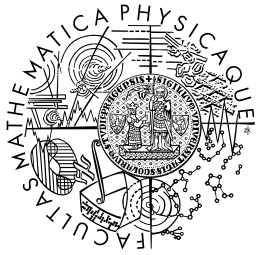

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Horký

Interoperability of compiled PHP framework with .NET environment and package management

Department of Software Engineering

Supervisor of the bachelor thesis:	Ing. Mgr. Robert Husák
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Foremost, I would like to thank my supervisor Ing. Mgr. Robert Husák for all his time and much advice in the course of this thesis. I want to offer thanks to Markéta Sauerová for hours of grammar and syntax consulting. I am also thankful to the whole Peachpie dev team, especially RNDr. Jakub Míšek, for their support and tips. Last but not least, I would like to express gratitude to my parents and family for their love and support.

Title: Interoperability of compiled PHP framework with .NET environment and package management

Author: Filip Horký

Department: Department of Software Engineering

Supervisor: Ing. Mgr. Robert Husák, Department of Software Engineering

Abstract: PHP language has been dominating the web development industry for a long time now. Each major PHP framework brings together a large community and a solid codebase, providing features and tools that make the PHP web application development even simpler. Joining this vast world with .NET is an aim of Peachpie, PHP compiler to .NET. This provides means to use Symfony framework in .NET, yielding advantages for both the PHP developers striving for more security and the .NET programmers lacking third-party content. Doing so with plain Peachpie compiler, however, brings problem of actual usability of the features Symfony provides. Each project would require to recompile Symfony Components again and thus result in both time overhead and tedious manual configuration. This thesis shows it is possible to improve this process and enable intuitive usage of Symfony framework in the .NET environment. It denotes utilities for semi-automatic compilation of the Symfony Components into the NuGet packages and migration of simple Symfony applications to .NET. Furthermore, it provides tools for additional interoperability between Twig and Razor template engines, and a set of examples of its actual usage.

Keywords: C# PHP ASP.NET PeachPie Symfony

Contents

Introduction

Speaking of programming languages, the present world of web development is divided into several nearly disjoint communities. Each of these communities focuses primarily on different aspect of development, for instance security, reliability or simplicity. However, the emphasized aspect of development is merely a consequence of the actual programming language selected for the implementation. Speaking of programming languages, we implicitly mean the server-side programming languages as the thesis is focused mainly on them.

When it comes down to numbers, PHP: Hypertext Preprocessor (PHP) is the ultimate winner with the percentual usage between the websites surveyed of 60 to 70 %. The second place holds ASP.NET with 10% - 20% usage. The third and fourth places are occupied by Java and Ruby, both situated around the border of 5 % of usage [1, 2].

Previous statistics show that if we were to sum the websites implemented in PHP and .NET, we would end up with almost 80 % of all surveyed, and presumably 8 out of 10 all existing, web applications. Moreover, unifying the two technologies would benefit both the communities. Where the PHP community could make use of security and manageability advantages of the pre-compiled code, the .NET developers will be given means to use dynamic features of the PHP language and a vast of community libraries. This is the ultimate goal of Peachpie, a modern PHP to .NET compiler, that is currently in development.

Many web oriented PHP frameworks have been developed by utilizing built-in functionalities of PHP, such as response and request handling, and providing them in a more abstract way. This results in making the process of web application deployment faster and thus less expensive. Due to a lack of any reliable statistics on the PHP web frameworks' usage, custom survey on the topic was performed. Considered parameters were Stack Overflow Trends, Google Trends, GitHub stars, Twitter followers and Facebook likes. This gives us quite accurate information about the popularity of selected frameworks compared to each other, which should correspond to their actual usage. Gathered data are displayed in the Figure 1 and apply to 15/04/2020. When evaluated, Wordpress is an absolute winner, followed by Laravel and Shopify that share the second place. Third place holds Symfony and after that comes the rest of compared frameworks such as CodeIgniter, PrestaShop, Zend or CakePHP.

Unfortunately, every web framework wraps PHP in its own unique way and provides special additional features. Some provide a basic functionality in a set of default scripts and a vast library of downloadable components with additional functionalities. Others deliver almost a working solution out of the box and leave the user only with the need to configure content and possibility to install whole features in a form of modules, such frameworks are usually called Content Management Systems or simply CMS. Therefore, each framework requires special treatment during its compilation to .NET, making their automatic compilation impossible. The Peachpie developers have successfully compiled Wordpress framework. Shopify is not open-source and thus its source code is not simply accessible. Laravel is open-source and the same applies to Symfony. Even though Laravel is in popularity superior to Symfony, Symfony is almost twice as old and its packages are produced mostly by teams recognized by the Symfony developers

themselves, for example FOS or KnpLabs can be named. Its long lasting stability and guaranteed quality of its packages has proved Symfony to be the perfect adept for compilation.

This work therefore examines the possibilities of compiling the Symfony framework into the .NET platform. However, the Peachpie compiler provides just the tools for a single PHP project into .NET compilation. This approach is unfortunate for the Symfony framework as its philosophy lays in minimal start-up project, where every secondary functionality is provided via additional libraries. Therefore, for the purpose of saving user's compilation and start-up time, those libraries are to be pre-compiled and supplied to projects in the quantity that they require, rather than compiled with the project and all of its sources for each project over and over again. For this task, the NuGet package manager has been selected as it shares the same ideology with Composer, a package manager that is tightly coupled with the Symfony framework. The ideology mentioned focuses mainly on package and dependency management at the project level. Besides, its wide spread of usage, Visual Studio integration ability and fairly easy usability were also taken into consideration.

	Stack Overflow Trends (i)	Google Trends (ii)	GitHub stars	Twitter followers	Facebook likes
Laravel	1.54%	26	58,650	113K	45,533
Wordpress	0.91%	100	13,690	633K	1,223,012
PrestaShop	0.02%	5	4,558	36.5K	115,352
Symfony	0.21%	3	23,062	36.9K	11,508
Shopify	0.05%	50	3,242 (iii)	271K	3,560,559
CodeIgniter	0.17%	3	18,036	23.3K	12,332 (iv)
CakePHP	0.04%	1	8,131	17.8K	8,969
Yii2	0.05%	1	13,328	13.9K	1,131
Zend	0.01%	1	5,714	47.1K (v)	50,460 (v)

(i).....percentual usage of tags across whole website in given month
(ii)....number of searches in given month, where 10 means the term was searched twice as often as 5
(iii)...repository of Shopify's product component library
(iv)...unofficial page
(v)....whole company, not just the framework

Figure 1: PHP frameworks popularity comparison

Thesis overview

The first chapter introduces the PHP programming language along with the tools like the Composer package manager, the Symfony framework and the Twig template engine. The second chapter provides general overview of the C# programming language and tools associated with it: the NuGet package manager, ASP.NET Core and the Razor template engine. Chapter 3 addresses the problematique of compiling one language to another in general and introduces Peachpie, a PHP to .NET compiler. The problem the thesis aims to resolve along with proposed solution and possible alternatives is introduced in Chapter 4. The fifth chapter focuses on the Symfony Components compilation and the tools that assist

the process. The way Symfony projects are compiled and deployed is discussed in Chapter 6. Chapter 7 describes actual usage of the toolkit provided with two exemplar applications.

1. PHP language

The first chapter will provide an introduction to the PHP programming language. It will cover its origin, development, process of execution and all of the language features that will be further built atop. After that, configuration and usage of Composer will be explained, as its knowledge simplifies the understanding of the Symfony framework, which will be discussed in the following part. The chapter ends with the explanation of the TWIG engine's syntax and usage.

