SEMESTRAL PROJECT

Filip Horký

Peachpied Symfony

Supervisor of the semestral project: Ing. Mgr. Robert Husák

Prague 2019

Contents

Introduction

Peachpied Symfony is a software addressing the current problem of community of server-side developers segregation into several disjoint groups. Each group is focused on slightly different aspect of development, such as security, scalability or simplicity, and thus is using a programming language that answer their preferences the best. The three most widely used programming approaches are PHP, .NET and Java. This software focuses on the first two mentioned.

However, more than a software, Peachpied Symfony might rather be considered a set of tools. These tools are build atop Peachpie, a PHP into .NET compiler. Their main purpose is to simplify the process of migrating, embedding or merging of an existing Symfony project into a .NET environment. Symfony is a widely used PHP framework and thus this software should prove useful to those that want to provide their PHP application with additional security, or enhance it with features of ASP.NET Core. Such process would of course be possible using pure Peachpie compiler as well, however, it would be rather tedious and complicated work. Therefore, Peachpied Symfony provides tools to make this integration as seamless as possible.

Additionally, each Symfony project takes advantage of the huge amount of libraries, called Symfony Components, that provide additional features such as templating, routing or security. As the typical Symfony project uses high tenths of these libraries, development and testing of a software covering such a variety of functionalities is currently not our primary goal. Peachpied Symfony provides only functionalities required for the bare bones Symfony Skeleton projects and for the use of Twig, Symfony's templating language. Therefore, it should be regarded more as a proof of concept, that can be further work based on, rather than a software ready to be used in a production.

Programmer's Guide

As was mentioned before, Peachpied Symfony is a set of tools providing seamless integration of Symfony projects within the .NET environment. Regardless the fact, that each of these tools provide different functionality, they are all parts of a single architectural design, that will be described in a section right ahead. The following section will sum up how are the libraries and Symfony projects compiled. Finally, an overview of additional tools that either simplify the process of integration or provide additional functionalities will be provided.

Architecture

Each Symfony project can be divided into two completely distinct parts. First, consisting of the `vendor` directory, where are all the Symfony Components located, and second, made up from everything else, like front-end and back-end implementation, routing configuration or static files. The only exception to this is the `vendor/Composer` folder, that contains data describing actual installed libraries and provides scripts for their auto-loading. Ultimately, for each set of Symfony components and fixed versions, the `vendor` folder, excluding the `Composer` directory, will always look the same. Moreover, each of those Symfony components can be recognized as an independent unit as well, only holding references to another libraries that it requires for its proper functioning.

Regarding this inner composition, we decided that by separating each of these components into a self contained unit, we will provide the best scalability and easy further usability from the .NET environment. NuGet package manager was selected as a best candidate for this task as it is widely used across the .NET platform and is fully integrated into MSBuild.

Compilation

Libraries Prior to a library compilation, one must first create a `.*proj` file, a configuration for the .NET compiler, containing all the necessary information about the library, from files to be included and excluded during the compilation, to library's dependencies and resulting nuget name. Following the philosophy of Peachpie, `.msbuildproj` extension is used within Peachpied Symfony, however, any another extension could be used as well, as MSBuild does not pay any attention to the actual prefix of `proj` extension.

As there would be many duplicate values between libraries' `.msbuildproj` files, three additional `.props` files were created, providing all the required values to the extent, where actual `.msbuildproj` files only need to specify SDK to

be used during the compilation and Symfony Component's dependencies. These three files are located in the root of the project inside the `Props` directory. The drawback of this approach is that each `.msbuildproj` file has to answer to a certain naming convention and has to be located in a folder containing the `vendor` directory. Furthermore, due to significant differences between PHP and .NET, and a work in progress status of the Peachpie compiler, there might occasionally exist uncompilable files. Props configurations does automatically exclude all content of any `test` and `tests` directories, as these are typical examples of such files. For further custom exclusions, or inclusions of automatically excluded files, `AdditionalExcludes` and `AdditionalIncludes` properties were defined.

