

Automating Build and Deployment processes with Build4D Component

By Soukaina BACHIKH, Customer Success Engineer, 4D Inc.

Technical Note 25-07

Table of Contents

Abstract.....	3
Introduction	3
What is Build4D?	3
Build4D Configuration	4
Build Scenarios.....	5
Automated Workflows with CI/CD and GitHub Actions.....	12
Integrating Build4D into CI/CD.....	14
Demo: Automating a 4D Build Pipeline	14
Conclusion	26

Abstract

As 4D evolves to support modern software development practices, the need for build automation and reproducible deployment pipelines becomes essential. Introduced as a 4D component by the 4D team, **Build4D** helps developers automate the process of compiling, building, and packaging 4D applications. It plays a crucial role in integrating DevOps principles into 4D workflows by allowing scripted control over creation of standalone applications, components, and client/server binaries. This technical note provides an overview of Build4D, explains how it works, and presents practical examples of building various types of 4D projects.

Introduction

In today's world of software development, things move fast. Teams are expected to deliver updates quickly, fix bugs efficiently, and maintain system stability—all without sacrificing quality. To keep up, automation is no longer just a convenience; it ensures consistent, repeatable outcomes regardless of who runs the build or where it's executed. That's where **Build4D** comes in.

Build4D is a 4D component designed to automate the process of building, signing, and packaging 4D applications. Rather than manually navigating menus and performing repetitive tasks for each build, Build4D enables fully scripted operations. This allows builds to be triggered by a single command or automatically integrated into Continuous Integration (CI) pipelines.

This technical note offers guidance on how to get started with Build4D, including:

- An overview of Build4D's features and its integration into development workflows
- Instructions for writing and customizing Build4D scripts to fit project requirements
- Examples demonstrating Build4D's usage in automated environments

Whether working alone or in teams, Build4D helps align 4D development with modern software engineering practices. Let's explore how to implement it effectively.

What is Build4D?

Component Overview:

Build4D is a 4D component providing developers with a straightforward way to automate the building and packaging of their applications through code. Rather than manually using the 4D interface for each build, developers use scripts to generate compiled files, standalone applications, client/server setups, or components consistently and repeatably. This is especially valuable in fast-paced environments like CI/CD pipelines or larger teams, where Build4D streamlines workflows and reduces manual errors.

Supported 4D Versions and Compatibility

Build4D supports project-based 4D applications, a format introduced in 4D v18. The Build4D component itself has been available since 4D v19 R3 and has improved with each release. To avoid compatibility issues and leverage full functionality, it's recommended to use the version of Build4D that matches 4D version.

To ensure compatibility:

1. **Check the current 4D version**
Find the version in the "About 4D" section of the 4D application.
2. **Download the corresponding Build4D version**
Head to the official Build4D GitHub repository (<https://github.com/4d-depot/Build4D>) and select the branch that matches the used 4D version (e.g., 4D20.x for 4D v20).

Compatibility between Build4D and the 4D environment is essential for maintaining a reliable and efficient development process.

How Build4D Integrates with 4D

Build4D is installed as a component in the 4D project:

- Add the `Build4D.4dbase` folder to the `Components` directory of the project, or clone it from the official GitHub repository, (refer to this [tech tip](#) for details on adding components).
- Use either the interpreted or compiled version of the component.
- Access Build4D functionality through classes such as `cs.Build4D.Standalone`, `cs.Build4D.Server`, and `cs.Build4D.CompiledProject`.

Note: When using an interpreted component:

- copy the "Build4D" folder in the project's "Components" folder,
- add the ".4dbase" extension to the "Build4D" folder.

Build4D Configuration

Once Build4D is integrated into the project, the next step is to configure it to meet the specific build needs. This configuration process involves defining the build parameters using specialized classes provided by the component.

Discovering Build4D Classes

Build4D offers several classes in the `cs.Build4D ClassStore`, each suited to different build scenario:

- **cs.Build4D.CompiledProject:** For creating .4dz compiled structure files.
- **cs.Build4D.Component:** For building reusable 4D components.
- **cs.Build4D.Standalone:** For building standalone applications.
- **cs.Build4D.Server:** For building the server part of a client/server app.
- **cs.Build4D.Client:** For building the client part of a client/server app.

All the classes extend the “**_core**” class and provide a build() method to trigger the build process after configuration.

Key parameters and their usage

Parameter	Description
projectFile	Path to the 4D project file.
sourceAppFolder	<ul style="list-style-type: none">- Path to the 4D Volume Desktop folder (Client)- Path to the 4D Server folder (Server)
destinationFolder	Output directory where the final build files will be placed.
buildName	Name of the resulting application or structure.
IPAddress	Add the Server IP Address to the Client build settings (in order to connect the Client with the right Server)
versioning	Version number to embed in the build metadata.
appShortVersion	Human-readable version number (e.g., 1.0.0).
iconPath	Path to a custom application icon.

These parameters are part of the object passed to the class’s build() method. Additional parameters can be found in the official documentation.

Build Scenarios

In this section, practical examples illustrate how Build4D can be used to generate different types of builds. These scenarios demonstrate how to use the component’s classes and parameters to automate builds tailored to the project’s structure.

The following scenarios correspond to the different build types available in 4D’s graphical **Build Application** interface:

The screenshot shows the 'Build Application' window. It has a title bar 'Build Application'. Inside, there are two main configuration fields: 'Application Name:' with a text input containing 'demo-Build4D-Win', and 'Destination Folder:' with a text input containing '"demo-Build4D-Win_Build" in volume "C:"' and a browse button '...'. At the bottom, there is a horizontal tabbed menu with five tabs: 'Compiled structure' (selected), 'Application', 'Client/Server', 'Plugins & components', and 'Licenses & Certificate'.

1. Build a compiled structure

A compiled structure (.4dz) is the foundation for production-ready applications. It offers better performance, protects source code, and is required for deployment in standalone or client/server formats. Build4D makes it easy to automate the compilation process.

Below is a commented example showing how to create a compiled structure (.4dz) using the `cs.Build4D.CompiledProject` class.

```
// Declare variables for the build object, settings, and result
var $build : cs.Build4D.CompiledProject
var $settings : Object
var $success : Boolean

// Create and initialize the settings object
$settings := New object()

// Define the output folder where the compiled structure (.4dz) will be saved
$settings.destinationFolder := Folder(fk desktop folder).folder("builtStructure/")

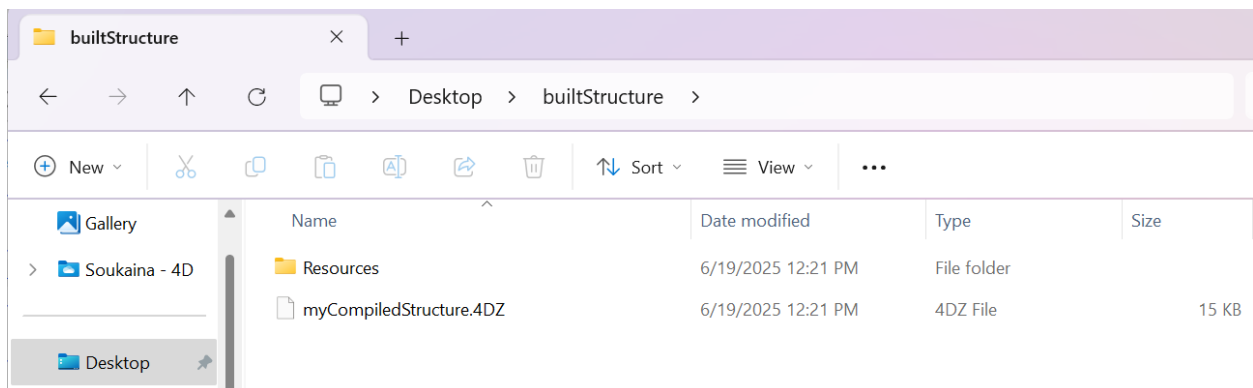
// Define the name of the compiled structure
$settings.buildName := "myCompiledStructure"

// Define additional files or folders to include in the build (e.g., resources)
$settings.includePaths := New collection
$settings.includePaths.push(New object("source"; "Resources/"))

// Instantiate the build object with the settings
$build := cs.Build4D.CompiledProject.new($settings)

// Launch the build process and store the result (True if successful)
$success := $build.build()
```

The result is a .4dz file, ready for packaging into other deployment formats.



