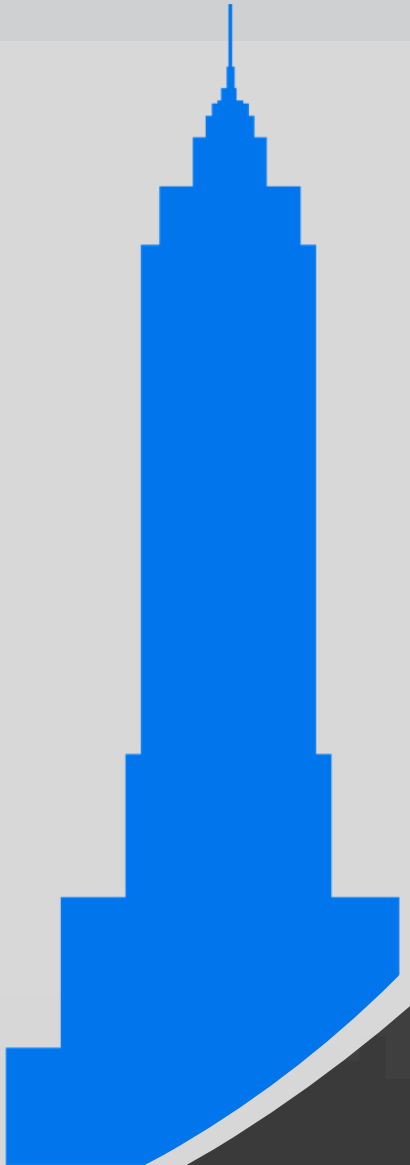
riganti



Assembly Bindings, Binding Redirections & Debugging

Andrej Čižmárik

Motivation: Weird Assembly Load Errors

- Failure to load an assembly during runtime
- Assembly binding fails usually manifest using an exception
  - TypeLoadException, FileNotFoundException, FileLoadException or BadImageException

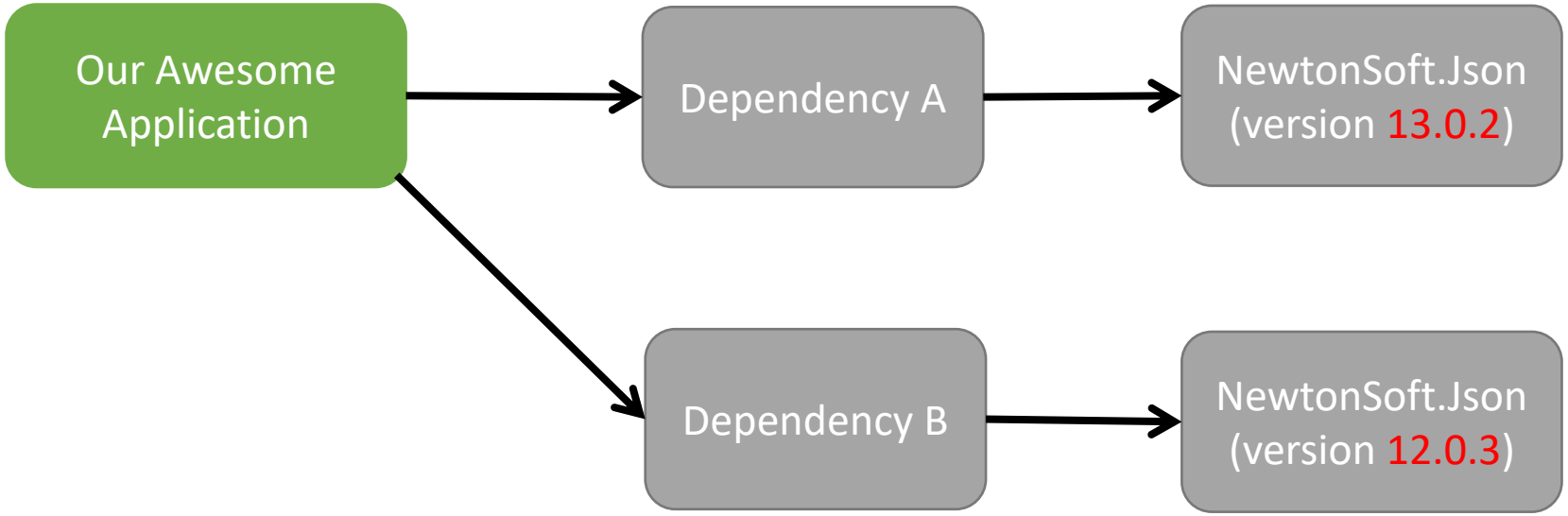
System.IO.FileLoadException: Could not load file or assembly 'MyCoolAssembly, Version=1.2.3, Culture=neutral, PublicKeyToken=367d582291c765f7' or one of its dependencies. The located assembly's ***manifest definition does not match the assembly reference***

