one copy of the application to run at the same time.	**Single**	**Both**

Syntax	`--allow-multiple-instances`
When to use	Use it only when separate simultaneous copies of the application are genuinely useful.
If omitted	If the application is already running, Snug activates the existing instance instead of starting another copy.
Example	`--allow-multiple-instances`
Important note	For most desktop and business applications, leaving the default single-instance behaviour in place is simpler for users.

`--localisation`	Optional	
Embeds launcher localisation bundles.	**Multiple**	**Both**

Syntax	`--localisation <TXT|DIR>`
When to use	Use it to add translated or customised launcher messages.
If omitted	Only the built-in English launcher messages are available.
Example	`--localisation snug\localisations`
Important note	Files may be supplied individually, or a directory may contain `snug-localisations.<tag>.txt` files.

`--list-tokens`	Optional	
Lists the built-in English message keys and the replacement tokens used by each key.	**Single**	**CLI only**

Syntax	`--list-tokens`
When to use	Use it when editing localisation messages to check which replacement tokens are available for each message key.
If omitted	Normal command processing continues.
Example	`--list-tokens`
Important note	This help command displays information without building the artefact. See **Section 5: Customising Launcher Messages**.

`--init-localisations`	Optional	
Creates starter localisation files and exits without building an application.	**Single**	**CLI only**

Syntax	`--init-localisations [DIR]`
When to use	To create English message files and optional translation templates.
If omitted	No localisation files are generated.
Example	`--init-localisations`
Important note	The built-in English baseline is generated; use `--init-localisations-tag` to add language templates.

`--init-localisations-tag`	Optional	
Adds a language template when generating localisation files.	**Multiple**	**CLI only**

Syntax	`--init-localisations-tag <TAG>`
When to use	With `--init-localisations`, to scaffold additional language files.
If omitted	Only the English baseline is generated.
Example	`--init-localisations --init-localisations-tag de`
Important note	Templates initially contain English values; translating them is a separate step.

`--init-localisations-force`	Optional	
Allows `--init-localisations` to overwrite existing localisation template files.	**Single**	**CLI only**

Syntax	`--init-localisations-force`
When to use	Only when deliberately replacing previously generated translation files.
If omitted	Existing files remain protected from overwrite.
Example	`--init-localisations --init-localisations-force`
Important note	Overwriting files can discard customised translations.

`--init-localisations-stdout`	Optional	
Prints generated localisation templates to standard output.	**Single**	**CLI only**

Syntax	`--init-localisations-stdout`
When to use	For inspection, scripting, or piping generated message templates.
If omitted	Templates are written to the selected directory.
Example	`--init-localisations --init-localisations-stdout`
Important note	This is a scaffolding command and does not package an application.

`--options`	Optional	
Selects an options file explicitly.	**Single**	**CLI only**

Syntax	`--options <PATH>`
When to use	Use it when the configuration file has a different name or location.
If omitted	Snug reads snug.options, then snug.windows.options on Windows, from the current working directory. Missing automatic files are skipped. The directory containing snug.exe is not searched.
Example	`--options config\release.options`
Important note	An explicit `--options` file is used alone. Command-line single-value settings take precedence; repeatable `--jvm-arg` and `--localisation` values accumulate.

`--init-options`	Optional	
Writes Snug's embedded example options file and exits.	**Single**	**CLI only**

Syntax	`--init-options [<PATH>]`
When to use	Use it to create a starter options file. (`snug.options`)
If omitted	No example options file is generated.
Example	`--init-options`
Important note	Snug will not overwrite an existing options file unless you also supply `--init-options-force`.

`--init-options-force`	Optional	
Allows `--init-options` to overwrite an existing destination file.	**Single**	**CLI only**

Syntax	`--init-options-force`
When to use	Use it only with `--init-options` when replacing an existing options file is intentional.
If omitted	Snug protects an existing destination from being overwritten by `--init-options`.
Example	`--init-options --init-options-force`
Important note	This option is deliberately explicit because overwriting an options file can discard local configuration.