1.1 Language introduction

PHP is a general-purpose scripting language. It originated as a collection of the Common Gateway Interface (CGI) scripts, written in PERL by Rasmus Lerdorf [3]. A CGI scipt is a program executed by a server upon a request, whose output is then returned to a client [4]. It is essentially what makes server more than a collection of static resources [4]. Soon after their release, these PERL scripts were rewritten in the C programming language to enhance their performance [3]. Since then, the underlying parser has been rewritten several times and many people were involved in implementing additional features and support libraries for this language [5].

PHP has been developed solely by individuals joining their effort. It has resulted in an organic growth of the language in a community of developers. On the one hand, environment in which the language grew caused many useful features and functions to be included in its core [5]. On the other hand, lack of supervision and pre-planning has produced significant inconsistency in function names and in order of functional parameters [5]. Furthermore, for over 20 years, no formal language specification defining correct language syntax existed [6]. This role was substituted by the PHP's engine and the "If it works, then it's fine" philosophy [6]. This task was addressed in 2014 and the resulting document is being further extended since by both the PHP community and the PHP Group [6, 7]. The PHP group is a comunity of developers that maintain the PHP language repository on github and the `php.net` web portal with all the necessary information about the language, from news about releases to actual language documentation.

Versioning of the language might prove to be quite confusing as the official name "PHP" language holds since version 3.0 [5]. Version 1.0 was more of a set of CGI tools than an actual standalone language and version 6.0 was left abandoned due to insufficient number of contributors considering the amount of work that promised unicode support brought [5, 8]. The PHP versioning officially uses the [major].[minor].[release] scheme, but as the releases of the same major and minor version differ mostly in the number of bugs fixed, only the [major].[minor] scheme is generally used. Currently, PHP 7.4 is the latest stable released version, therefore, when speaking of PHP language without explicit version number mentioned, PHP version 7.4 is meant.

Execution PHP is an interpreted scripting programming language. Although both terms can be defined in several different ways, for the purpose of this text, we

understand these terms as that the language was designed for a special-runtime environment without the need to be compiled. the PHP scripts are executed by the Zend Engine, an open-source set of components than provide the PHP language with tools for its processing and execution [9, 3]. It was written in C programming language by Zeev Suraski and Andi Gutmans for PHP version 4, after their previous collaboration with Rasmus Lerdorf on the PHP version 3 [5]. Since then, 2 new versions of Zend Engine has been introduced, each time bringing new major version of the PHP language [5, 10].

The Zend Engine Undoubtedly, the most important component of the Zend Engine is the Zend Virtual Machine (VM), which is responsible for interpretation of PHP scripts [9]. The first step the Zend VM performs when executing a PHP script is a lexical analysis [11]. There it converts human-readable code to tokens for further procession. The parser then takes the stream of tokens and generates an intermediate code [11]. Code optimizations can be performed during this stage, including simple function calls with literal arguments resolving, constant mathematical expressions folding or unreachable code removing [11]. This part of PHP script processing is called the compilation phase, preceding the execution phase [11]. After that, intermediate code in a form of an ordered array of opcodes is passed to the executor that steps through the array and executes each instruction in turn [11].

One of the main features of the Zend Engine is its extensibility [11]. The engine supports modifications either by using the Zend extension Application Programming Interface (API) or through alterable function pointers [11]. For example, both `zend_compile` and `zend_execute` functions that handle script processing phases are implemented internally as function pointers, making it possible for further custom overloading of these functions at runtime [11].

The process of the PHP application's execution is also illustrated in the following Figure.

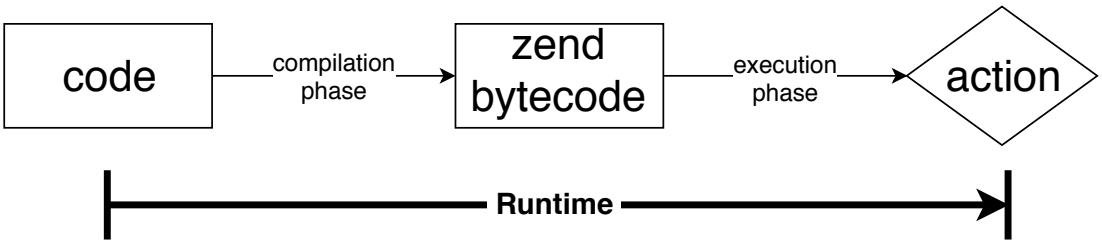

Figure 1.1: PHP's execution

1.2 Language features

Object Oriented Programming Since the language version 5, Object Oriented Programming (OOP) is possible [5]. This enables developers to introduce another layer of abstraction above the PHP code, possibly bringing higher level of organization and reducing maintenance burden. PHP supports all expected constructs of the OOP such as Classes, Interfaces, Constructors and Inheritance

[5]. On a class instantiation, the `object` data type is created [5]. Every object is completely independent, holding its own properties and defined solely by a class it was instantiated from [5].

Type System The PHP programming language is dynamically typed. This means that the language implicitly performs conversions between two types from different families, such as numbers and strings. This approach provides the language with high flexibility and agility but does so at a cost of increased complexity and reduced safety. For instance, a developer with knowledge of how does inner conversion work is able to pass `integer 42` to a function expecting the `string` data type knowing, that the implicit conversion between the types will result in `string 42`. On the other hand, an inexperienced developer might succeed in comparing given variable with the equality operator `==` to `integer 42` without knowing that the variable actually holds `string 42`.

To decrease the required level of understanding of inner conversions and increase type safety, PHP provides several features for easier type manipulations and actively works on the development of new ones. One of the most widely used amongst these features is the identity operator `===`, which extends the standard behaviour of equality operator with additional requirement for type equality of compared variables. Another much more recently added features are type hints, also called type declarations, and strict type checking [5]. These provide the means to specify both the expected types of function's parameters and its return type [5]. In case it is possible to perform an implicit conversion between the type provided and the type expected, the types declared in the function signature are by default for the purpose of backwards compatibility ignored [5]. However, strict type checking feature presents a way to force the PHP interpreter to perform this additional type checking, enabling it to throw runtime errors on type mismatch, even when the implicit conversion is defined [5].

Because of the dynamic nature of the PHP data types, generic type relations are possible without any special constructs. However, this does only provide a way for a basic usage of generics in a form of what is in other languages called generic collections and generic function declarations. It does not allow for any further type relationships, such as bounding of generic types, to be declared and checked.

A special place between the PHP data structures is held by arrays. Internally, arrays are represented as ordered maps [5]. When provided, arrays use custom string or integer values as keys for indexing, basic 0-based keys are used otherwise. Arrays with custom keys are called associative arrays. It is also notable that PHP does not throw an exception when `float`, `boolean` or `null` values are used as keys of an array [5]. Instead, as strings and integers are the only supported key types, implicit conversions between the types are performed [5]. On top of that, string keys representing valid integer numbers are implicitly converted to integer keys [5].

Variables When a variable is declared by a standard syntactic mean of $ sign followed by a case sensitive name of the variable, PHP disallows any explicit type definitions. Variable's type is determined by the context in which it is used, making it possible to seamlessly switch between types with a simple value

assignment [5]. Moreover, PHP itself does not require any formal declaration of variables [5]. The variable is automatically created the first time it is used in a scope and ceases to exist when it is set to null or the `unset` method with this variable is called [5].

Request and Response As was mentioned before, PHP was designed to run on servers as a form of abstraction over the typical request-response based conversation between a client and a server. To abstract developers from the actual HTTP request and to relieve them of the need for repetitive request parsing, several superglobal variables were built-in [5]. As the name suggests, these variables are always accessible, regardless the current scope. `$_SERVER, $_GET, $_POST` and `$_REQUEST` are all members of this group of variables. All of them contain parsed and ready to use data either about the request itself, such as uri or headers, or about the data provided by the request, like submitted form fields or URL encoded parameters [5].