Projects The `.msbuildproj` configuration file of each Symfony project is significantly different from those of the packages. By default, it excludes content of all the directories inside the `vendor` folder, with the `Composer` folder being the only exception. This folder is tightly bond to an application's execution, and therefore it has to be present in the project by the time of compilation. In addition to, the project's configuration file currently has to exclude the `bootstrap.php` file from the compilation as Peachpied Symfony provides custom bootstrapping mechanism, and present version of Peachpie does provide the script overwriting only for the scripts that does not exist. Moreover, each project has to contain the pre-generated `cache` folder inside the `var` directory as well, because its generating on runtime is still not fully supported. Ultimately, the package references to `Peachpied.Symfony.Skeleton`, `PeachPied.Symfony.AspNetCore`, and any other Symfony Components' nugets required have to be configured.

PeachPied.Symfony.Skeleton

This nuget has no content and its sole purpose is to group together all the nugets that correspond to the Symfony Skeleton project's `vendor` directory content. There are about 24 Symfony Components that such "minimal" project requires for its start-up. After building a dependency tree of these libraries, we found out that this number can be in fact reduced to 6, while the rest will be provided via package references and dependency resolution. The `Peachpied.Symfony.Skeleton` nuget was created in order to reduce this number even more, by referencing these 6 nugets for the user.

PeachPied.Symfony.AspNetCore

Peachpied.Symfony.AspNetCore nuget provides an easy to use API that abstracts the user from the compiled Symfony project's manual binding to the server's pipeline by supplying a simple to use middleware with the single `UseSymfony` method. Furthermore, it also provides promising features of Twig and Razor templating engines interoperability.

UseSymfony method As the method's name suggests, it adds middleware to the pipeline that handles requests targeting Symfony application. Everything that needs to be provided is a relative path to the root of the project.

Aside from calling the `UsePhp` method provided by the `Peachpie.AspNetCore.Web` that implements the actual middleware, the method configures static files' serving middleware, URL rewriting to satisfy Symfony's expectation of the `public` directory being configured as a root of the application, and overrides Symfony's application bootstrapping. The later mentioned is done in order to enable the Symfony project to be configured either by the `.env` and the `.env.local` files as is usually done in Symfony, or by the `appsettings.json` and the `appsettings.Development.json` as is common practise in .NET.

The actual configuration is then handled by the `SymfonyConfig` class, that defines properties corresponding to the environmental variables in PHP, and the `SfConfigurationLoader` class that provides the configuration loading and the `.env*` files parsing.

Twig into Razor The `TwigRenderer` class provides very promising tools to embed Twig templates within Razor templates. Furthermore, it can be all done without the need for the whole `Peachpied.Symfony.Skeleton` nuget, but only through using two additional nugets, the `Twig.Twig` and its single dependency, the `Symfony.Polyfill-mbstring`.

In order to make this process as simple as possible, the `PeachPied.Symfony.AspNetCore.Templating` namespace provides the `TwigRenderer` class, with the static `RenderTwig` function that can be directly accessed and used from a Razor template. Furthermore, additional `DataToPhp` function was implemented in order to provide seamless means for converting a Razor data into the Twig comprehensible form. Finally, the `MergePhpArrays` function offers merging of converted data into a single unit that can be supplied to the `RenderTwig` function afterwards.

The result of the Twig rendering process is stored as a `string` and returned into a Razor template. After that, it can be further processed or directly embedded into the code using the `@Html.Raw()` method.

Razor into Twig Due to a certain level of complexity Razor into a `string` rendering brings, the whole functionality is separated into three functional units, each placed into its own class.