`--init-options-stdout`	Optional	
Prints the example options configuration to standard output instead of writing a file.	**Single**	**CLI only**

Syntax	`--init-options-stdout`
When to use	Use it when you want to inspect or redirect the example configuration without creating a file.
If omitted	`--init-options` writes to its normal destination instead.
Example	`--init-options --init-options-stdout`
Important note	This is useful for scripting or comparing the embedded example configuration with an existing file.

`--dry-run`	Optional	
Prepares and validates the payload, then reports what would be built without writing an executable.	**Single**	**CLI only**

Syntax	`--dry-run`
When to use	Use it to check configuration before performing a real build.
If omitted	Snug performs the normal build.
Example	`--dry-run`
Important note	A successful dry run does not prove that the final build or application launch will succeed. Icon loading and manifest XML checks occur later, during resource stamping.

`--emit-payload`	Optional	
Writes the encoded embedded payload to standard output instead of building an executable.	**Single**	**CLI only**

Syntax	`--emit-payload`
When to use	Use it for low-level diagnostics or tooling around Snug's embedded payload.
If omitted	Snug performs the normal build.
Example	`--emit-payload`
Important note	This is an advanced diagnostic option and is not normally needed for application packaging.

`--snug-version`	Optional	
Prints Snug's own version and exits.	**Single**	**CLI only**

Syntax	`--snug-version`
When to use	Use it when checking which Snug build is installed.
If omitted	Snug continues with normal command processing.
Example	`--snug-version`
Important note	This reports the Snug tool version, not the packaged application's `--version` value.

`-h, --help`	Optional	
Displays Snug's command-line help and exits.	**Single**	**CLI only**

Syntax	`--help`
	
	Or
	
	`-h`

When to use	Use it to check the options supported by the installed version of Snug.
If omitted	Snug continues with normal command processing.

Example	`--help`
	
	Or
	
	`-h`

Important note	The built-in help is the definitive reference for the version of `snug.exe` you are running.

Snug Preview (`snug_preview.exe`) Options Reference

`--icon`	Optional	
Changes the icon shown in Snug Preview dialogs.	**Single**	**CLI only**

Syntax	`--icon <PNG/ICO>`
When to use	Use it when you want the preview dialogs to show your application's own icon rather than the Snug Preview icon.
If omitted	Preview dialogs continue to use the built-in Snug Preview icon.
Example	`--icon snug\icon.png`
Important note	This changes preview dialogs only, not a built application. The preview window keeps the Snug mascot and shows your icon beneath its magnifying glass. Supply `--icon` once; if repeated, the last value wins and a warning is printed.

`--localisation`	Optional	
Loads localisation messages so Snug Preview can display launcher dialogs in additional languages.	**Multiple**	**CLI only**

Syntax	`--localisation <FILE|DIR>`
When to use	Use it when checking customised or translated launcher messages before building the application.
If omitted	Only the built-in English messages are available in the preview language selector.
Example	`--localisation snug\localisations`
Important note	Repeat the option for additional files or directories. Directories are scanned at the top level only. English remains available; supplying `snug-localisations.en.txt` replaces the built-in English baseline.

`--help / -h`	Optional	
Displays the built-in help for `snug_preview.exe`.	**Single**	**CLI only**

Syntax	`--help`
	
	Or
	
	`-h`

When to use	Use it when you need to check the options supported by the installed version of Snug Preview.
If omitted	Snug Preview starts normally instead of displaying help.

Example	`--help`
	
	Or
	
	`-h`

Important note	Use `snug_preview.exe --help` to check the options supported by your installed preview version.

Building Snug from Source

This chapter is for developers and technically experienced readers who want to compile Snug itself. You do not need to build Snug from source to package a Java application, use `snug.options`, or follow the earlier chapters. Most readers can safely skip this chapter.

Snug is an open-source Rust project. Its source code is available at

> https://github.com/synapticloop/snug

These instructions describe a conventional starting point for working with the repository; the README and build files for the particular release remain the authoritative source if commands, dependencies, or outputs change.

Why Build Snug Yourself?