Futhermore, PHP provides means to easily create the HTTP response in just a few lines, taking care of the actual HTTP response creation itself. For instance, the `header` function accepts a string as a parameter and sets it as one of the returned response's headers [5]. Additionally, the most profound PHP construct `echo` is also used in this context as it takes its parameters and appends them to the response's body [5].

1.3 Composer

Composer is a tool for managing dependencies in PHP. Composer works with libraries, sometimes referred to as packages or projects, on a per-project basis. This is what differs the tool from traditional Linux package managers, such as APT, as they usually install packages globally for computer's operating system, making it more similar to NPM or YARN package managers [12]. These libraries Composer works with are `.zip` archives of `composer.json` configuration file, code, optional `composer.lock` file and any other code-related resources [12].

When Composer executes a command, in most cases, either input or output of the command is a `composer.json` configuration file or its update. This file provides means to specify basic information about the package like its name, description, dependencies or type, which can be of project, library, metapackage or composer-plugin [12]. It can be also used to specify additional package repositories and to define scripts as aliases of command-line executables whose execution can be linked to Composer's commands [12]. While some commands, like `install` or `update`, do require the `composer.json` file's existence prior to their execution, as their functionality lays in loading current package's configuration and acting upon it, other commands, such as `require` or `create-project`, only settle for their arguments and the `composer.json` file's creation or modification is a result of their work [12].

Third-party libraries installation is done via the `require` command suffixed with library's name and an optional version number [12]. When resolving a package, one possible outcome is Composer finding a `.zip` archive of this library, in which case, its content is unzipped to a `vendor/[libraryName]` folder [12].

Sometimes, a metapackage is found instead. These are empty packages containing just a `composer.json` file with basic information and a list of dependencies and `readme.md`, because they are usually hosted on version control services [12]. In this case, new instance of Composer is created for this `composer.json` configuration file and installation of listed dependencies is triggered. Both ways result in one or more added libraries to the `vendor` directory. Every package installation or update also results in either update or creation of a `composer.lock` file. Where `composer.json` file holds a list of packages and basic information about current project, the `composer.lock` file notes actual version of each package resolved by the Composer and can be therefore used to ensure that the same set of dependencies is installed in all the instances of the project.

1.4 Symfony framework

Symfony is a PHP framework for web application development. It is built on top of the Symfony Components, a set of reusable PHP libraries [13]. Each library provides certain functionalities and many of them are completely independent, making it possible to use them without the need for using the Symfony framework itself [13]. This universality gave birth to many new web frameworks such as Laravel or Drupal version 8, which are built atop the Symfony Components as well [13].

The Symfony framework evolves a lot each time a major version is released but its philosophy remains. It is to give developers full control over the project's configuration, from the directory structure to a set of used libraries [13]. When the work on the thesis began, Symfony 4.2 was the latest stable version of the framework released, for this reason, further description would primarily focus on this version.

Installation Bare bones Symfony application is created via Composer, more precisely the `create-project` command with either the `symfony/skeleton` or the `Symfony/website-skeleton` argument [13]. On execution, Composer downloads a predefined set of core libraries that handle base framework functionality [13]. After that, everything from routing set-up to additional libraries downloading with the `composer require` command is up to developer.

Request and Response Symfony's way of request-response conversation handling is nothing more than an object-oriented abstraction of the PHP's one [13]. For this purpose, the `Request` and the `Response` classes, with specialized functions for request and response manipulation were introduced [13]. These two classes are part of a Symfony Component called `symfony/http-foundation`, which is included by Composer on application's skeleton installation [13].

Front Controller Traditional small PHP applications usually follow a simple pattern, where each site's page is represented by a physical file, for example `index.php`. As the size of an application increases, this approach brings an overhead in the form of code repetition, complicated route changing and overall decreased flexibility. For this purpose, larger application may define single PHP script

called Front Controller, that handles every incoming request and calls another, more specialized controllers that provide actual response [11].

The Symfony framework is no exception to this approach. However, instead of specialized controller calling, everything Symfony's Front Controller does is booting Symfony and passing information about the request [13]. This is done by executing the `src\Kernel.php` script [13]. Afterwards, `Kernel.php` uses the Routing Component to interpret the request and to resolve the appropriate controller to be called [13]. Finally, based on the request's method and parameters, selected function from the controller is executed, resulting in the Response object, that is converted by the Symfony framework to the HTTP response and sent back to the client [13].

1.5 Twig

TWIG is a PHP library used to combine template files with data model to produce HTML documents. In other words, it is a template engine. Its syntax consists of HTML extended with the TWIG code and three special delimiters.

Output delimiter The first one is `{{...}}`, used for outputting results of expressions [14]. These can be either variable names or function calls [14]. TWIG has a small set of predefined functions [14]. Some have straightforward functionality and self-explaining name such as `dump` and `max`, while the other are much more complex, such as `include`, resulting in included TWIG template loading, generating and embedding the result in the current template [14]. Every expression can be suffixed with an arbitrary number of the `|[filterName]` expressions to further process the result before its outputting [14]. TWIG comes with approximately 40 filters such as `upper`, `split` and `replace` [14]. The predefined filters and functions usually cover the basic usage of the engine. However, extending built-in functions and filters is also possible by writing a PHP script extending the `AbstractExtension` class and defining the `getFilters` or the `getFunctions` method, which is used to link the filter's or the function's implementation method to its alias through which it can be later used [14].

Other delimiters The second delimiter is `{%...%}`, used for executing statements and controlling the flow of a program [14]. Supported structures are for instance `{% for - in - %}...{% endfor %}`, `{% if - %}...{% endif %}` or `{% set - = - %}` [14]. Finally, comments are supported via the `{# ... #}` syntax [14].

Template inheritance The most useful and powerful feature of TWIG is template inheritance. This allows for building a template acting as a basis that child templates further extend. Such template usually contain all the common elements of a site - header, footer and a navigation bar. Furthermore, it can define blocks with the `{% block - %}...{% endblock %}` syntax, whose content can be either fully overwritten by child templates or combined with child's content via the `{{ parent() }}` function [14]. Inheritance is enabled for selected templates by specifying a parent template with the `{% extends - %}` construct in

the beginning of each child template [14].

Environment Environment is an object used for storing the configuration with possible extensions, loading and rendering of a template [14]. The lookup and loading of a template is performed by a loader, class implementing the `\Twig\Loader\LoaderInterface` interface, which is provided as a first parameter to the `\Twig\Environment` class constructor [14]. Template is loaded with the `load` method of the Environment class and rendered with the `render` method of the `\Twig\TemplateWrapper` class, whose instance is returned by the `load` method [14]. The `render` method takes either one parameter - an associative array of variable names and values that are to be provided to rendered template, in which case it acts as described before, or two parameters - template name and an array [14]. The second form wraps both the loading and the rendering of the template internally, returning directly a rendered string of the HTML markup [14]. The `templateWrapper` class also provides the self-explanatory method `renderBlock` [14].

A simple example of Twig template providing a header for calendar implemented as a table is provided in the following Listing.

```
<thead>
    <tr>
        <th colspan="7" class="month">
            {{ monthnames[month].full }}
        </th>
    </tr>
    <tr>
        {% for dow in range(0, 6) %}
            {% set day = (dow + firstDay) % 7 %}
            <th class="
                day {% if (day == 6 or day == 0) %}weekend{% endif %}
            ">
                <span title="{{ daynames[day].full }}">
                    {{ daynames[day].short }}
                </span>
            </th>
       {% endfor %}
    </tr>
</thead>
```

Listing 1.1: Twig example

2. C# language

The goal of this chapter is to introduce the C# programming language and the .NET platform. The following part of the chapter will focus on the NuGet package manager as it forms a building block for the upcoming text. Furthermore, ASP.NET Core framework's background will be described, accompanied by an introduction to Razor markup, which is tightly entangled with ASP.NET Core. Both the ASP.NET Core part and the Razor part will also present an overview of their usage with simple code examples.