2. Build a component

A 4D component (.4dbase) is a reusable module that encapsulates logic, forms, and methods to be shared across multiple projects. It helps promote modular development and code reuse. With Build4D, components can be generated automatically, including only the necessary resources and excluding development files.

Below is a commented example showing how to build a reusable 4D component (.4dbase) using the `cs.Build4D.Component` class.

```
// Declare variables for the build object, settings, and success flag
var $build : cs.Build4D.Component
var $settings : Object
var $success : Boolean

// Create and initialize the settings object
$settings := New object()
// Define the output folder where the component will be generated
$settings.destinationFolder := "./Test/"

// Set the name of the resulting component
$settings.buildName := "myComponent"

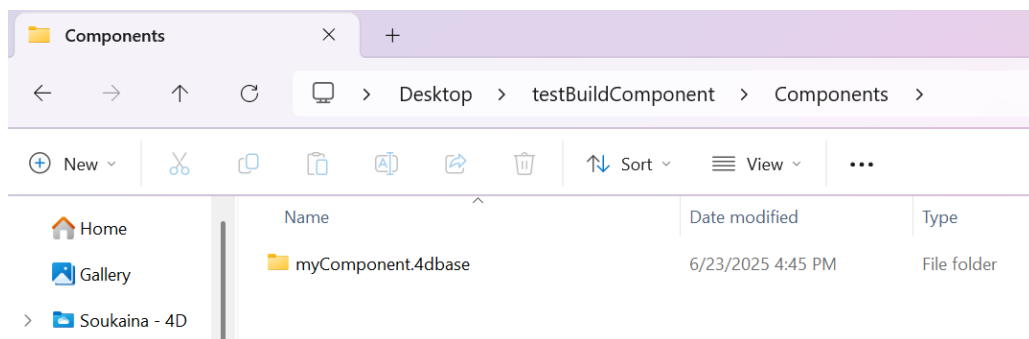
// Define which folders or files to include in the component package
$settings.includePaths := New collection
$settings.includePaths.push(New object("source"; "./Documentation/"; "destination"; ""))

// Define which folders or files to remove before packaging the component
$settings.deletePaths := New collection
$settings.deletePaths.push("./Resources/Dev/") // Exclude development resources

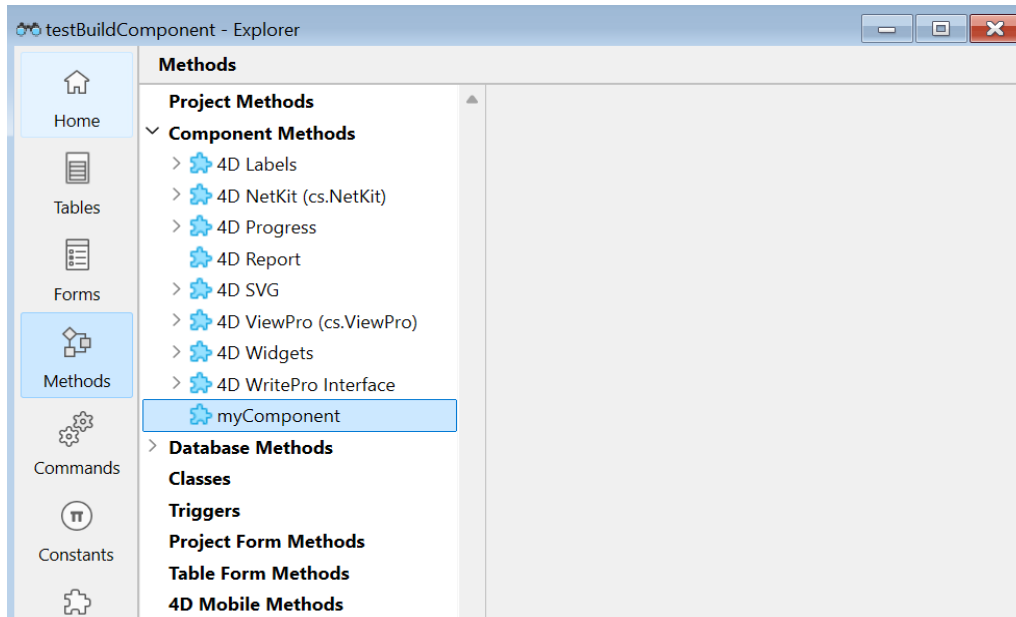
// Instantiate the build object with the configured settings
$build := cs.Build4D.Component.new($settings)

// Execute the build process and store the result (True if successful)
$success := $build.build()
```

After executing the method, the .4dbase project is now ready to be added to a host project.



Once complete, the “.4dbase” component can be copied into the Components folder of any other 4D project.



3. Build a standalone application

A standalone application is a self-contained executable that includes both the structure and the data file. It is designed for single-user use or desktop distribution without needing a separate server. Build4D allows developers to create standalone applications with advanced options such as obfuscation, embedded licenses, and UI preferences—all through scripting.

```
// Declare variables for the build object, settings, and success flag
var $build : cs.Build4D.Standalone
var $settings : Object
var $success : Boolean

// Initialize the settings object
$settings := {}

// Define the path to the project file to build
$settings.projectFile := Folder(fk desktop folder).file("demo-Build4D-Win/Project/demo-Build4D-Win.4DProject")

// Set the name of the resulting application
$settings.buildName := "myApp"

// Define the output directory where the standalone app will be generated
$settings.destinationFolder := Folder(fk desktop folder).folder("standaloneApp")

// Configuration options
$settings.obfuscated := True           // Obfuscate the source code (protects intellectual property)
$settings.packedProject := False       // Choose whether to pack project files into a single bundle
$settings.useSDI := False              // Use Single Document Interface (for Windows apps)
$settings.startElevated := True        // Run the installer with admin rights if needed
```



```

$settings.lastDataPathLookup := "ByAppPath" // Determines how the app finds its last used data file

// Define the path to the source 4D application
$settings.sourceAppFolder := Folder(fk documents folder).folder("4D 20 R8 100333/4D Volume Desktop")

// Exclude development folders or files from the final build
$settings.deletePaths := New collection("Resources/Dev/")

// Application metadata
$settings.versioning := New object()
$settings.versioning.version := "1.0.0"
$settings.versioning.copyright := "copyright"
$settings.versioning.companyName := "4D"

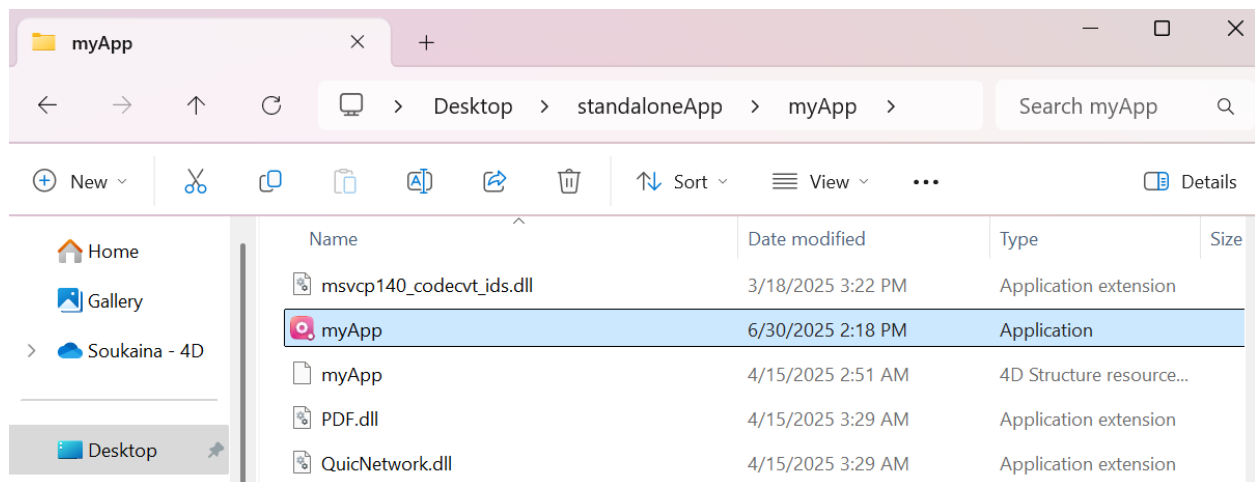
// Specify the path to the deployment license file
$settings.license := Folder(fk licenses folder).file("4USDxxxxxx.license4D")

// Instantiate the build object with the settings
$build := cs.Build4D.Standalone.new($settings)

// Launch the build process and store the result (True if successful)
$success := $build.build()