Building from source can be useful when checking how a release was produced, investigating a problem, testing a change, or contributing to the project. It also means that the project can be maintained independently of the precompiled executables. If you simply want to use Snug, download a published release instead.

Before You Begin

For the Windows x64 release described in this book, use a Windows x64 development environment with Git, rustup, and the Microsoft C++ build tools and Windows SDK needed for MSVC linking. The current repository pins Rust 1.98.1, including Clippy and rustfmt, in rust-toolchain.toml; Cargo.toml declares a minimum of Rust 1.89. Follow the files in your chosen release checkout if these versions change.

Open PowerShell or another terminal and confirm that the tools are available:

```
git --version
rustc --version
cargo --version
```

The version numbers will vary. If a command is not recognised, install or configure that tool before proceeding. A Java Development Kit is not necessarily required to compile the Rust launcher itself; build or test fixtures involving Java may have their own requirements.

Obtaining the Source Code

To obtain the current development source, clone the repository and enter its directory:

Command(s)

```
git clone https://github.com/synapticloop/snug.git
cd snug
```

For a reproducible build associated with a published release, use a tag rather than assuming that the main branch contains the same code. List the available tags, then check out the tag corresponding to your chosen release:

Command(s)

```
git tag --list
git checkout <release-tag>
```

Replace `<release-tag>` with an actual tag from the repository. A checkout of a release tag is normally in a detached HEAD state; that is expected if you are building rather than making changes. Record the tag and commit identifier when reporting a build problem.

Understanding the Project Layout

At a high level, the project contains Rust source code, Cargo manifests, resources used by its executables, and project-specific build or packaging assets. Start with the root README and `Cargo.toml`, then inspect the source directories and any workspace members. The exact directory layout may change between releases, so the repository itself is more reliable than a fixed directory tree printed in this book.

Snug's build-time executable and the executable it generates for a Java application serve different purposes. Building the Snug project produces the tooling; running that tooling with an application's JAR and configuration produces the application-specific output described in the earlier chapters.

Building the Complete Windows Release

For a complete Windows distribution, run the repository's build script from the repository root, not from the scripts directory. The script expects Cargo.toml and the demonstration application's configuration files in the current working directory. For a checked release build, use `release.cmd` instead: it runs the project's checks before the same distribution build.

Command(s)

```
.\scripts\windows\build.cmd
```

The script performs the distribution pipeline in the required order. It builds the Windows launcher, refreshes `bin\launcher-stub-windows-x86_64.exe`, then rebuilds `snug.exe` with that launcher embedded. The sequence matters: rebuilding the builder without first refreshing the stub can leave it containing outdated launcher code. To run checks as well, use

Command(s)

```
.\scripts\windows\release.cmd
```

which invokes `check.ps1` before `build.cmd`. The wrapper also provides `--CheckOnly` and `--SkipCheck`; these are mutually exclusive. For `build.cmd` development flags, invoke `build.cmd` directly, since `release.cmd` forwards other arguments to both scripts.

Next, the script packages the JavaFX demonstration JAR, stamps the appropriate icons into `snug.exe`, `snug_preview.exe`, and `Snug Quick Build.exe`, and builds the associated tools. It also checks that the correct launcher bytes were embedded and that each executable contains the expected icon artwork. These checks protect against subtle problems that an ordinary Cargo compilation would not detect.

Finally, the script stages the distribution files in `release\windows-x86_64\`. The standard release folder contains:

- `snug.exe`
- `Snug Quick Build.exe`
- `snug_preview.exe`
- `snug-javafx-demo-windows.jar`

The demonstration executable is also created in the assets directory as `assets\snug-javafx-demo-windows.exe`. Keep `Snug Quick Build.exe` in the same directory as `snug.exe` when distributing it: the Quick Build tool looks for `snug.exe` alongside itself.

The script expects the demonstration JAR at `assets\snug-javafx-demo-windows.jar`. If that file is absent, the normal packaging step fails; build or obtain the demonstration JAR first, or explicitly use `--SkipPackage` while working on the builder. The script also supports focused flags such as `--SkipLauncherRebuild`, `--SkipDevTools`, `--SkipDropper`, `--SkipVerify`, `--SkipRelease`, and `--Clean`. These are useful when developing individual parts of Snug, but a full release should be built with the normal pipeline and its verification enabled.