2.1 Language introduction

C# is a general-purpose programming language under the management of Microsoft Corporation. It is a part of a developer platform called .NET. One of the main features of .NET is language interoperability, giving programmer opportunity to combine multiple languages and libraries in a single project. That is possible because of the same specification that all the .NET components are build atop, called the Common Language Infrastructure (CLI) [15]. The CLI has been standardized by ECMA International in ECMA-335 and approved as ISO/IEC 23271 [15]. This document covers following 4 areas of interest: type system, metadata, the Common Language Specification (CLS) and the Virtual Execution System (VES) [15].

At the time of writing, there are 8 major versions of C# programming language released [16]. When speaking of C# we implicitly mean C# version 8, as the latest officially released version.

Type system The question of types and their definitions is answered by the Common Type System (CTS) [15]. In order to cover type systems of wide range of programming languages, the CTS defines large number of types and operations [15].

Metadata *"Metadata are used as a way to describe and reference the types defined by the CTS. For that purpose, they are stored in a way that is independent of any particular programming language"* [15].

Assemblies With language-unique compilers, each .NET project is compiled to the Common Intermediate Language (CIL), also known as managed code or simply IL [15, 17]. Resulted compiled code is then saved as either a Dynamically Linked Library (DLL) or en Executable file (EXE) [17]. Basically, when the compiled code specifies the application's entry point, the EXE file is generated, the DLL file is generated otherwise [17]. Exception to this behaviour brought .NET Core, which makes it possible to generate the DLL files for both the code with and without an entry point [16]. Generated DLLs and EXEs together are called assemblies [17].

The Common Language Specification The CLS specifies a subset of the CTS and a set of rules that publicly exported aspects of compiled assemblies

shall follow in order to ensure language interoperability [15]. Its aim is to provide developers with the same base types and usage conventions for each framework that adhere the CLS. [15]

Virtual Execution System The VES is responsible for loading and running programs written for the CTS [15]. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding) [15]. On runtime, platform-specific implementation of the VES performs just-in-time (JIT) compilation of the CIL code to the native code [15].

.NET Standard In addition to the CLI, .NET Standard was introduced as a way to establish greater uniformity in the .NET environment [17]. In contrary to the CLI, .NET Standard only specifies a set of the .NET APIs that are intended to be available on all the .NET implementations [16]. Each version of the major .NET implementation, such as .NET Core, .NET Framework or Mono, targets specific versions of .NET Standard [16]. Each application can target a version of a specific .NET implementation or version of .NET Standard, making it possible to run on any .NET implementation which supports that version of the .NET Standard [17].

The process of a C# application's deployment from compilation to execution is summed up in the following Figure.

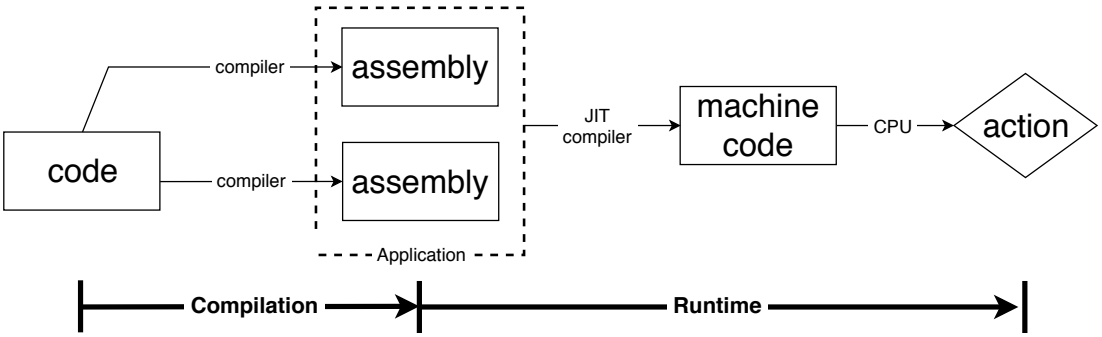

Figure 2.1: C# application's deployment

2.2 Language features

Type system Types in C# are divided into two main categories, value types and reference types. In contrary to the value types, that directly hold their data, the reference types hold only references to objects. There is also a type called `System.Object`. It is the base class of all types in C#, meaning that every type in C# can be converted to an object type (boxing) and vice versa (unboxing) [16]. Because of that, the type system of C# can be label as a Unified Type System.

Slightly aside from this stands the dynamic type. Its features are, in terms of boxing and unboxing, very close to those of the base object type [16]. The

point where the dynamic type differs from the base object type is in the amount of information compiler has about the type, and in the way the type is processed on runtime [16]. An example illustrating both those differences is casting [16]. Dynamic variables do not require any explicit type casting, fully relying on the programmers' knowledge of what he or she is doing [16]. Main purpose of dynamic types is to allow interactions with dynamic objects, such as objects from other programming languages with type systems different than the one of C# [16].

Variables In C#, every variable has a type that determines what values can be stored in the variable [16]. On compilation, the C# compiler validates whether each value stored in a variable is of the appropriate type and whether there are no operations with given variable, that are invalid for its type [16]. In other words, C# is type safe.

There is also requirement for each variable to be definitely assigned before the C# compiler allows for its value to be obtained [16]. Microsoft documentation defines definitely assigned variable as follows: *"At a given location in the executable code of a function member, a variable is said to be definitely assigned if the compiler can prove, by a particular static flow analysis, that the variable has been automatically initialized or has been the target of at least one assignment"* [16].

2.3 NuGet

NuGet is Microsoft-supported mechanism for creating, sharing and consuming code under the .NET environment [16]. For this purpose, NuGet defines a `package`, a single unit of compiled code, related resources and a descriptive manifest containing necessary information about both the package and the code it holds [16]. These packages are generally nothing more than `.zip` archives of mentioned three groups of files with the `.nupkg` extension [16].

In order to allow other developers to download and use a package, when created, a package is usually published to some kind of a sharing service [16]. Depending on desired accessibility level, it can be publicly accessible hosting service, such as `nuget.org`, a cloud, a network or a local file system [16].

Prior to the package usage, one must first ensure that the package and its required version has been installed [16]. Regardless the method used for the package installation, NuGet always merges `packageSources` from `NuGetDefaults .config` file with any other `Nuget.config` files found and retrieves desired package using resulting collection [16]. After that, the content of the `.nupkg` archive is simply copied to a user-specific package folder [16]. Finally, the project files are updated and any missing package dependencies are installed [16].

2.4 ASP.NET Core

ASP.NET Core is a framework focused on building web applications, based on .NET Core [16]. The fundamentals of ASP.NET Core lie in request raising and handling [16]. Every web application defines a pipeline that each component

uses for its requests publishing [16]. This pipeline composes of series of middleware components, ordered regarding the desired order of their request processing [16]. When request is processed, each middleware component performs an asynchronous operation on `HttpContext` and decides, whether to invoke the next middleware component in the pipeline or to terminate the request by short-circuiting the pipeline [16].

Host One Layer above the pipeline stands the `Host` object [16]. Its purpose is to encapsulate an HTTP server implementation, middleware components, logging, dependency injection and configuration by a single object, in order to simplify the application start up and shut down control [16]. ASP.NET Core provides the `WebHost` class with the `CreateDefaultBuilder` method to set up a host with commonly used options, making most of the hard job in developer's stead [16].