```

The result is a self-contained application ready to be distributed or installed on user machines.



4. Build a Client/Server application

A **client/server application** allows multiple users to access a centralized 4D database, and business logic hosted on a server. The server manages the data, while each client provides a user interface. Build4D makes it easy to generate both the server and client sides of the application in a consistent and automated way.

The process is divided into two parts:

- First, build the **client**, which includes an archive that will be embedded into the server.

- Then, build the **server**, which will integrate the client archive for deployment.

1. Client Build Method:

Responsible for defining the build configuration of the client side. Execute the code below to retrieve the client.exe and the archive file that will be integrated into the server side.

```
#DECLARE()->$result : Object

// Declare variables
var $build : cs.Build4D.Client
var $settings; $archive : Object
var $success : Boolean

// Initialize settings
$settings := {}

// Set elevated rights if required
$settings.startElevated := True

// Define the path to the 4D project
$settings.projectFile := Folder(fk desktop folder).file("demo-Build4D-Win/Project/demo-Build4D-Win.4DProject")

// Define the path to the source 4D Volume Desktop app
$settings.sourceAppFolder := Folder(fk documents folder).folder("4D 20 R8 100333/4D Volume Desktop")

// Define the output folder and application name
$settings.buildName := "clientApp"
$settings.publishName := "myAppCli"
$settings.destinationFolder := Folder(fk desktop folder).folder("buildApp/Client")

// Define the IP address of the server
$settings.IPAddress := "127.0.0.1"

// Set versioning and metadata
$settings.versioning := New object()
$settings.versioning.version := "1.0.0"
$settings.versioning.copyright := "copyright"
$settings.versioning.companyName := "4D SAS"

// Create the client build
$build := cs.Build4D.Client.new($settings)
$success := $build.build()

// Generate the client archive to be included in the server build
$archive := $build.buildArchive()

// Check the status
If ($success & $archive.archive.exists)
    $result:=New object("success"; $success; "logs"; $build.logs; "archiveClientWinPath";
    $archive.archive.platformPath)
```

```

Else
    $result:=New object("success"; $success; "logs"; $build.logs)
End if

```

2. Server Build Method:

Configures and builds the server-side application, including network parameters to support multiple client connections. Execute the code below to get the server side of the application.

```

#DECLARE()->$result : Object

// Declare variables
var $build : cs.Build4D.Server
var $settings : Object
var $success : Boolean

// Initialize settings
$settings := {}
$settings.startElevated := True

// Define the path to the 4D project
$settings.projectFile := Folder(fk desktop folder).file("demo-Build4D-Win/Project/demo-Build4D-Win.4DProject")

// Define the path to the 4D Server application
$settings.sourceAppFolder := Folder(fk documents folder).folder("4D 20 R8 100333/4D Server")

// Define output folder and application name
$settings.buildName := "serverApp"
$settings.destinationFolder := Folder(fk desktop folder).folder("buildApp/Server")

// Set versioning and metadata
$settings.versioning := New object()
$settings.versioning.version := "1.0.0"
$settings.versioning.copyright := "copyright"
$settings.versioning.companyName := "4D SAS"

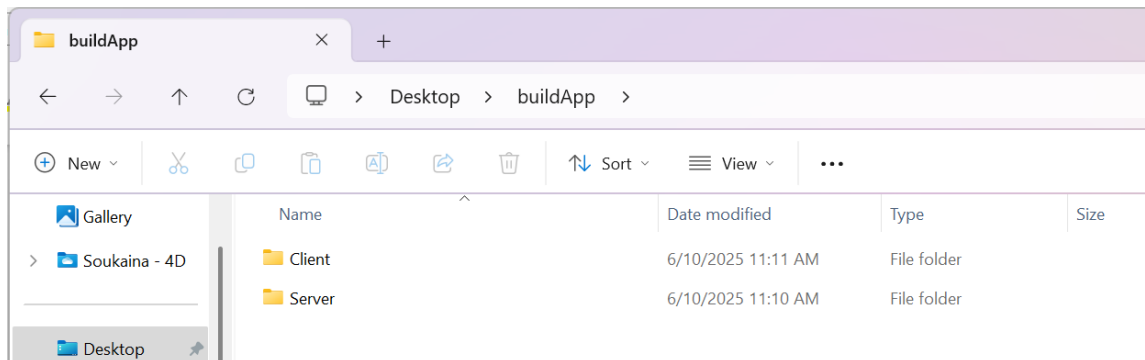
// Add the widows client archive
If ($archiveClientPath#Null & File($archiveClientPath; fk platform path).exists)
    $settings.windowsClientArchive:=$archiveClientPath
End if

// Create the server build
$build := cs.Build4D.Server.new($settings)
$success := $build.build()

// Return build status and logs
$result := New object("success"; $success; "logs"; $build.logs)

```

After executing the two methods, both the client and server apps are now available in the destination folder.



Automated Workflows with CI/CD and GitHub Actions

In modern software development, **CI/CD pipelines** (Continuous Integration and Continuous Deployment) are essential for automating code testing, building, and deployment. They improve code quality, speed up release cycles, and ensure consistency across environments.

This chapter introduces the basics of CI/CD and explains how to create automated workflows using **GitHub Actions**, including how to define “.yaml” files and configure workflow steps.

1. What is CI/CD?

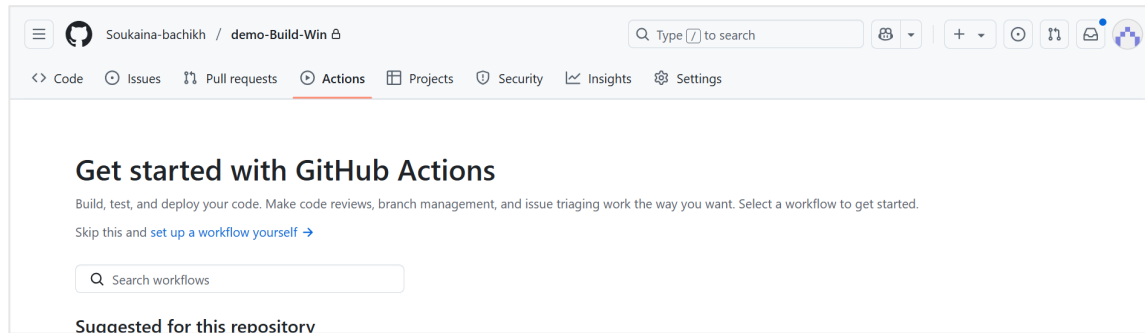
- **Continuous Integration (CI)** is the practice of automatically building and testing code whenever changes are pushed to the source repository.
- **Continuous Deployment (CD)** extends CI by automatically deploying the application once the build passes all tests and validations.