The release directory is updated in place, so files from an earlier run can remain after a partial build. Missing staging inputs are reported as skipped. `--Clean` cleans snug-launcher, snug-cli, and snug-dropper Cargo outputs; it does not empty the release directory. Use a fresh staging folder before treating the output as a clean distribution.

The script targets Windows x86_64. It also parses `--CrossCompile` for a GNU-linked build through `cargo-zigbuild`, but the current script explicitly requires a Windows toolchain for resource compilation. Treat this as a development path, not a supported route to build Windows release artefacts on another operating system.

Building Individual Components with Cargo

Building a Rust component directly with Cargo is useful for development and diagnosis, but it is not a substitute for the complete Windows distribution pipeline. From the directory containing the relevant `Cargo.toml`, a standard Rust release build is:

Command(s)
`cargo build --release`

If the repository uses a Cargo workspace, the command above builds the default members. Consult Cargo.toml or the repository's build instructions to determine whether you need `--workspace`, `-p <package-name>`, a specific feature, or an additional packaging command to produce every distributable executable. Do not assume that a single Cargo invocation creates the same collection of files as an official release.

For a faster development build while making changes, use:

Command(s)
`cargo build`

Cargo normally places development build outputs under `target/debug` and release outputs under `target/release`, unless the project configures a different target directory. Check the build output and release scripts for the precise executable names and the files that must accompany them.

Running Tests and Checks

To run the project's defined Windows checks, use `.\scripts\windows\check.ps1` from the repository root. This checks formatting, runs Clippy, and executes the workspace tests. For individual development checks, you can also run standard Cargo commands:

Command(s)
`cargo test` `cargo check`

For a workspace, consult the project configuration to see whether all members are included. Some tests may depend on Windows features, external downloads, local Java installations, or sample projects. Read any failure message before treating it as a defect in the application itself. A passing unit test suite does not replace testing a generated Windows executable on a representative machine.

Checking the Result

Locate the compiled Snug tool, run its `--help` option and compare the available options with the release you intended to build. Next, package a small known-working Java application using the workflow from this book. Verify that the generated executable starts, handles its runtime requirements, and behaves as expected. Keep this smoke test separate from the build process: a successful compilation only confirms that the compiler produced output.

Common Build Problems

If Cargo is not recognised, check that Rust was installed and that the terminal has been restarted so the toolchain is on PATH. If linking fails on Windows, confirm that the MSVC target and required Visual Studio C++ tools are installed. If dependency downloads fail, check connectivity and any proxy settings. If an executable builds but will not launch, distinguish between a failure in Snug's own build and a failure in the Java application or its runtime.

When asking for assistance, include the checked-out tag or commit, the Windows version and architecture, the Rust version, the command run, and the complete error message. Avoid making unrelated toolchain upgrades while trying to reproduce a release build.

Making Changes and Contributing

Develop changes on a branch rather than directly on a checked-out release tag. Keep individual changes focused, run the relevant tests, and follow the contribution guidance in the repository before opening a pull request. Issues, review discussions, and internal implementation details belong primarily with the source code, where they can stay in step with changes to the project.

A Note About Versions

This chapter deliberately describes the stable outline of the process rather than promising that all command-line flags, package names, or build artefacts will remain unchanged. For a particular Snug release, its Git tag, Cargo manifests, README, and release instructions take precedence. The book explains how to approach building Snug; the repository documents the exact mechanics of the current implementation.