An example of the `Host` object configuration and execution is provided by the following Listing.

```
static void Main(string[] args)
{
    var host = WebHost.CreateDefaultBuilder(args)
                      .UseStartup<Startup>()
                      .UseUrls("http://*:5004/")
                      .Build();

    host.Run();
}
```

Listing 2.1: Startup.Configure method

Middleware The middleware component mentioned, sometimes called request delegate, can be either one of the built-in components or a custom one [16]. Each component is added to the pipeline in the `Startup.Configure` method by invoking its `Use`, `Run` or `Map` extension method [16]. By convention, the `Use[middlewareName]` method decides whether to invoke the next middleware component in the pipeline [16]. The `Run[middlewareName]` methods run at the end of the pipeline, terminating the pipeline after its processing [16]. Finally, the `Map[middlewareName]` method is used for branching the pipeline based on the matches of the given request path [16].

Following Listing displays one possible `Startup.Configure` method appearance. Such configuration adds `SessionMiddleware` to enable session state for the application [16]. The `UsePhp` method is an extension of Kestrel provided by the PeachPie software and will be talked about later. Remaining `UseDefaultFiles` and `UseStaticFiles` handle the file mapping and the static file serving [16].

```
public void Configure(IApplicationBuilder app)
{
    app.UseSession();
    app.UsePhp(new PhpRequestOptions(scriptAssemblyName: "pageName"));
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

Listing 2.2: Startup.Configure method

UseMvc method The `UseMvc` extension method adds routing middleware to the request pipeline and configures Mvc as the default handler [16]. ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern [16].

The concept of the Model-View-Controller pattern lays in separation of application logic into three parts kept in different folders and connected together via naming conventions [16]. The model, representing the state of the application, usually holds data, performs user actions and retrieves results of queries [16]. The View is a Razor template, responsible for a user interface building and content presenting [16]. The Controller's job is selecting the view that is to be rendered in the response to the given request [16]. This request can be either initial entry of given route or a user interaction [16]. Additionally, it also provides chosen view with necessary information from the model [16].

Given controller can be mapped to a route by calling the `MapRoute` extension method of the `routes` object with targeted controller, its action (method) and possible suffix, all following the default conventional route syntax [16]. Routing middleware also defines attribute routing, which allows for route definitions to be placed next to the controllers and actions which they are associated with [16].

Services In ASP.NET Core, services are preregistered classes used by the application [16]. The user-defined services are registered using the `ConfigureServices` function of the `Startup` class [16]. New classes can be registered as services to the `IServiceCollection` provided as a parameter to the function in order to be further available along with built-in services [16]. There is also a possibility to directly instantiate a service without the need for previous registration, using `ActivatorUtilities` [16].

By convention, each service is registered by calling the `Add[serviceName]` extension method of `IServiceCollection` [16]. When a service is registered, it is expected that both its configuration and its dependencies will be resolved, resulting in registration of all the required services [16]. This process is called the Dependency Injection (DI) and it is a practise from a design pattern with the same name.

Example of the `Startup.ConfigureServices` method is provided by the following Listing. It demonstrates a minimal configuration for a project using the distributed cache service and the session service.

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds a default in-memory implementation of IDistributedCache.
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.IdleTimeout = TimeSpan.FromMinutes(30);
        options.Cookie.HttpOnly = true;
    });
}
```

Listing 2.3: Startup.ConfigureServices method

2.5 Razor

Razor is a programming syntax used to create dynamic web pages through embedding server-generated code into websites [16]. Its syntax consists of the Razor markup, C# and HTML [16].

HTML rendering is controlled via the @ symbol. This symbol tells Razor to evaluate following C# code, and optionally to render it in the HTML output [16]. Rendered C# code can be either a single space-less expression, called implicit expression, or a spaces-supporting explicit expression [16]. Implicit expressions are created as the @ symbol directly followed by a single space-less C# statement, usually directly returning the desired output [16]. Explicit expressions can consist of several C# statements, combining their outputs into the result [16]. These expressions are built using the `@(...)` pattern, where all the code inside the round brackets is processed as an explicit expression [16]. Non-rendered C# code uses the `@{...}` pattern and can be further transformed into control structures, such as `@for(-;-;-){...}`, `@switch(-){...}` or `@using (-){...}` [16]. This block can also allow parts of its content to be rendered by the @ symbol, the @: directive or the `<text>` tag [16].

In order to follow the DRY (Don't Repeat Yourself) principle, Razor introduced the `Layout` property. By specifying name or a full path of required layout file to be used, we tell the Razor rendering engine to encapsulate rendered content of current file with content of given layout file [16]. This generated content is put in a place of the `RenderBody` method call inside a layout file [16]. Additionally, custom sections can be defined in Razor views and Razor pages using the `@section` directive and rendered in layout files using the `@RenderSection` [16]. This provides a way of additional code organizing [16].

Razor View Razor Views are `.cshtml` files rendered by MVC controllers [16]. Views associated with controllers are grouped in the folders named after them, inside the Views folder at the root of the application [16]. Additionally, view can correspond to a specific action, in which case, it is to be named after it [16]. This arrangement makes application build with Razor views significantly easier to maintain, making it possible to build and update the application's views without necessity to update other parts of the application [16]. It also contributes to the testability and the code reusability.

Process responsible for determining which view file is used, is called view discovery [16]. Actions initiate this process through calling the public `View` method [16]. By default, it searches folders `Views/[ControllerName]` and `Views/Shared` for view with the same name as the action method from which it is called [16]. This behaviour can be changed by specifying a name or a file path of the view [16].

Razor Page Besides the `UseMvc` method, Razor pages require additional services to be registered in `Startup.ConfigureServices` with either the `AddMvc` or the `AddMvcCore` extension method of `IServiceCollection` [16].

The main difference between a Razor view and a Razor page lays in a controller [16]. Razor page handles requests directly, without the need for going through the controller [16]. For Razor pages, new directive `@page` has been introduced to

distinguish in other aspects similar page files from view files [16]. In all Razor pages, `@page` must be the first directive used [16]. All razor pages are located in the `Pages` folder at the application root [16]. This folder is also used as root for associations of URL paths to pages, as they are determined by the `.cshtml` file location on the file system, relative to the `Pages` folder [16].

Pages can be interconnected with the `PageModel` classes using the `@model [pageModelName]` directive [16]. In this case, `PageModel` should be located in the same folder as the page file and its name should be created from the page's name by appending the `.cs` suffix [16].

A simple example of Razor template providing a header for a calendar is in the following Listing.

```
<p class="month">
    @currentMonth - @current.Year
</p>
@for (int i = 0; i < 7; i++) {
    if (i == 0 || i == 6) {
        @:<div class="box_short weekend">
    } else {
        @:<div class="box_short">
    }
        @:@dayNames[i]
    </div>
}
```

Listing 2.4: Razor example

3. PHP to .NET compilation

The third chapter speaks about programming languages and communities around them in general. It provides a possible way of building a bridge between two selected communities and outlines some aspects of its practical implementation. Peachpie, an actual implementation of previously discussed tool is presented.

3.1 Motivation

In the recent days, web development, and software development in general, has evolved by a great bit in both the number of people involved and in the size of code produced. Web pages and applications are no longer just a several thousands of lines of code. For instance Facebook, one of the most profound social networks in the world, has a codebase of over 50 millions of lines [18]. Moreover, it's not unusual for a codebase of software of an average modern high-end car to reach a hundred million lines of code [18].

Natural consequence of such technological growth is that it is no longer possible to develop complex software alone and almost everything requires teaming up. Above the project-related groups of developers stands a community, where knowledge is shared and people learn from each other. While some patterns and practices are applicable in general, with increasing complexity of problems their amount and usefulness decreases. This need for more definite knowledge has caused the community to be rather granulated and the number of breaching criteria is immense. Undoubtedly the most commonly used criteria is the one of programming languages.