Together, CI/CD enhances team collaboration, reduces production bugs, and ensures consistency across development, staging, and production environments.

2. GitHub Actions Workflows

GitHub Actions is a CI/CD platform built into GitHub that enables developers to automate tasks such as building, testing, and deploying applications. Workflows are triggered by events like pushes to a branch, pull requests, or scheduled runs. This makes GitHub Actions highly suitable for modern development pipelines, including automated 4D builds.

Each workflow is defined using a .yaml file placed inside the .github/workflows/ directory of the repository. These YAML files describe the sequence of operations that should occur, such as checking out code, executing build commands, or deploying the result.



This view of GitHub Actions appears when no workflows are yet configured. From here, it's possible to create new workflow files manually or select from prebuilt templates to get started with CI/CD automation.

3. Setting Up .yaml Files

A workflow file typically consists of three main elements:

1. **Triggers:** Events that start the workflow (e.g., `push`, `pull_request`, or `workflow_dispatch` (manual trigger))
2. **Jobs:** Independent tasks that run on a GitHub-hosted or self-hosted runner
3. **Steps:** Commands or actions executed within a job

Example: Here's a simple example of a workflow that runs on every push to the main branch:

```
# Name of the workflow as it will appear in the GitHub Actions UI
name: Build 4D Application

# Define the trigger: this workflow runs on every push to the 'main' branch
on:
  push:
    branches: - main

# Define the job(s) to run
jobs:
  build: # Job name (can be any identifier)
    runs-on: windows-latest # Specifies the OS/environment for the job runner
    steps:
      # Step 1: Checkout the code from the repository
      - name: Checkout repository
        uses: actions/checkout@v3 # Official GitHub Action to clone the repo

      # Step 2: Run a shell command as a placeholder for the actual build process
      - name: Build step
        run: echo "Running build steps..." # Replace this with actual build commands (e.g., Tool4D)
```

Integrating Build4D into CI/CD

To support automated builds in CI/CD pipelines, 4D provides a command-line utility called **Tool4D**. It enables executing specific project methods without launching the full graphical environment, making it ideal for automation workflows.

Automating 4D Tasks with Tool4D

Tool4D is a command-line utility provided by 4D that operates in a headless, minimal-footprint mode. It enables developers to:

- Execute 4D methods directly via the CLI (--startup-method),
- Run projects in headless or dataless mode (e.g., for compilation, testing, maintenance),
- Integrate seamlessly into CI/CD workflows—making it ideal for automated builds, unit tests, project compilation, and scripting tasks—all without launching the full GUI interface.

Example:

```
"C:\Program Files\4D\4D 20.6\tool4d.exe" "C:\Projects\MyApp\MyApp.4DProject" --build
```

Where:

- "C:\Program Files\4D\4D 20.6\tool4d.exe" is the path to the tool4d executable (adjust based on 4D version and installation).
- "C:\Projects\MyApp\MyApp.4DProject" is the path to 4D project.
- --build is a command-line option that tells Tool4D to build the project (others include --compile, --check, etc.).

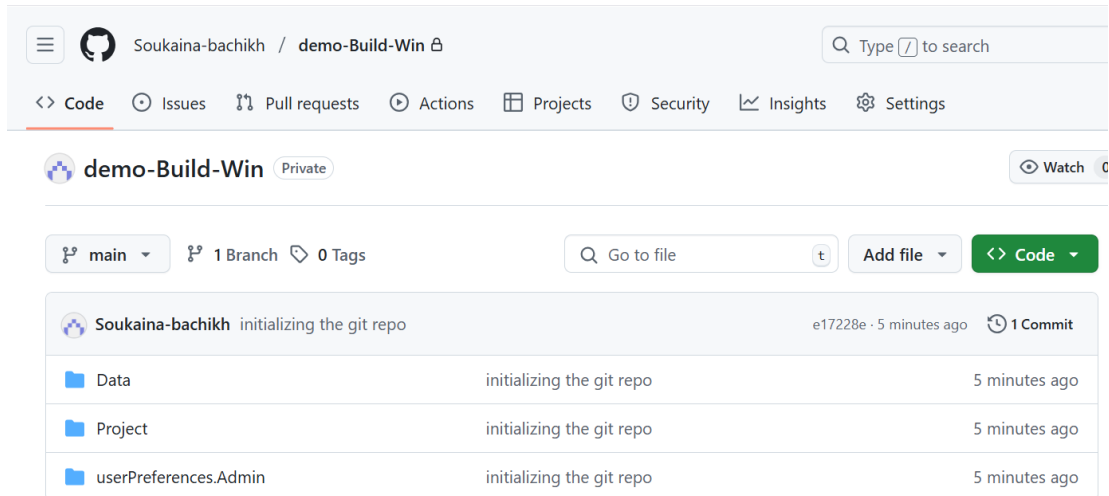
Note: Ensure the project is compatible with the version of Tool4D being used.

Demo: Automating a 4D Build Pipeline

This demonstration outlines the automation of a 4D client-server build pipeline using version 20 R8 (build 100333) for Windows platform. The process is broken down into a series of steps to implement the automation effectively. Please refer to project ".zip" attached for the demo materials.

1. Clone the project to a GitHub repository

After creating a 4D project locally, the next step is to push it to a GitHub repository to take advantage of GitHub Actions.

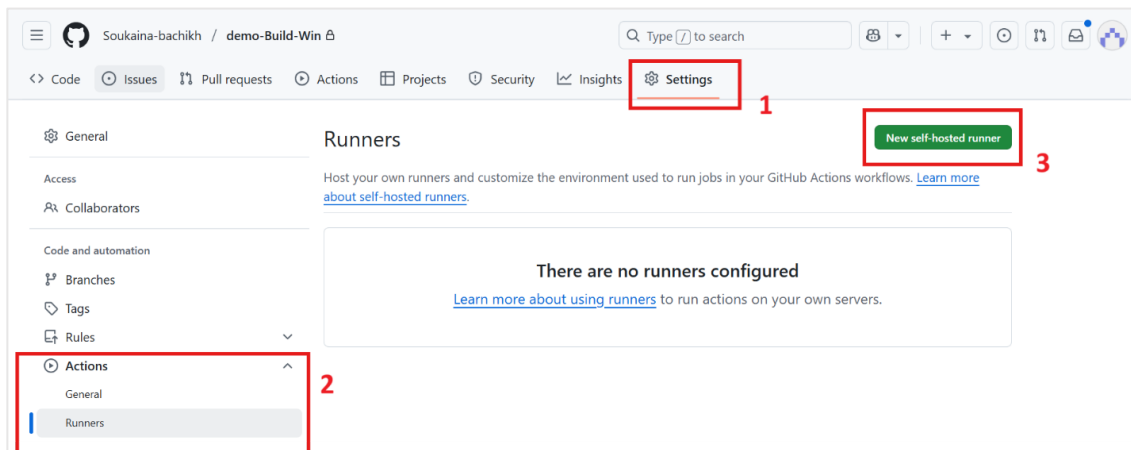


2. Create a runner

To run Tool4D with Build4D on GitHub, a runner is required. GitHub supports two types of runners: **GitHub-hosted runners**, which are managed by GitHub and run in the cloud, and **self-hosted runners**, which run on machines managed by the user. In this demo, a **self-hosted runner** on Windows is used.