Definitions

#

42: *The meaning of life*

A

Application: *A computer program designed to perform a particular task. Applications may have a graphical interface with windows and buttons, or they may run from a command line. In this book, the Java program being packaged by Snug is generally referred to as the application.*

Application Store: *An online service used to publish, distribute, and sometimes sell applications. Examples include the Microsoft Store and Apple App Store. Snug does not require an application store; the executable it creates can be distributed directly to users.*

Argument: *Information supplied to a program when it starts. With the standard builder defaults, arguments passed to the finished Snug executable are forwarded to the Java application. Builder* `--jvm-arg` *values instead configure the JVM itself.*

The Snug builder accepts arguments either from the command line or from an options file such as `snug.options`*.*

Artefact: *A file produced as the result of a build or packaging process. In this book, a Snug artefact normally means the Windows executable created by Snug from your Java application.*

B

Binary: *A general term for a compiled program that a computer can run. Unlike source code, which is written for people to read and edit, a binary has been converted into a form intended for the computer to execute.* `snug.exe` *is the Snug binary used in this book.*

Build: *The process of turning source files or other application components into something that can be run or distributed. With Snug, building generally refers to creating a Windows executable from one or more Java JAR files.*

Builder: *The part of a software tool responsible for creating the finished application or package. The Snug builder is* `snug.exe`*, which takes your Java JAR files and configuration and produces a Windows executable.*

C

CLI (Command-Line Interface): *A way of interacting with a program by typing commands rather than clicking buttons and menus. Snug is primarily a command-line utility.*

Code Signing: *A process that digitally signs an application so that an operating system can verify who published it and whether the file has been changed since it was signed. Commercial code-signing certificates can involve additional cost and administration.*

Command: An instruction typed into a terminal, Command Prompt, or PowerShell window *that tells the computer to run a program or perform an action.*

Command Line: *A text-based way of interacting with a computer by typing commands. For example, Snug can be run by typing* `snug.exe` *followed by its options and the JAR file that should be packaged.*

Command-Line Option: *An additional instruction supplied when starting a command-line program. Options usually change how the program behaves. In Snug, options are used to set things such as the application name, company name, and other packaging settings.*

Command Prompt: *A command-line program supplied with Microsoft Windows. It allows commands and applications to be run by typing their names rather than opening them through the graphical desktop.*

Compiled: *Converted from programming code into a form that can be run by a computer or another software runtime. A compiled Java application is commonly distributed as one or more JAR files.*

Configuration: *A collection of settings that control how a program behaves. Snug configuration determines how an application is packaged, branded, and launched.*

Console: *A text-based area where a program can display messages and where commands may be entered. When Snug is run from a command line, its progress and results are displayed as console output.*

D

Deployment: *The process of getting an application from the person who created it onto the computers of the people who will use it. Snug is intended to make deployment easier by turning a Java application into a familiar Windows executable.*

Desktop Application: *An application that runs directly on a user's computer rather than primarily through a web browser. Snug is designed to package Java applications so that they behave more like conventional Windows desktop applications.*

Directory: *A location on a computer used to organise files and other directories. It is commonly called a **folder** in graphical programs such as Windows File Explorer.*

Distribution: *The process of giving an application to the people who need to use it. Distribution might involve copying a file, placing it on a shared drive, making it available for download, or using another process already used by a business.*

Double-Click: *Pressing the primary mouse button twice in quick succession. In Windows this is the familiar way of opening files and starting applications. One of Snug's goals is to allow a Java application to be started simply by double-clicking its executable.*

E

`.exe`***:*** *The file extension commonly used for executable applications on Microsoft Windows. A file such as `my-app.exe` can normally be started by double-clicking it or running it from a command line.*

Executable: *A file containing a program that the operating system can run. Snug packages a Java application into a Windows executable so that the user can launch it like a conventional Windows application.*

F

Fat JAR: *A Java JAR file that contains the application's own code together with most or all of the libraries it depends upon. This makes the application easier to distribute because its required Java libraries are collected into a single JAR file. Sometimes called an **uber JAR** or, depending on the build tool being used, a **shadow JAR**.*

File Extension: *The letters appearing after the final dot in a filename that commonly indicate what type of file it is. For example, `.jar` identifies a Java archive and `.exe` identifies a Windows executable.*

G

Git: *A version-control system that records changes to files in a project, making it possible to review or return to earlier versions.*

GitHub: *A platform built around Git for hosting source code, tracking issues, and collaborating on software projects. Snug's source repository is hosted on GitHub; consult its Releases page for any published downloads.*

Graphical User Interface (GUI): *The visual part of an application that people interact with using windows, buttons, menus, icons, and other on-screen controls. This is different from a command-line application, which is primarily controlled by typing commands.*

For JavaFX applications, the graphical interface is provided by the JavaFX library.