The first unit, the `RenderRazorService` class, provides a functionality for Razor into a `string` rendering, accessible via the `RenderToString` function in the `PeachPied.Symfony.AspNetCore.Templating` namespace. However, such feature requires access to objects like the `RazorViewEngine` or the `ServiceProvider`. To take the advantage of the Dependency Injection of Asp.Net Core, the class needs to be registered as a service in the server's `Startup` class, within the `ConfigureServices` method, implementing the `IRenderRazorService` interface provided. This way, the application provides the necessary objects for us, creating an instance of the `RenderRazorService` that can further accessed through the `ServiceProvider`.

The second functional unit is called the `RSEnvironment`. This class forms a wrapper around the Twig's `Environment` class. The wrapper is located inside the `Twig` namespace and thus directly accessible from a Symfony project. Its purpose is to instantiate the standard Symfony `Environment` class and extend it with an additional function for Razor templates rendering. This is done by using the Twig `Environment`'s `addFunction` extension method. This way, new

`render_razor` function is defined within this instance of the `Environment` that is bind to an actual Razor rendering function situated in the third functional unit. For this reason, the wrapper is named the Razor Supporting Environment, `RSEnvironment` in short.

The last functional unit is titled the `RazorRendererBridge`, as it provides bridging between the `RSEnvironment` and the `RazorRendererService`. Each call of the `render_razor` function within a Twig template results in the `RenderRazorBridge` class instantiating with the current PHP `Context`, and the `renderRazor` function invocation. This method converts the data provided into the Razor comprehensible formate, extracts the current `HttpContext` from the Peachpie's PHP `Context` provided on the instantiation, uses it to access the `RazorRenderService` and returns output from its `RenderToString` method, invoked with the data provided.

Again, the output of Razor rendering is stored as a `string` and returned into Twig. After that, it can be further processed or directly embedded using the `|raw` filter.

Auto Compilation Tool

Mentioned `.props` files reduce the actual work one must do in order to configure Symfony Component's `.msbuildproj` file to just a dependencies specification. However, each Symfony Component contains metadata file `composer.json` that holds general information about the package, such as its name, description, and aside from many other, also a list of Symfony Components it depends on.

The `Additional-Tools/autocompile-vendor.php` is a script that scans content of the `vendor` directory and generates the `.msbuildproj` file using the `.props` files and the `Peachpie.NET.Sdk` SDK for each Symfony Component inside. Prior to the MSBuild configuration generating, it searches configured repositories to check processed library's existence and skips such Symfony Component's compilation in case of success. Afterwards, it parses the `composer.json` metadata, resolves each component's dependencies recursively, and for each successfully resolved component sets a package reference inside the `.msbuildproj` configuration. Other dependencies are simply skipped from dependencies list. This is done because the `composer.json`'s `require` field does not usually list all the Symfony Components that are actually required for its compilation, and `require-dev` on the other hand provides the rest of required with some additional that can be omitted. For this, reason, the auto-compilation script tries to resolve all components from both of those lists.

The recursive dependency resolution is done in two phases. In the first one, the configured repositories are searched for the given package id. In case of failure, the `vendor` directory is searched for a package and is compiled recursively in case of success.

Composer Generator Tool

Each Symfony project contains the `vendor/Composer` directory that holds autoloading scripts and metadata describing actual packages and versions installed.

Naturally, with any library addition, the content of this folder is regenerated. In Symfony, Composer package manager does this automatically during the process of library downloading. The content of this folder is also bound to the cache that the Symfony project produces. This tool does address the problem of the `Composer` folder prior to the project's compilation existence requirement.

At the moment, this script internally calls Composer package manager with the `install` command inside specified project's root. This command results in either the `composer.lock` if present, or the `composer.json` file load-up and reinstallation of all the packages listed. Basically, what this command does is that it completely restores the `vendor` directory. After the `composer install` command finishes, the script executes Cache Generator Tool and then performs final clean-up that removes everything unnecessary from the `vendor` folder.

Cache Generator Tool

The Cache Generator Tool might be regarded as a part of the Composer Generator Tool as it is called during its execution, however, its functionality is completely different. As was discussed, the cache of a Symfony project are bond to the `Composer` folder, therefore its regeneration means we have to perform a regeneration of the cache as well.