To create a self-hosted runner on GitHub:

Go to the GitHub repository → Settings → Actions → Runners → New self-hosted runner.



Then, follow the instructions provided by GitHub to configure the runner. Start by selecting the appropriate operating system on the GitHub interface—Windows in this case—then open PowerShell on the local machine (recommended: Run as Administrator) and execute the setup commands provided.

Once the configuration is complete, the self-hosted runner will be registered and ready with the latest settings.

```
PS C:\Users\Admin\actions-runner\actions-runner> ./config.cmd --url https://github.com/Soukaina-bachikh/demo-Build-Win
-token [REDACTED]

-----
  G I T H U B  A C T I O N S
-----
Self-hosted runner registration

# Authentication

V Connected to GitHub

# Runner Registration
```

Finally, start the GitHub runner to connect it to the repository.

```
PS C:\Users\Admin\actions-runner\actions-runner> ./run.cmd
1 file(s) copied.

V Connected to GitHub

Current runner version: '2.325.0'
2025-06-24 09:31:52Z: Listening for Jobs
```

3. Create the Core Build Methods

Before setting up the workflow files, it's essential to create two core methods within the 4D project: `compileMethod` and `buildMethod`. These methods serve as entry points that will be triggered by Tool4D during the automated GitHub workflows.

- `compileMethod`:

This method is responsible for compiling the entire project and verifying that the source code is free of syntax errors. It uses the built-in `Compile project` method provided by 4D to perform a full project compilation. This step ensures that only valid, error-free code proceeds to the build phase, preventing failures during deployment.

```
// Log that the compilation process has started
LOG EVENT(Into system standard outputs; "Compilation starts\r\n")

// Declare variables
var $projectPath : Text
var $result : Integer
var $compilationReturn : Object

// Retrieve the project path passed through the --user-param parameter in the Tool4D CLI
$result := Get database parameter(User param value; $projectPath)
```



```
// If a valid project path is provided, compile the project and log the result
If ($projectPath#""")
    $compilationReturn := Compile project(File($projectPath))
    LOG EVENT(Into system standard outputs; "Compilation returns : \r\n" + JSON
Stringify($compilationReturn; *))
End if
```

- buildMethod:

This method contains the logic for building the final client/server application using the Build4D component. It should include the necessary build settings based on the targeted deployment (e.g., client-server architecture, folder paths, versioning, etc.).

The buildClient() and buildServer() are mentioned in the build scenarios section.

```
// Log the launch of the build process
LOG EVENT(Into system standard outputs; "✅ Starting Build Process\r\n")

// Declare result objects
var $resultClient; $resultServer : Object

// Run the client build
LOG EVENT(Into system standard outputs; "👉 Running buildClient...\r\n")
$resultClient := buildClient()
LOG EVENT(Into system standard outputs; "👉 buildClient result: " + JSON Stringify($resultClient) +
"\r\n")

// Check if client build succeeded
If ($resultClient.success)
    // Proceed with server build
    LOG EVENT(Into system standard outputs; "👉 Running buildServer...\r\n")
    $resultServer := buildServer()
    LOG EVENT(Into system standard outputs; "👉 buildServer result: " + JSON
Stringify($resultServer) + "\r\n")

    // Final check for both builds
    If ($resultServer.success)
        LOG EVENT(Into system standard outputs; "✅ All builds succeeded!\r\n")
    Else
        LOG EVENT(Into system standard outputs; "❌ Server build failed.\r\n")
    End if
Else
    // Client build failed – stop early
    LOG EVENT(Into system standard outputs; "❌ Client build failed. Aborting server build.\r\n")
End if
```

4. Create the workflows files

In this demo, two workflows are used: one for compiling and verifying the project, and another for building the project using the Build4D component. To create the workflows,

navigate to the Actions section on GitHub or add workflow files under the .github/workflows folder in the repository.

- Compiling workflow:

This workflow automates compiling the 4D project using Tool4D on a self-hosted Windows runner and checks the compilation result.

```
name: Compile 4D Project

on:
  push:
    paths:
      - 'Project/**'      # Trigger workflow only if files inside the 'Project' folder change
  workflow_dispatch:      # Also allow manual triggering of the workflow at any time

jobs:
  compile:
    runs-on: [self-hosted, Windows] # Run this job on a self-hosted Windows 64-bit runner

    env:
      TOOL4D_PATH: C:\Users\Admin\Downloads\tool4d_win\tool4d\Tool4D.exe # Environment variable
                                # for Tool4D executable path

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4 # Step to clone the repo so the runner has access to project files

      - name: Compile with Tool4D
        shell: cmd # Use Windows CMD shell for this step
        run: |
          %TOOL4D_PATH% --project "C:\Users\Admin\Desktop\demo-Build4D-Win\Project\demo-
          Build4D-Win.4DProject" --dataless --log-level=0 --user-param "C:/Users/Admin/Desktop/demo-
          Build4D-Win/Project/demo-Build4D-Win.4DProject" --skip-onstartup --startup-method
          "compileMethod" > compile_output.txt
          # Run Tool4D to compile the 4D project using the 'compileMethod' startup method
          # Output is redirected to compile_output.txt for later analysis

      - name: Check compilation result
        shell: powershell # Use PowerShell to analyze the compilation output
        run: |
          $content = Get-Content compile_output.txt -Raw # Read entire output log file
          Write-Output "📄 Tool4D Output:"
          Write-Output $content # Print the output content in the workflow logs

          # Check if output contains failure indicators: success=false or errors present
          if ($content -match '"success"\s*:\s*false' -or $content -match '"errors"\s*:\s*\[^\]]+') {
            Write-Output "❌ Compilation completed with errors!"
            exit 1 # Fail the workflow if errors detected
          } else {
            Write-Output "✅ Compilation succeeded with no errors!"
          }
        }
```

- Building workflow:

This workflow automates building the 4D client-server application using Tool4D on a self-hosted Windows runner and reports the build outcome.

```
name: Build 4D Project

on:
  workflow_dispatch:      # Still allow manual trigger anytime

jobs:
  build:
    runs-on: [self-hosted, Windows, X64]    # Run this job on a self-hosted Windows 64-bit runner

    env:
      TOOL4D_PATH: C:\Users\Admin\Downloads\tool4d_win\tool4d\Tool4D.exe # Path to the Tool4D executable

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4    # Clone the repository so the runner has access to the project files

      - name: Build with Tool4D
        shell: cmd    # Use Windows CMD shell for this step
        run: |
          %TOOL4D_PATH% --project "C:\Users\Admin\Desktop\demo-Build4D-Win\Project\demo-Build4D-Win.4DProject" --dataless --skip-onstartup --startup-method "buildMethod" > build_output.txt
          # Execute Tool4D to build the 4D project using the 'buildMethod' startup method
          echo "📄 Tool4D Build Output:"
          type build_output.txt

      - name: Check build result
        shell: powershell    # Use Windows PowerShell for this step
        run: |
          $content = Get-Content build_output.txt -Raw
          # Verify the build result if it has been succeeded or failed
          if ($content -notmatch '✅ All builds succeeded!') {
            Write-Output "❌ Build did not complete successfully."
            exit 1
          } else {
            Write-Output "✅ Build log confirms successful execution."
          }
        }
```

5. Trigger the workflows for compiling and building

After setting up the workflows, the next step is to trigger them to initiate the compilation and build processes. This can be done manually or automatically based on the configured triggers.

Trigger the compiling workflow

Under Actions, click on “Compile 4D Project” to trigger manually the workflow

The screenshot shows the GitHub Actions interface for the 'Compile 4D Project' workflow. On the left, the 'Actions' tab is active, and 'Compile 4D Project' is selected under the 'Build 4D Project' section. A red box highlights this selection, with a red '1' next to it. The main panel shows '0 workflow runs'. A 'Run workflow' button is visible, and a dropdown menu is open, showing 'Branch: main' and a 'Run workflow' button, which is also highlighted with a red box and a red '2'.