Assembly Binding & .NET Framework

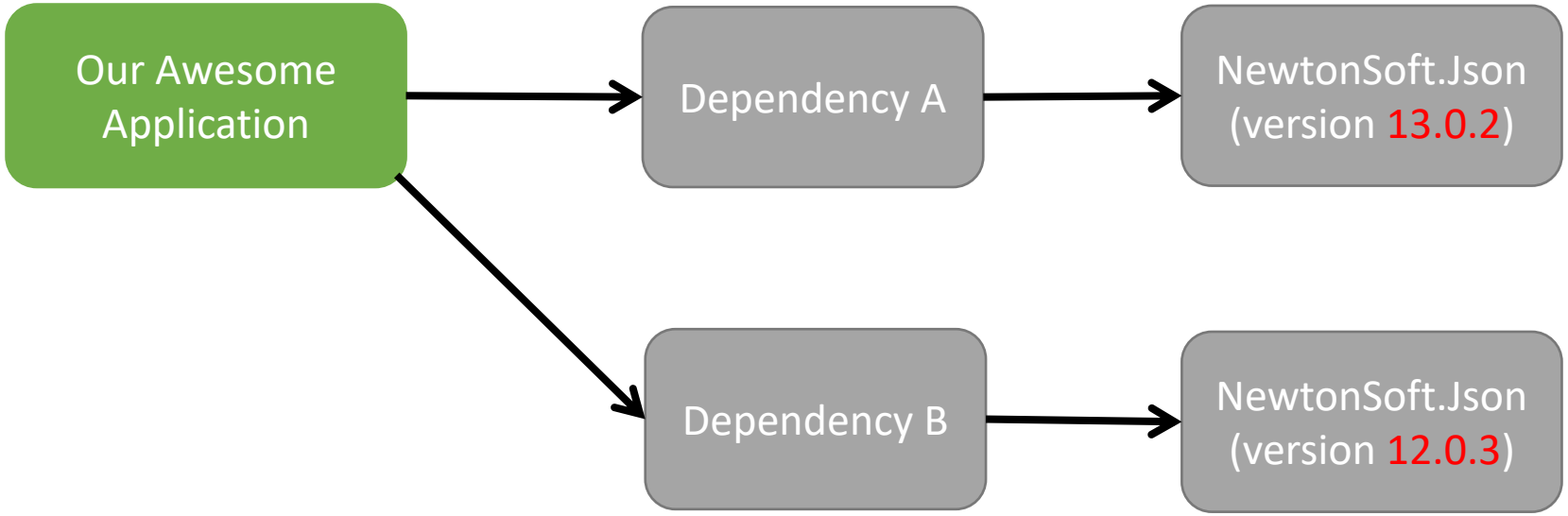
- Somewhat successful approach to solve the **DLL Hell** [1] problem
- However:
  - Way **too strict and complex rules** for real-world applications [2]
  - Introduces new problem: **Assembly Binding Redirects**

- https://en.wikipedia.org/wiki/DLL_Hell
- https://learn.microsoft.com/en-us/dotnet/framework/deployment/how-the-runtime-locates-assemblies