I

Icon: A small image used to visually represent an application, file, or action. *Snug can place your application's own icon into the Windows executable it creates.*

Installer: *A program that places an application and its supporting files onto a computer, and may also create shortcuts, Start menu entries, and uninstall information. Snug creates an executable and manages its Java launch requirements, but it is not intended to be a complete installer framework.*

Internationalisation: *Designing software so that its text and other user-facing elements can be adapted to different languages and regions. Snug allows its launcher messages to be customised for different languages.*

J

Java: *A programming language and software platform used to create applications that can run on many different operating systems. Snug is specifically designed to package Java applications for easier use on Windows.*

Java Application: *An application written for the Java platform. Java applications are commonly distributed as JAR files and require a suitable Java runtime in order to run.*

JavaFX: *A framework used to create graphical desktop applications with Java. The demonstration application used in this book is a JavaFX application.*

JAR: *Short for **Java Archive**. A* `.jar` *file is a package that can contain compiled Java program code, resources, and other files used by a Java application. Snug packages one or more JAR files into the Windows executable it creates.*

Java Runtime: *The software required on a computer to run Java applications. The user does not normally interact with the runtime directly. If a compatible Java runtime is not available, Snug can offer to download and prepare one, depending on the application's runtime-download setting.*

JRE (Java Runtime Environment): *A name for the software needed to run Java applications. In this book, the more general term Java runtime is used because Snug may use an installed or Snug-managed Java runtime.*

L

Launcher: *A small program whose job is to start another application and prepare anything that application needs first. The Snug launcher is built into the executable produced by Snug. It checks the Java requirements, prepares the application, and then starts it.*

Local Application: *An application that runs on the user's own computer rather than being hosted on a remote web server. Snug is particularly suited to local applications such as internal business tools.*

M

Metadata: *Information stored about a file or application rather than the application's main content. Application metadata can include details such as its name, company, version, and description. Snug can add this information to the executable it creates.*

Modal Dialog Box: *A window that requires a response before the user can continue with the current task. Snug uses dialogs when it needs the user to act, for example, to install a required Java runtime or respond to a launch error.*

N

Native Executable: *An executable designed to be recognised and launched directly by a particular operating system. Snug creates a native Windows* `.exe` *launcher around a Java application, giving users a familiar Windows application to start.*

O

Operating System: *The software that manages a computer's hardware and provides the environment in which applications run. The Windows x64 release described in this book creates applications for that platform; the Java runtime and any native libraries must also be compatible with it.*

Option: *A setting supplied to a program that changes how it behaves. Snug options control things such as application naming, branding, Java requirements, and launch behaviour.*

P

Package / Packaging: *The process of collecting an application and the information it needs into a form that is easier to distribute and run. Snug embeds the Java JAR files and the settings needed by its launcher into a Windows executable. It does not include the* `snug.options` *text file itself.*

Parameter: *A value supplied to an option or program. In this book, an option is a named setting such as* `--name`*, and its parameter is the value that follows it, such as* `"My Application"`*. Argument is the broader term for information passed to a program.*

Payload: *The application data carried inside another file or package. In a Snug executable, the Java application's JAR files and related information form part of the data carried by the executable.*

PowerShell: *A command-line and scripting environment included with modern versions of Microsoft Windows. Snug commands can be run from PowerShell as well as from the traditional Windows Command Prompt.*

Preview: *A way of seeing how something will look before producing the finished version. The optional* `snug_preview.exe` *utility allows Snug's dialog boxes to be viewed while they are being customised.*

Publishing Platform: *A service that hosts software and provides a way for users to discover or download it. Snug does not publish or host applications itself; it creates the executable and leaves you in control of how that file is distributed.*