When triggering the compilation workflow, two scenarios can occur: success or failure.

Compilation Success

The project compiles without errors, confirming that the source code is valid and ready for the build step. The logs indicate a successful compilation, allowing the pipeline to proceed.

After a while the workflow has succeeded

The screenshot shows the GitHub Actions interface for the 'Compile 4D Project' workflow. On the left, the 'Actions' tab is active, and 'Compile 4D Project' is selected. The main panel shows '1 workflow run' with a green checkmark indicating success. The workflow name 'Compile 4D Project' is highlighted, and the status 'main' is shown. The run was triggered manually by Soukaina-bachikh 2 minutes ago and took 35s to complete.

To view more details, click on the workflow name and go through the steps to check each step

compile

succeeded now in 31s

Search logs

Compile with Tool4D

6s

```
1  run /Tool4D_FAT16 --project "C:/Users/Admin/Desktop/demo-Build4D-Win/Project/demo-Build4D-Win.4DProject" --data1233
--log-level=0 --user-param "C:/Users/Admin/Desktop/demo-Build4D-Win/Project/demo-Build4D-Win.4DProject" --skip-
onstartup --startup-method "compileMethod" > compile_output.txt
6
```

Check compilation result






1s

```
1  Run $content = Get-Content compile_output.txt -Raw
15
15  ðŸŽŸ, Tool4D Output:
16  Compilation starts
17  Compilation returns :
18  {
19      "success": true,
20      "errors": []
21  }
22  âœ… Compilation succeeded with no errors!
```

Post Checkout repository

6s

Note: on push, an automatic trigger occurs (in this case: merge...)

	Merge branch 'main' of https://github.com/Soukaina...	main	 now	...
	Compile 4D Project #2: Commit e82a4d5 pushed by Soukaina-bachikh		 In progress	
	Compile 4D Project	main	 22 minutes ago	...
	Compile 4D Project #1: Manually run by Soukaina-bachikh		 35s	

Compilation Failure

If the compilation detects errors, the workflow stops and reports them. Detailed logs help identify the issues, allowing developers to resolve them before attempting a new build.

In this demo, a syntax error is intentionally introduced in the `host_startup()` method to trigger a compilation failure.

demo-Build4D-Win-4D - [Method: host_startup]

File Edit Run Design Records Tools Method Window Help

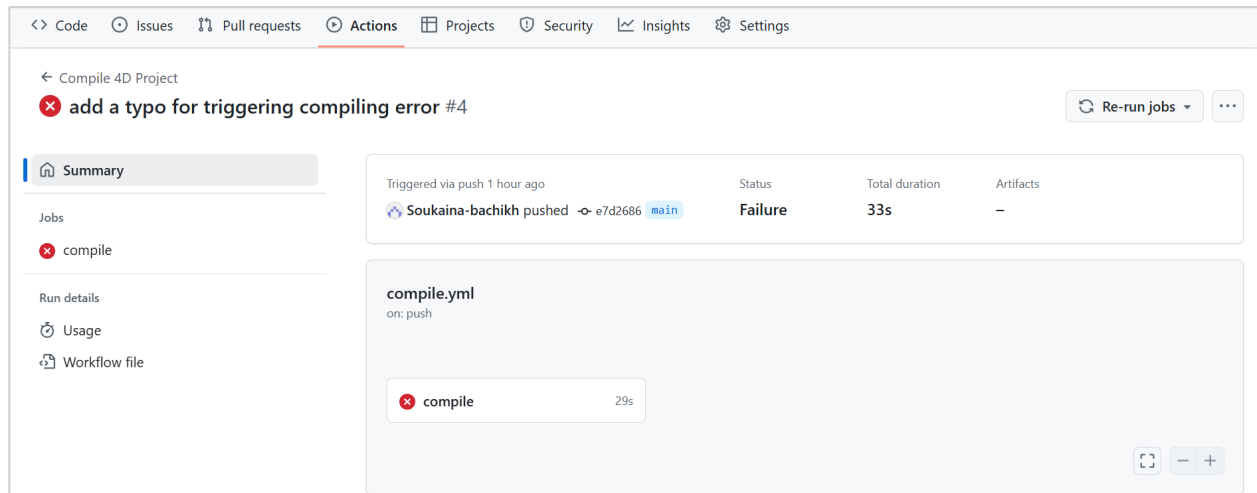
New Open Explorer Tool Box Structure Compiler Find in design Data Tables Query Tools Execute MSC Settings

1 **ALERT**("Hello Build4D !")

2

3 typo:=|

After pushing the changes to GitHub, the compilation workflow is automatically triggered and reports the error.



Compile 4D Project

add a typo for triggering compiling error #4

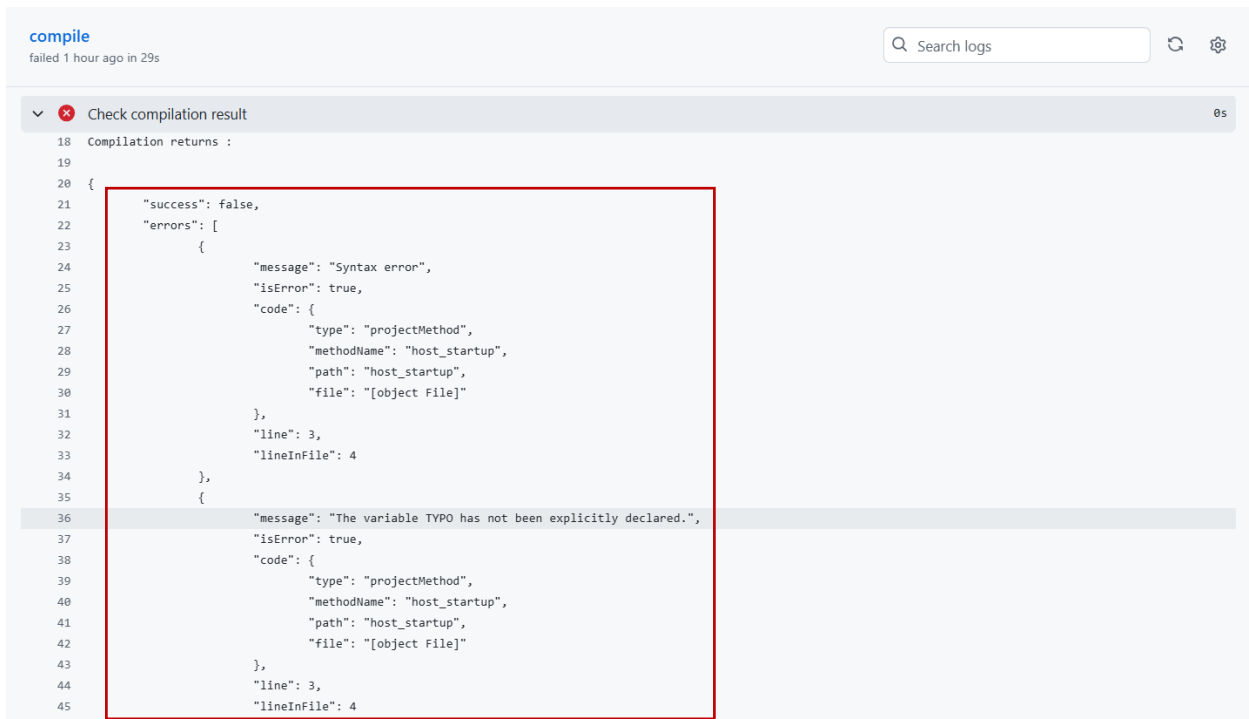
Triggers: Soukaina-bachikh pushed · e7d2686 · main