Assembly Binding Failure



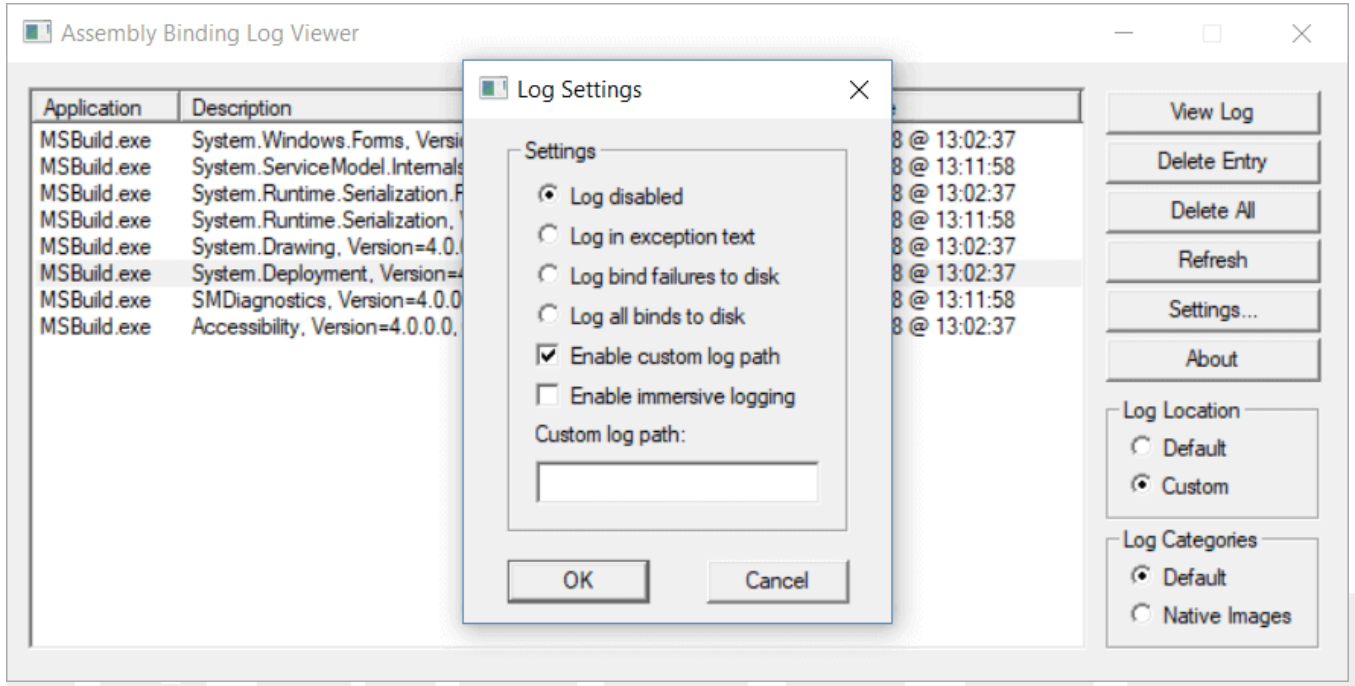
Assembly Binding Fixed




```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <dependentAssembly>
    <assemblyIdentity name="Newtonsoft.Json" publicKeyToken="30ad4fe6b2a6aeed" culture="neutral" />
    <bindingRedirect oldVersion="0.0.0.0-13.0.0.0" newVersion="13.0.0.0" />
  </dependentAssembly>
</assemblyBinding>
```