A reasonable effort is put into attempts to bring together selected language-level sub-communities. Going through all of the possible methods of achieving this is beyond the scope of the theses, so we will simply pick the one that is further relevant, the compilation method. It is worth mentioning that for this compilation method we require the target language to be an interpreted language. It is not to be misunderstood with method of compiling the second language to the source code of the target language, which would in that case be called the transcompilation method. Simply speaking, our compilation method means that we have a language A compiling into an interpreted language that is later executed. Then we take a language B and compile it with special tool into the same intermediate language as the language A. This allows us to embed parts of code from the language B within our application written in the language A. While this approach does not literally merge the sub-communities of languages A and B, it clearly provides a bridge between the two in a way of sharing features, knowledge and finally the developers of both languages within a single project [19].

For the later usage, we require the target language to be a compiled language with an efficient way to execute compiled code separated into multiple files. As was mentioned before, C# is a compiled language with strictly defined foundation [16]. In addition to, it has quite intuitive Integrated Development Environment (IDE) provided by Microsoft Corporation with many built-in features such as profiler and web designer [16]. Ultimately, its assembly system allows for splitting

of an application code between multiple files and their later efficient combination on runtime. For these reasons it is a perfect target for a compilation. On the other hand, PHP is an interpreted language with a large community of developers, many implemented libraries and vastly popular frameworks. Making it a possible candidate for a compilation.

3.2 Languages comparison

In programming generally, it is fairly unusual for two programming languages to share all, the same philosophy, capabilities and the area of usage. When a problem in a certain language is encountered, solutions in other languages are mostly useless. This can be addressed by a compilation method introduced before. When we encounter a problem in one language, have the solution in another language and ultimately have the means of compiling one language to another, the former solution quickly comes in hand. The crucial spot in this process lays in mentioned means for achieving the compilation. Prior to learning how to operate with the compiler tool, we need to further analyse and compare our candidates for the source and the target language. That is necessary in order to prepare for possible complications that might occur during the actual compilation.

Type System Even though there are significant differences in PHP and C# languages' type systems - one is statically typed, while the other one is dynamically typed, this difference will introduce no noticeable difficulties. PHP's engine uses types internally and those basic types are subset of the CTS [9, 15]. In addition to, PHP has no sense of topology in its type system [5]. PHP types simply exist without any base types or inherited functions [5]. This makes their mapping to the .NET types and into its type hierarchy rather easy.

The only type-related complication lays in PHP's implicit conversion between the types. However, this problem can be solved easily by denoting explicit conversions, implementing special functions that emulate PHP's implicit conversions between the types or by using the base type `Object` [20].

Variables As we already know, PHP does require type-less variable initialization whereas C# requires explicit type definition when a new variable is initialized. Furthermore, PHP needs no variable initialization at all. On the contrary, C# presents a rule for each variable usage called the definitive assignment.

The problem of PHP's type-less initialization can be easily solved using implicit conversions mentioned in preceding paragraph and C#'s base class `Object` [20]. The problem of definitive assignment and PHP's ability to define variables in-place can be addressed on compilation [19]. This is done based on each variable's usage or by an implicit initialization to `null` if needed. This operation will satisfy C#'s variable requirements and will not divert from the behaviour observed in PHP, where uninitialized variables return `null` upon reading.

Execution One of the main differences between C# and PHP lays in their execution. PHP is interpreted language whereas C# is compiled. Although this difference in a deployment process might seem problematic, it is actually in our

favour. If we were attempting to compile C# to PHP, this might not be true because PHP application's execution would require the compiler to transform a C# code to the native PHP code.

On the other hand, in our case, we want to compile a code of an interpreted language to the code of a compiled language. Although converting a PHP code directly to the C# code is also a solution, it is rather poor one as there would be two compilations in the process of deployment of such application, which would bring significant overhead.

The solution to this is .NET, more precisely the CLI. If we can compile a PHP code into the C# code that can be later compiled into the managed code, by the law of transitivity, we have to be able to compile a PHP code into the managed code directly [20]. The comparison of languages' deployment extended by the provided solution is depicted in the Figure 3.1.

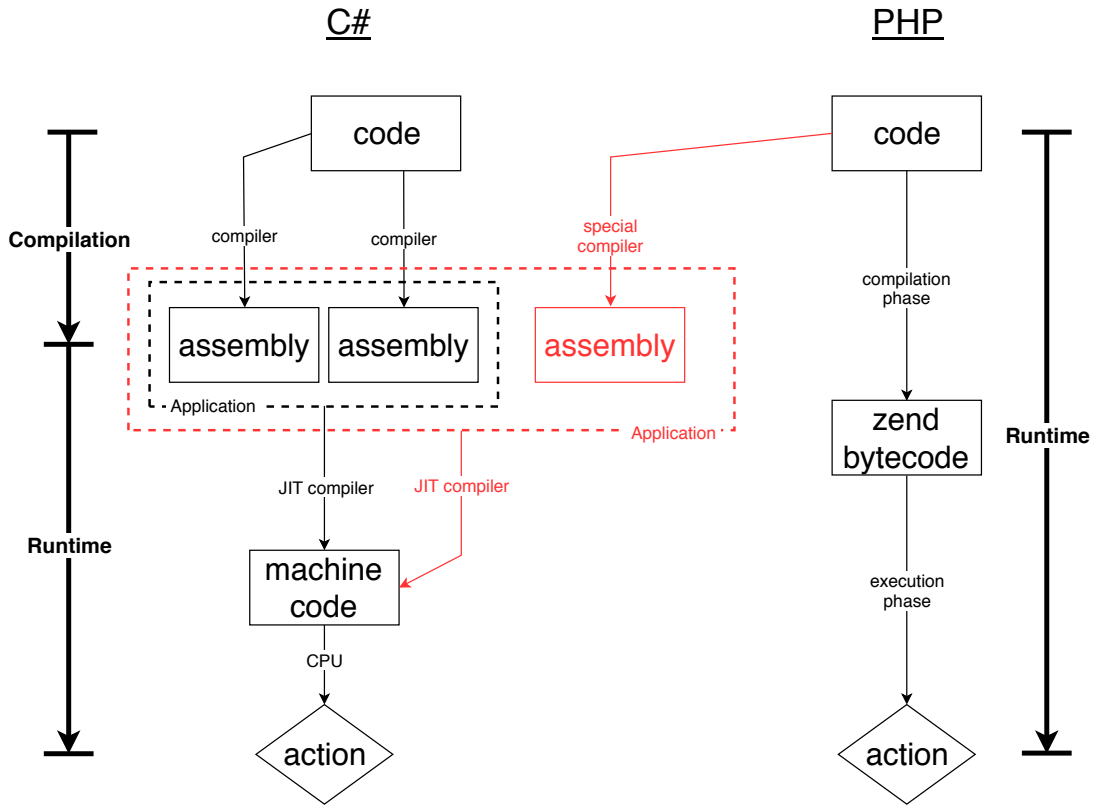

Figure 3.1: PHP to C# deployment comparison

Consequences The first consequence originates from the way PHP is executed. Because everything in PHP is done on runtime, parts of the code may, and sometimes do, contain by a program flow non-accessible invalid code, that would cause PHP's interpreter to throw a runtime exception when executed. It is obvious that such feature cannot be brought to the world of .NET because of the compilation that validates the code in the process. For such code, there is no other way than to either remove it from the source, or exclude the whole file from the application compilation.

The second consequence does not originate from anything mentioned but is more of an architectural nature. The CLR does require the whole `finally` block to be executed and for this purpose forbids statements redirecting the execution flow elsewhere [16]. On the other hand, PHP guarantees only `finally` block entering but not its whole execution, allowing it to contain flow control statement such as `return`, which will result in overwriting the possible `try/catch` block return value [5]. Again, as such behaviour is inconsistent, there is no other way than to either rewrite these statements to an equivalent form without these unsupported constructs, or to exclude them if the former is not possible.