The script is currently based on the mechanism Symfony uses for the cache folder generating during its start-up. As this mechanism works with the libraries stored inside the `vendor` directory, the script requires them to be present there as well. For this reason, it is called after the `Composer` folder is generated, but before it performs clean-up of restored packages.

Dependencies Cache Tool

This tool provides a simple way of describing dependencies between NuGet packages using the `.json` file formate. The script is currently without actual purpose, but is planned to be used inside the Composer Generating Tool to enable its generating directly from NuGet packages. This will remove the need for Composer package manager usage and thus avoiding the necessity to download all of the components during each `Composer` folder regeneration.

The script's functionality is very simple. It works with a fact, that each NuGet package is a plain archive that can be easily unzipped and explored. This way, it reaches each nuget's `.nuspec` metadata file and parses out its dependencies' names and versions. The results are then stored in the `.json` structure inside the `dependencies.json` file.

Further Plans

Regardless the fact that the software is fully operable in the way it was formerly intended, some of its current functionalities are achieved in either performance or ideologically nonoptimal way. Therefore, possible future enhancements and refactorizations will be discussed ahead.

The first and the biggest current problem of the solution is its partial dependence on PHP and Composer package manager in the process of the `Composer` folder generating. One of the possible solution for this problem would be implementation of the process via the NuGet repository instead of using Composer. Another way of achieving the transfer from the PHP world into the .NET environment would be compiling the whole Composer software into .NET using the Peachpie compiler.

Furthermore, the cache generating system is based on the PHP as well, using the Symfony's `Kernel` class to do the job. When there is always a way to simply compile the PHP script into .NET with the Peachpie compiler, Peachpie also provides a way for the PHP cache generating in general. However, this mechanism is not fully usable with Symfony projects and thus will require additional work, possibly from both our and their sides in order for it to be used instead of the current Cache Generating Tool.

Another much more design oriented plan is to separate the Twig-Razor interoperability features into their own NuGet package. This would prove beneficial mostly to those, that want to use only the possibility to embed a Twig template within their Razor ones. In addition to, the former purpose of the `PeachPied.Symfony.AspNetCore` nuget was only to provide an API and features regarding the Symfony middleware in general, hence it would be conceptually correct to have those two functionalities separated as well.

Finally, nuget packages are currently stored locally and thus require each user to compile them manually prior to their usage. Even though, the `Additional-Tools` folder provides the `rebuild-nugets.ps1` script that automatizes this process, it is still far from optimal solution. Moving the precompiled libraries into a remote repository, possibly providing means for additional custom library compilation based on the Autocompile Vendor Tool directly on the server would be definitely a step into the right direction.

User's Guide

Bare bones Symfony application

Using the tools provided, deployment of the Symfony application on the .NET environment was simplified into the following steps that are further divided into two parts, the Preparation phase and the Deployment phase.

The Preparation phase Prior to the actual deploy, the user must first ensure that the NuGet repository is accessible and the Symfony project is prepared for the compilation. At the moment, there is no remotely accessible repository with precompiled Symfony Components. However, the `Peachpied-Symfony/empty-page` project contains Symfony version 4.2.3 project with all the currently supported libraries and all the `.msbuildproj` configuration ready. Furthermore, the `Additional-Tools` folder provides the `rebuild-nugets.ps1` script that performs the whole NuGet repository generation.

Additionally, each Symfony project must contain both the `vendor/Composer` and the `var/cache` folders. When migrating an existing project from Symfony, those two directories will be probably present as they are required for the project to be executable. Nevertheless, there are also scenarios where these two files might be missing. For such cases, the Composer Generator Tool is provided, handling their generation in a single task.