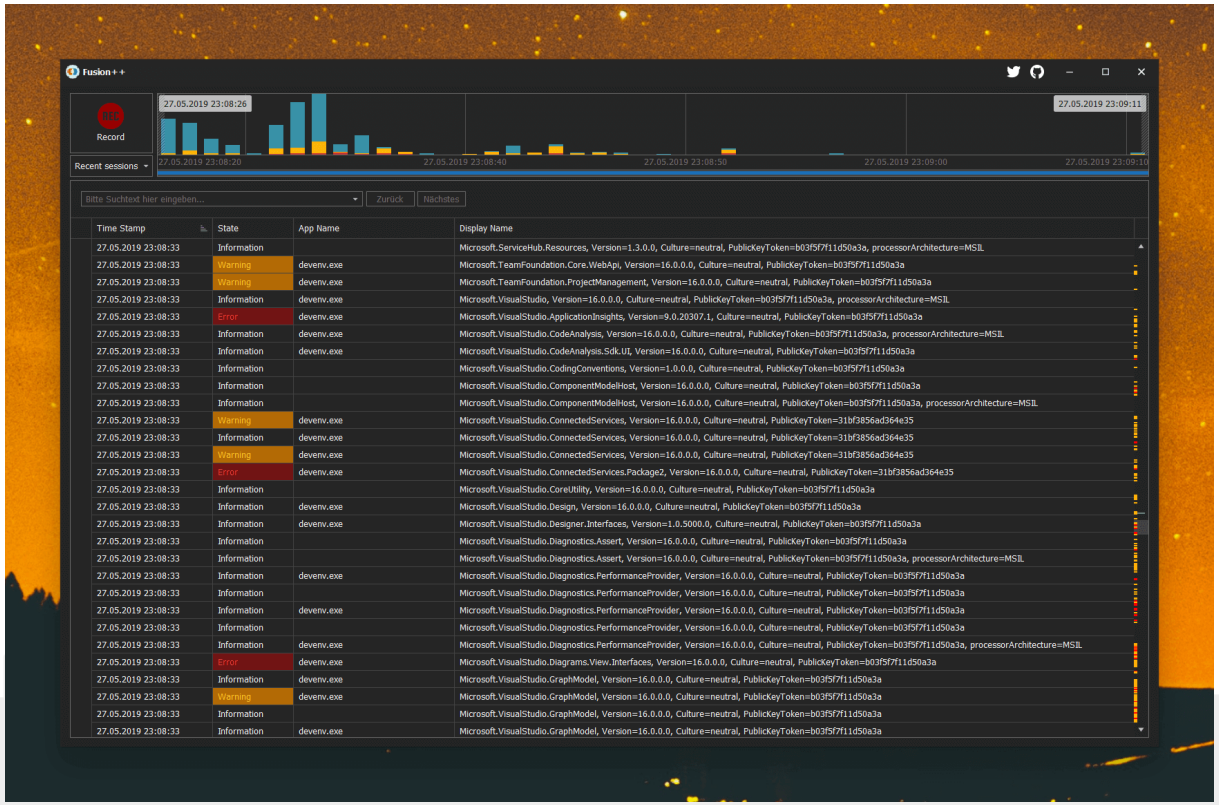
Fusion

- Original application shipped with every .NET Framework installation
- Very clumsy, old and error-prone UI



Fusion++

- Modern, open-source alternative to original fusion
- https://github.com/awaescher/Fusion

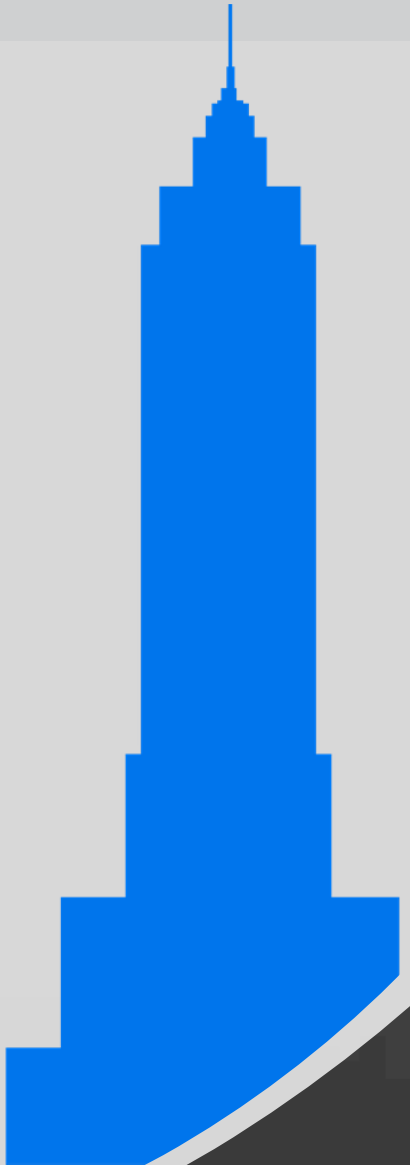


Examples

What about .NET Core?

- Assembly versions **do not need to strictly match** build / runtime
- New runtime approach
  - Ignore strong naming
  - Allow almost any version at runtime
  - **`AssemblyLoadContext`** (replaces AppDomains)

riganti



.NET 7 & AOT
Native Interoperability

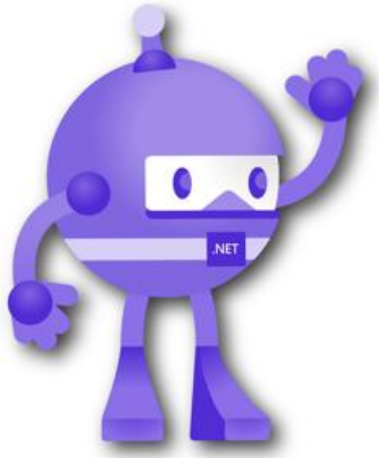
Andrej Čižmárik

Motivation for AOT

- Compile .NET class libraries into **native libraries** (static or shared)
- **Linked with a lightweight runtime** (no need for JIT, metadata, …)
- Better **startup times** (methods are already compiled)
- Overall **performance improvements**
- **Small executables**

Use-cases for AOT

- **Interoperability** with native environments
  - C, C++, Pascal, Rust…
- **Short-lived cloud applications** (for example, Azure Functions)
  - No lag from JIT, execute faster, pay less ☺
- Access to **restricted platforms** (no runtime code generation)