Status: **Failure** | Total duration: 33s | Artifacts: -

Workflow file: **compile.yml** (on: push)

Jobs: **compile** (29s)

To review the errors, developers can consult the logs under the **Compile** job in the workflow run. This helps ensure that issues are addressed before proceeding to the build process.



compile

failed 1 hour ago in 29s

Search logs

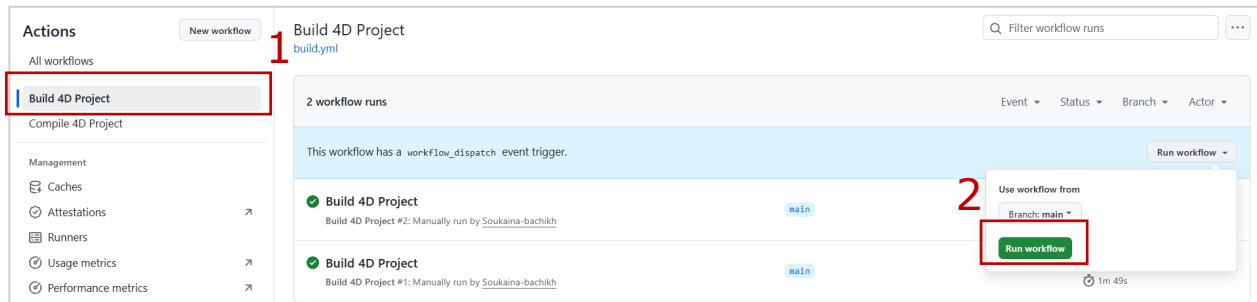
Check compilation result

```
18 Compilation returns :
19
20 {
21   "success": false,
22   "errors": [
23     {
24       "message": "Syntax error",
25       "isError": true,
26       "code": {
27         "type": "projectMethod",
28         "methodName": "host_startup",
29         "path": "host_startup",
30         "file": "[object File]"
31       },
32       "line": 3,
33       "lineInFile": 4
34     },
35     {
36       "message": "The variable TYP0 has not been explicitly declared.",
37       "isError": true,
38       "code": {
39         "type": "projectMethod",
40         "methodName": "host_startup",
41         "path": "host_startup",
42         "file": "[object File]"
43       },
44       "line": 3,
45       "lineInFile": 4
46     }
47   ]
48 }
```

Trigger the building workflow

Once the compilation workflow completes successfully, the building workflow can be triggered manually from the **Actions** tab by selecting “**Build 4D Project**” and clicking “**Run workflow**”.

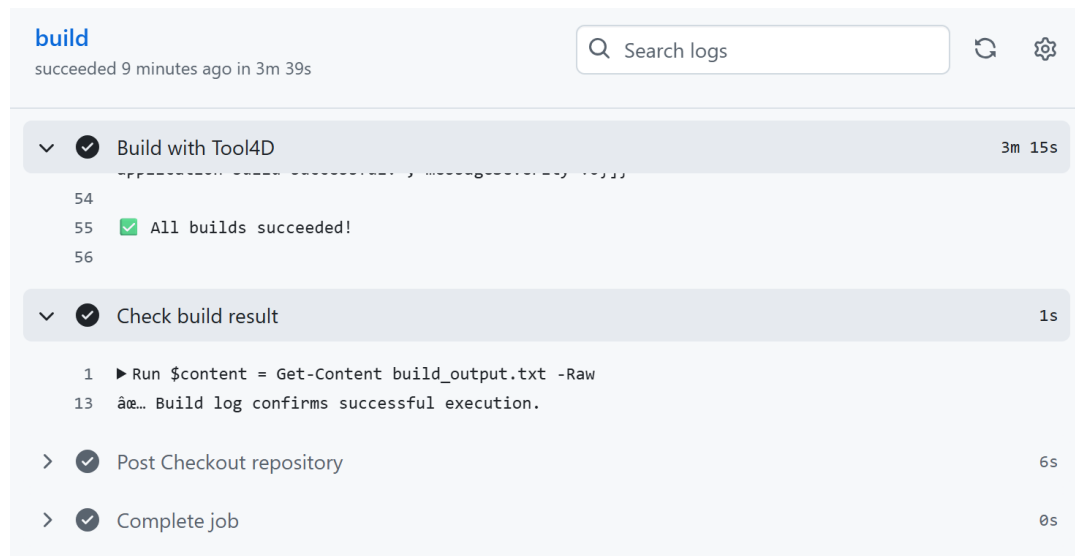
This step executes the buildMethod to generate the client/server.



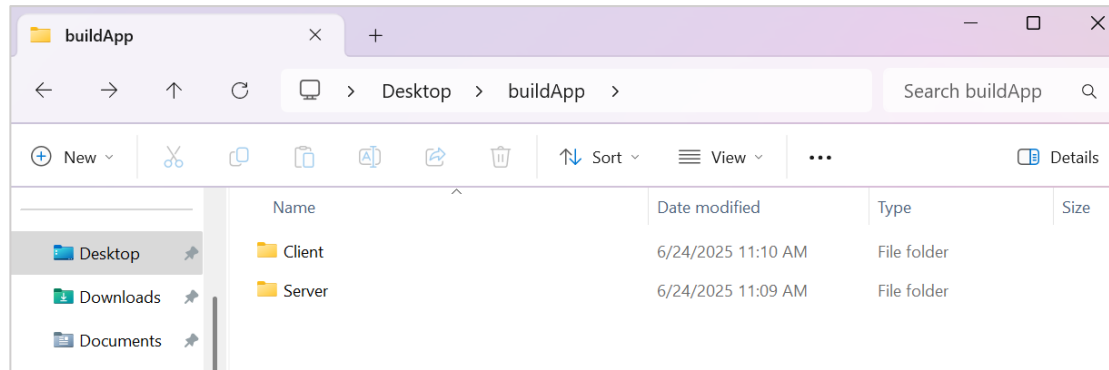
When triggering the building workflow, two scenarios can occur: success or failure.

Build Success

The client/server application is built successfully. The workflow completes without errors, and a success message is displayed in the logs, confirming that the build process executed as expected.

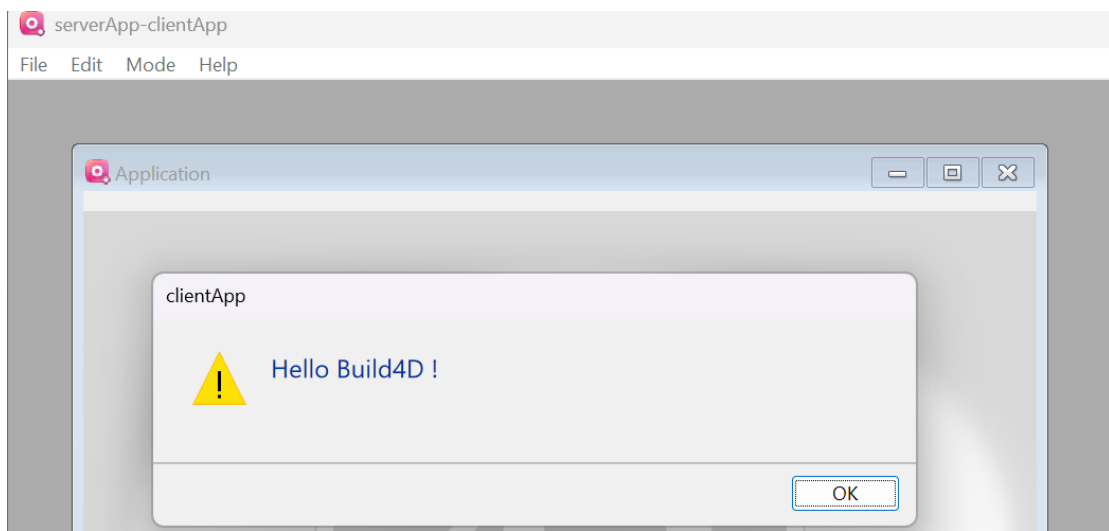


On the local machine where the self-hosted runner is configured, the built application is available in the specified output directory.