The Deployment phase With the local NuGet repository ready and all the required folders on place, the user can now proceed to the actual project compilation and execution. First, the user must define the `.msbuildproj` configuration file for the Symfony project. Aside from the standard user defined values, the project's SDK has to be set to the requested version of the `Peachpie.NET.Sdk`. Moreover, package references have to be filled in order to take the advantage of Symfony Components precompilation. For this reason, package with id `PeachPied.Symfony.Skeleton` has been created, that groups together all the package references of the Symfony Skeleton project, and thus leaving the users only with the need to specify any additional libraries that were included during project's development. Finally, the existing `.msbuildproj` file corresponding to the Symfony project has to be referenced from the ASP.NET Core application.

For the actual usage of the compiled Symfony project, the `UseSymfony` middleware was defined, as a part of the `Peachpied.Symfony.AspNetCore` nuget's API. This way, all the necessary configuration will be provided only with the user supplementing the relative path to the Symfony project.

Application using Symfony and both Twig and Razor template engines

Usage of the Twig-Razor interoperability features will be demonstrated on the `Peachpied-Symfony-Twig-Razor` application provided.

This example shows a possible way of splitting front-end of an application into two distinct parts, one build with Razor of ASP.NET Core, and second using Twig of the Symfony framework. Whether it might be dues to global transfer from PHP to .NET, because of submerging of existing Symfony project into the existing .NET one, or just to reuse existing Razor and Twig templates within your project, in all of the cases, these features can save both time and money.

The example application consists of five pages. The index page is defined within the Symfony project's `GeneralController` class and returns static `.html` file that is used as a signpost of the application. There are four locations being exposed inside the index, corresponding to the four remaining pages of the application. Two of them match Razor Pages defined within ASP.NET Core part of the application, other two are defined as a routes recognized by the Symfony project and target selected actions inside the `GeneralController` class. The `Razor Page` and the `Twig Page` present a certain scenario of pages displaying content retrieved from the server, and for each element within the retrieved content, providing a small calendar that is embedded using either partial View or including additional Twig template. The remaining two pages take the advantage of the Razor-Twig interoperability features and embeds Twig calendars within Razor Page and vice versa.

Twig rendering Rendering of a Twig template from the Razor context is provided as seamlessly as possible. The only step the user has to perform is to reference the `PeachPied.Symfony.AspNetCore.Templating.TwigRenderer` class from the Razor template. This will expose three functions. The `DataToPhp` support function, that handles conversion of the .NET data into the PHP understandable form. The `MergePhpArrays` support function that can be further used to merge all the converted data into a single unit. And finally, the `RenderTwig` function, that handles rendering of provided template into a `string`.

Razor rendering In order to enable Razor rendering within Twig templates in the Peachpied Symfony project, the user must add the `RenderRazorService`. This is done standardly inside the `ConfigureServices` method in the server's `Startup` class. The service can be registered as `transient`, `scoped` or `singleton`. It is up to the user to select a lifetime that suits his usage of the functionality the best.

Furthermore, special wrapper around the `Environment` class in the `Twig` namespace, called the `RSEnvironment` is provided and can be found in the same namespace. It can be used exactly as the `Environment` class and its only difference lays in additional `render_razor` function that is accessible within Twig templates rendered with it.

After that, the user can render a Razor template into a `string` by calling the `render_razor` function and providing it with a relative path to the template and the data that are to be passed to the template's `ViewBag`. The data provided will

be automatically converted into the .NET environment comprehensible formate and thus are expected to be supplied in PHP's native form.

Application using Twig and Razor template engines without Symfony

Usage of the Twig within Razor embedding without any actual Symfony project will be demonstrated on the `Twig-Razor-Only` application provided.

This example illustrates how easily can be the Twig rendering feature used from plain ASP.NET Core application, without any unnecessary bonds to a Symfony project. This application uses the `Razor Page` and the `Twig In Razor Page` from the previous application, without any further modifications, revealing this feature's actual independence on Symfony. All the Twig templates are stored inside the `Templates` folder and referenced on the `RenderTwig` function invocation just as they would if they were stored inside a Symfony project. Furthermore, previous example's index was moved into ASP.NET Core and modified in order to satisfy this application's needs.