Disadvantages of AOT

- Many **NuGets** are not compatible yet
- **Reflection** can get tricky
  - Some use-cases are not supported out of the box and require additional configuration (for example, do not strip metadata)
- **Diagnostics** can get tricky
  - Can not use managed debugger, profiler…
  - However, application can be analyzed using managed build
- No support for **dynamic loading** and **runtime code generation**

Example: Calling C# from C [1/2]

```
[UnmanagedCallersOnly(EntryPoint = “csharp_add_method")]
internal static int Add(int arg1, int arg2)
{
      return arg1 + arg2;
}
```

- Create native entrypoint using **UnmanagedCallersOnly**
  - This method can not be called from managed code
  - However, it can call other managed code
- Publish with `/p:PublishAot=true p:NativeLib=Shared -r win-x64`

Example: Calling C# from C [2/2]

- Implement sample C application

```
#include <stdint.h>

int32_t csharp_add_method(int32_t a, int32_t b);

int main()
{
	printf("Hello world!");
	printf("%d", csharp_add_method(1, 2));
}
```

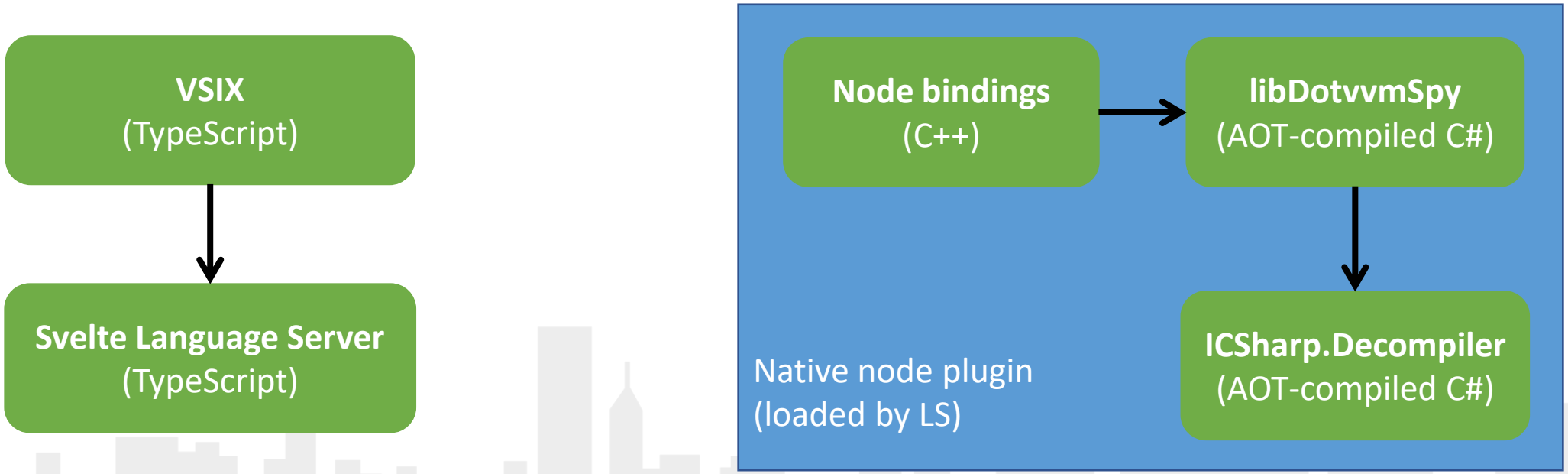
Example: Reflection and AOT

- Let's now consider that we need to do some **reflection-heavy stuff**, such as serialization using Newtonsoft.Json NuGet
- We need to instruct **IL Linker** not to strip metadata
- Add file **rd.xml** in the root folder of C# library

```
<Directives xmlns="http://schemas.microsoft.com/netfx/2013/01/metadata">
  <Application>
    <Assembly Name="Newtonsoft.Json" Dynamic="Required All"/>
  </Application>
</Directives>
```

Example: DotVVM VS Code Extension

- We are working on **libDotvvmSpy** (cca 8.2MB library)
  - AOT-compiled class library that uses ILSpy to inspect assemblies
  - Used to inspect classes for @viewmodel auto-completion



Configuring AOT

- IlcInvariantGlobalization – remove support for non-english cultures
- IlcOptimizationPreference – prefer speed or executable size
- IlcGenerateStackTraceData – disable metadata for stack traces

- Configurability is quite powerful: https://github.com/nikouu/TinyWordle