Very similar differences can be observed in iterators as well. While C# does place further restrictions on the usage of the `yield` keyword in combination with `try, catch` and `finally` blocks, PHP does not [16, 5]. This diversity originates again from the language-specific procession of exception handling blocks and its possible solutions are for this reason the same as those in the preceding paragraph.

Last but not least stands the problem of standardization. Because the PHP language does not have any complete formal standard, there is no list of all defined functions, classes and possible parameters. This makes it almost impossible task to create a compiler tool which would be able to process every PHP's built in function, because that would require having cumulative knowledge of all of these. For this reason, it is beyond the capabilities of a few individuals who usually work together on such compiler and so they have to be always prepared for further compiler enhancing and additional features implementing.

3.3 Peachpie

After theoretical analysis of the languages, providing a brief outline of possible complication that might arise during the compilation and presenting some of the possible solutions for each of them, we can finally proceed to the actual compiler tool, Peachpie.

Peachpie is a modern PHP compiler based on Roslyn, a set of open-source compilers and code analysis APIs, also called the .NET compiler platform [19, 16]. Peachpie took heavy inspiration in its predecessor Phalanger, a PHP language compiler for .NET Framework targeting the PHP language version 5 and ASP.NET 2.0 [19, 20]. Peachpie allows for a full compatibility with .NET, which enables the development of hybrid applications, where part of the code is written in C# and part in PHP [19]. Finally, Peachpie brings a way to develop PHP applications directly in the Microsoft Visual Studio IDE [19]. This way, developers can make full use of its built in features such as profiling or debugging.

Development When Phalanger was developed, its main goal was to compile a legacy PHP code into .NET, at which it ultimately succeeded. However, its development slowed down through out the time and finally stopped at the PHP language version 5.6 [19].

Several years later, Microsoft introduced compiler platform called Roslyn [19]. Aside from granting tools to potentially speed up both the compilation and the runtime phase, it also provides a way to help developers directly perform phases of compilation, such as lexical and syntactic analysis [19]. Inspired by the Roslyn

platform, former maintainers of Phalanger revived the concept of a PHP compiler for .NET and Peachpie's development was initiated [19].

The compiler's development continues to the present day and new development versions of software are released several times a month, each time bringing additional functionalities and missing features implementation [21]. At the time of writing, Peachpie is capable of compiling PHP versions 5.4 up to 7.4, and using them within .NET Core SDK 2.1 as well as .NET Core 3.1 [19, 21].

Benefits of Peachpie Comparing and benchmarking speed of Peachpie and PHP is rather complicated task and to achieve certain reliability, it would require to perform several tightly designed tests across multiple platforms and on a wide variety of different set-ups. Although no such work has yet been done, Peachpie has been conceptually compared to Phalanger many times and each time resulted as a definite winner [19]. In addition to, iolevel has published several benchmarks that provide at least provisional comparison of Peachpie and PHP [19]. To come to a conclusion, Peachpie can be, in terms of speed, considered to be comparable to the PHP version 7.2.

Much more promising benefits of using Peachpie are provided interoperability and security [19]. As was mentioned before on several occasions, compiling one language into another does theoretically enable one to use features and libraries written in both of the languages and thus significantly narrow the gap between the two communities. Peachpie does put this theoretical concept into practice and presents full interoperability between PHP and .NET. In addition to, Peachpie introduces a way of eliminating potential security vulnerabilities by compiling the code into .NET [19]. Between provided security improvements can be named the ability to deploy PHP applications without sources or using the security restrictions configured for the process [19].

4. Problem analysis

This chapter describes the idea behind the thesis and introduces specific tools and technologies that will be used on the way to its achieving. Moreover, the chapter provides general overview of selected approach and its possible alternatives, and explains reasons for each choice.

Peachpied Symfony Regarding the name of the project, we decided to hold a convention laid down by the previous project made by the PeachPie developers, where the objective was to compile the Wordpress framework into .NET. The resulting project was named Peachpied Wordpress, therefore, this project was entitled Peachpied Symfony and all custom tools provided use the Peachpied.Symfony prefix in their names.

Symfony project structure As was mentioned before, each Symfony project is made from libraries, generally called Symfony Components. All these libraries reside in the `vendor` directory of the project where they are downloaded by the Composer package manager on the project initialisation. The rest of the project consists mainly of settings, such as routing definition, environmental variables or database credentials configuration files, and user defined content, such as controller classes responsible for handling requests, images, definition of styling and front-end scripts. Aside from this, every Symfony project also contains a set of autoload files which are managed by Composer as well. These are responsible for the proper load-up of all of the libraries in the `vendor` folder and are stored under the `vendor/composer` directory. Finally, each Symfony project generates cache on the first start-up to speed up later processing of requests. By default, these are stored under the `var/cache` directory. High-level overview of the whole Symfony application development is depicted in the following Figure.

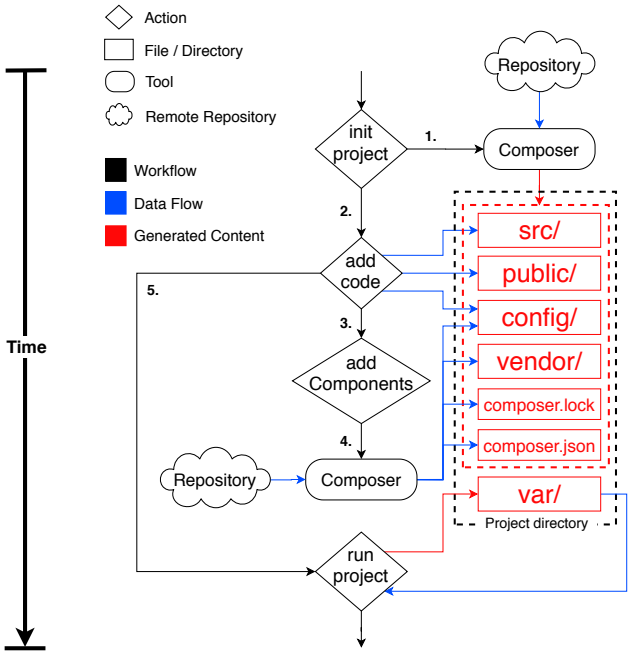

Figure 4.1: Symfony application development process

Current problems The ability to compile Symfony projects into .NET would yield undeniable contributions to the both communities. In terms of plain feasibility, this is what the Peachpie compiler already does, nevertheless, aside from its dependence on using PHP and Composer to generate certain files, there are also other problems that recede this approach from being usable in practise. These will be discussed right ahead.

When using plain Peachpie to compile a Symfony project, the major problems lay in efficiency. Each time a Symfony project is compiled, the whole content of the `vendor` folder is recompiled. For the bare bones project this means over a 15MB of `.php` code sources distributed over 3000+ files. Moreover, we are talking only about the bare bones project with approximately 30 Symfony Components. Actual projects may use hundreds of them making this approach highly ineffective in practice. Finally, due to differences in type systems and execution models of PHP and .NET, compilation errors calling either for some files' exclusion from the compilation or their direct modifications are needed as well.

Aside from using Peachpie in the previously mentioned manner to embed an existing Symfony project into our new .NET Core application, there is also a way of using Peachpie to include only a selected functionality from Symfony. The Twig template engine was selected for demonstration of these possibilities due to its vast popularity in the PHP environment. This can be done by referencing just a small set of Symfony Components in our application and with the use of specialized tools providing an API for the usage of the functionality. This way of usage seems to be more promising as there would definitely be less files to compile and thus less compilation errors, that would otherwise have to be tended to. Nevertheless, these files would require to be compiled over and over as well. Furthermore, there would certainly be an urge to write an API for using the desired functionality from .NET which would require a non-trivial knowledge of both Peachpie and the Symfony Component.