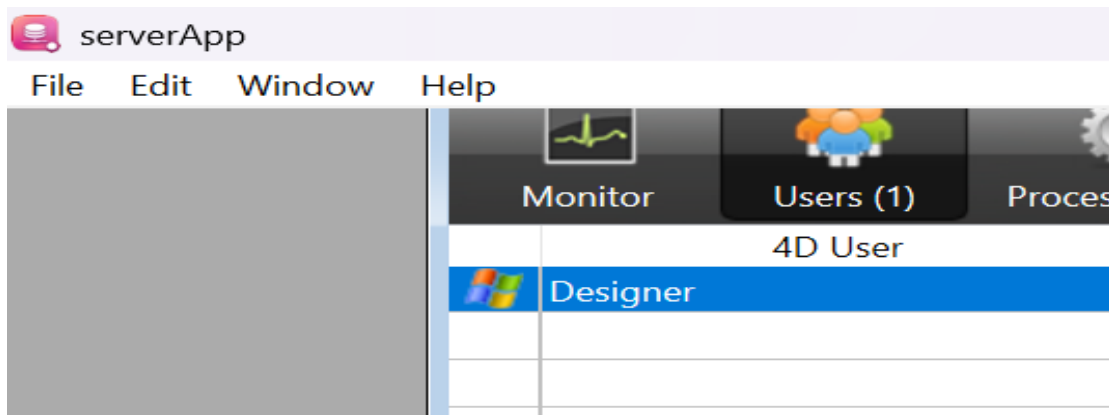


Run the client and server application to get the following result:

Client side: an alert appears within the startup method representing the host database



Server side: the client is connected to the server successfully



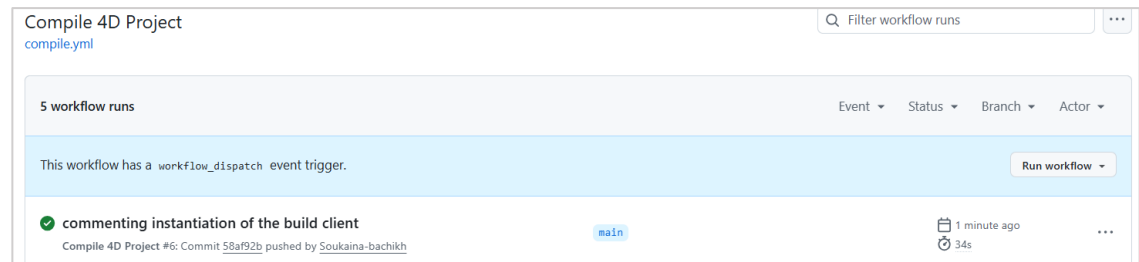
Build Failure

If an issue occurs during the build (e.g., misconfiguration, missing files, or logic errors), the workflow stops and logs the failure. These logs provide useful information to help identify and resolve the problem before reattempting the build.

For example, commenting the destination folder

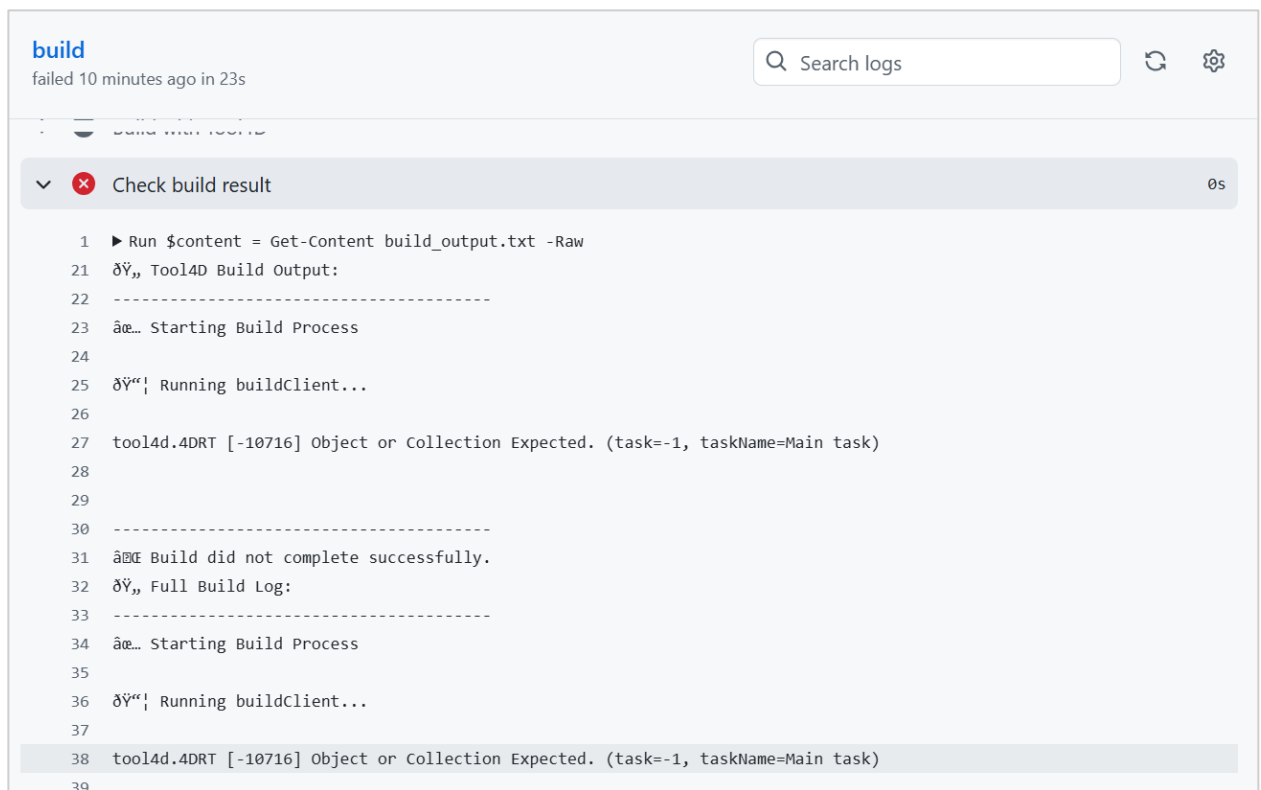
```
33
34 // Create the client application
35 // $build:=cs.Build4D.Client.new($settings)|
36 $success:=$build.build()
37
```

The compilation has succeeded



But when triggering the build we got this error on the job details:

“tool4d.4DRT [-10716] Object or Collection Expected. (task=-1, taskName=Main task)”



Here's a brief recap of the demo steps:

1. **Push the 4D project to GitHub** to enable CI with GitHub Actions.
2. **Set up a self-hosted Windows runner** in GitHub to run builds locally.
3. **Create two workflows:**
 - Compile workflow to check for compile success using **compileMethod**.
 - Build workflow to build the client/server app using **buildMethod**.
4. **Capture logs and check for errors or success markers** to determine workflow status.
5. **Trigger workflows:** compilation runs on push, build runs manually.
6. **Review results:** successful compile allows building; failures show detailed logs for fixes.

Conclusion

This technical note outlined the core principles and usage of the Build4D component, designed to simplify and automate the build process for 4D projects. It covered how to integrate Build4D with and configure it within a CI/CD pipeline using GitHub Actions and self-hosted runners. With this guidance, developers can efficiently set up automated builds, reduce manual intervention, and ensure consistent application delivery.