R

Repository: *A project and its history as managed by Git. A repository normally contains the files that make up the project together with a record of how those files have changed over time.*

Release: *A particular version of software made available for use. A release tag identifies source code; a downloadable release may also include compiled executables and other assets.*

Runtime: *Software that provides the environment another program needs while it is running. Java applications require a Java runtime. Snug can locate or download an appropriate runtime before starting the packaged application.*

Runtime Components: *The files that make up the runtime environment needed by an application. Snug can download and reuse Java runtime components rather than requiring every packaged application to contain its own complete copy.*

S

Shadow JAR: *A JAR file produced using the Gradle Shadow plugin, commonly combining an application's code and its libraries into a single file. It is a type of fat JAR. See also: Fat JAR.*

Shared Drive: *A storage location that can be accessed by multiple people, usually through a business or home network. An internally distributed Snug application could, for example, be placed on a shared drive for colleagues to copy or run.*

Source Code: *The human-readable instructions written by a programmer to create software. Source code is normally compiled or otherwise processed before the finished application is distributed to users.*

Splash Screen: *An image or small window displayed while an application is starting. It commonly shows an application's name, logo, or branding. Snug allows a customised splash screen to be displayed while the Java application is being launched.*

Startup Behaviour: *The actions a program performs when it is first launched. In Snug this can include checking for a suitable Java runtime, displaying messages or a splash screen, and then starting the Java application.*

Synapticloop: *The software project and publisher behind Snug.*

T

Terminal: *An application that hosts a text-based command session. Command Prompt and PowerShell are command environments that can run in a terminal.*

U

Uninstallation: *The process of removing an installed application and its associated files, shortcuts, and configuration from a computer. Snug is not a complete installer or uninstaller system.*

URL (Uniform Resource Locator): *The address used to identify a resource on a network or the internet, such as a web page or downloadable file. For example, the Snug documentation and releases are accessed using URLs.*

V

Version: *A number or other identifier used to distinguish one release of an application from another. Versions make it possible to tell whether one copy of a program is newer or older than another.*

W

Web Application: *An application that is primarily accessed through a web browser and usually runs partly or entirely on a remote server. Snug is aimed at applications that run locally on a user's computer rather than applications that need to be hosted as web applications.*

Windows: *Microsoft's desktop operating system. The version of Snug described in this book packages Java applications as Windows executables (`.exe` files).*

Windows Executable: *An executable file designed to run on Microsoft Windows, normally identified by the `.exe` file extension. Producing a familiar double-clickable Windows executable from a Java application is one of Snug's main purposes.*

Copyright, Licensing, and Third-Party Notices

Copyright Notices

Snug is a product of Synapticloop.

• Oracle® and Java® are registered trademarks of Oracle and/or its affiliates.

• Microsoft® and Windows® are trademarks of the Microsoft group of companies.

• Rust™ and Cargo™ are trademarks of the Rust Foundation.

• GitHub® is a registered trademark of GitHub, Inc.

All other trademarks, product names, company names, logos, and brands mentioned in this publication are the property of their respective owners.

References to third-party products, technologies, and trademarks are provided for identification and information purposes only. This publication is not affiliated with, sponsored by, endorsed by, or otherwise associated with Oracle, Microsoft, the Rust Foundation, or any other third-party trademark owner mentioned herein.

The information in this publication is provided for informational purposes and is believed to be accurate at the time of publication. Software, operating systems, APIs, command-line tools, and third-party services may change over time. Readers should consult the relevant project or vendor documentation for current information.

The author and publisher make no warranties regarding the completeness or accuracy of this publication and, to the extent permitted by applicable law, disclaim liability for loss or damage arising from its use.

Copyright and Licensing

This book is protected by copyright. You are welcome to use the ideas, techniques, commands, and examples in this book when creating your own applications.

Unless otherwise stated or permitted by applicable law, reproducing, redistributing, publishing, or selling substantial portions of this book requires permission.

Snug

Snug is software separate from this book. Its current Cargo.toml declares MIT OR Apache-2.0. Consult the licence texts and notices supplied with the particular source checkout or release for the terms governing use, copying, modification, and distribution.

Packaging an application with Snug does not replace the licences that apply to your application or its components. Check the terms for Snug's launcher code and any libraries, assets, runtimes, or other software included in your distribution.

Third-Party Software

Snug and the applications you create with it may use third-party software, libraries, runtimes, or other components.

Third-party components retain their own licences and terms. Before distribution, identify the components you include and retain any notices, attributions, or licence texts required by their applicable terms.

© 2026 Synapticloop. All rights reserved.

About Snug

Why Is It Called Snug?

The word 'snug' describes something comfortable, compact, and neatly fitted. That feels appropriate for what Snug is trying to do.

A Java application can involve quite a few moving parts: JAR files, Java runtimes, launchers, icons, metadata, splash screens, and configuration. Snug packages the application and launcher settings into a single Windows executable; the Java runtime is provided separately.

The name also reflects the experience Snug is intended to create: the person using the application should not need to think about Java, runtimes, classpaths, or how the application was packaged. They should be able to double-click the application and get on with their work.

In that sense, Snug wraps the technical details around the application so that everything fits neatly together.

Compact, neatly packaged, and comfortable to use.



Say Hello to Snug

Afterword: On Documentation

My view on documentation is that everyone appreciates having it, but few people enjoy writing it. Documentation comes in many forms: source code and tests, project manuals, command-line help, blog posts, videos, books, and increasingly, information assembled and presented by AI. Whatever form it takes, somebody has to explain how things work.

The interesting thing about writing documentation is that it forces you to look at what you have created from somebody else's perspective.

I am a firm believer in ease of understanding and implementation over architectural purity. Something may be technically elegant, but if it is unnecessarily difficult to use or explain, perhaps it isn't quite as elegant as you first thought.

Writing documentation can be a grind. There is the book or user guide, the application's interface, the command-line help, the repository's Markdown files, and all the other places where things need to be explained. Every new feature can mean revisiting several of them, and every change risks making something else out of date. It is not difficult to see why documentation often falls towards the bottom of the priority list.

However, writing this book has taught me something.

Having to explain how Snug works, particularly to someone who may never have packaged a Java application before, has made me reconsider parts of its design. Explaining the command-line options, the runtime installation process, and the messages presented to users has prompted changes to both the documentation and the software itself.

When you have to explain a decision, you get another opportunity to question it. You start asking yourself:

- Why did I choose to do things this way?
- Could it be simpler?
- Would somebody else understand how to use or configure it?
- Is there something missing that would make the experience easier?

Documentation also makes it harder to overlook gaps in functionality. Sometimes a feature is missing for a perfectly good reason. Sometimes it was never considered. And sometimes you discover that something you thought was obvious is not obvious at all.

That does not mean every possible feature should be added. Quite the opposite. Being able to explain why something has deliberately been left out is just as important as explaining what has been included.

Throughout the writing of this book, Snug's code, features, and behaviour have changed as a direct result of trying to explain them. In several cases, what began as an attempt to improve a paragraph ended with changes to the application itself.

Naturally, those changes meant rewriting the paragraph. Again.

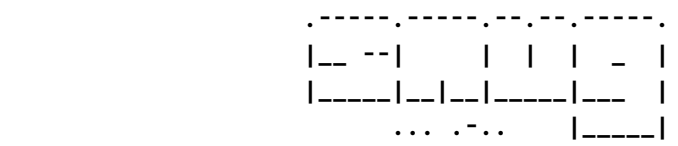
I believe Snug is a better product because of this process.

So, if there is one thing I would recommend, it is to write about whatever you are building. It does not need to be a book. A simple HOW-TO, written for somebody who has never seen your project before, is a good place to start.

You may find that the hardest parts to explain are the very parts worth reconsidering.

And that, perhaps, is one of the best reasons to write documentation in the first place.

End Plate



~ ~ ~ * ~ ~ ~

"Parting is such sweet sorrow"

William Shakespeare
Romeo and Juliet
Juliet, Act 2, scene 2