Aim of the project When measured by the complexity of the task in terms of time needed for its manual build configuration or its actual build efficiency, the Peachpie compiler would definitely appreciate a helping hand of more specialized tool to deliver the solution in practically usable form. The aim of this project is to address this need and reduce compilation times of mentioned ways of embedding Symfony within .NET Core applications, while increasing actual usability by providing simple API for its usage. It is expected that an appropriate replacement for the Composer package manager from the .NET world will be used, for which the NuGet package manager was selected. Our goal is to provide Symfony Components as NuGet packages on a Symfony project compilation while ensuring minimal project-specific configuration is required.

4.1 Proposed Solution

Taking its inner structure into account, one can perceive a Symfony project as two completely distinct parts. First, the `vendor` directory containing all of the libraries that the project uses, and second, basically the rest of the project folder. Moreover, for the set of Symfony Components and fixed versions, the `vendor` folder will always look the same. Ultimately, each of these Symfony Components

can be recognized as an independent unit as well, only holding references to other libraries that it requires for its proper functioning via its own `composer.json` file. From this point of view, one can perceive Symfony Components as libraries rather than plug-ins. This perspective is supported by the fact, that the Symfony framework does not perform any automatic functionality loading of imported components. Their actual usage is solely up to the programmer.

The whole solution of the previously mentioned problem is therefore based on these conclusions:

1: User does not manually modify content of the `vendor` directory.

2: The `vendor` directory for the set of Symfony Components with fixed versions will always be identical (excluding the autoloaders part).

3: Symfony Components are self sustained with functionalities depending only on libraries listed in included `composer.json` file.

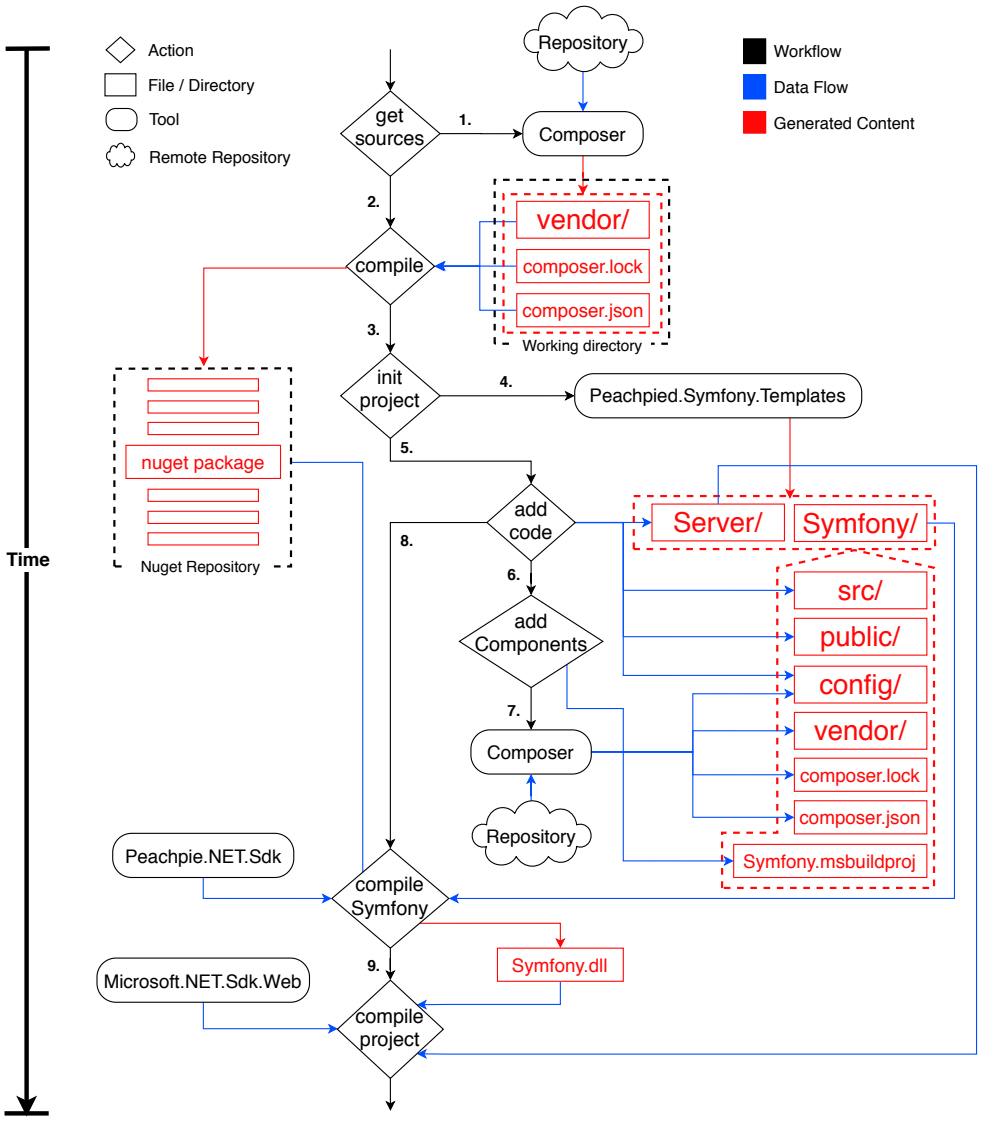

Figure 4.2: Peachied Symfony application development process

Figure 4.2 presents a high-level overview of further discussed assessments and will be accordingly used in the remaining text.

Component compilation Taking the library-like nature of Symfony Components into consideration, we decided that pre-compiling them and providing each one on demand would bring the most desirable outcome while preserving the way they are intended to work. This step will save considerable amount of time by deferring it to the pre-compilation. It also brings the highest flexibility regarding future extensibility because it theoretically enables the usage of these compiled Components from any .NET application and provides a way of including custom NuGet packages to Symfony applications out of the box. Steps `1.` and `2.` of Figure 4.2 cover this part of the process.

The NuGet package manager was selected to handle this task as the most widespread package manager under the .NET environment. This way, each Composer dependency will be simulated by a package reference and the NuGet package manager will handle the rest.

Each Symfony Component will be compiled into the NuGet package and thus it will require its own build configuration file. In order to save both time and effort, we decided to separate common configuration for all Components into several files and simply include them into each Component's configuration. Furthermore, certain information about a Component can be parsed out from provided `composer.lock` file, and can be used to automatically provide additional compilation configuration. Unfortunately, due to state of the art of Peachpie and non-perfect overlapping of PHP and C#, we can expect requirements for further Component-specific manual configuration. These will be grouped into a single file that would each Component take its specific configuration information from. We also have to be prepared to encounter `.php` source files that will require direct edits in order to remove unsupported PHP constructs and features. Of course, these will have to be performed in a way to preserve former functionality.

One drawback of introduced approach is the necessity of significant number of packages to be pre-compiled in order to comply the requirements of even the bare bones Symfony 4.2.3 project, created with the `composer create-project Symfony/Skeleton [projectName] [<version>]` command. Such application is called Symfony Skeleton and consists of 24 Symfony Components.

These Components, along with several other that were required for further presented Twig interoperability, were pre-compiled and included in the project repository for the presentational purposes. In order to achieve high efficiency not only for now but for the future usage as well, we provided partially automatized way of producing custom NuGet packages for other Symfony Components.

Project compilation Step `8.` of the Figure 4.2 corresponds to the Symfony project compilation. By supplying the Symfony project with pre-compiled libraries in a form of NuGet packages of required versions, we reduced the compilation time and removed the need for additional component-specific manual configuration. This also resolved some of the previous dependencies on PHP and Composer, however, it brought up several new questions that had to be addressed in order to move forward. How to handle Composer's autoloading? What about Symfony's caching directory containing `.php` files produced on application start-

up? And how will the Composer's dependencies and the `composer.lock` file be mapped to package references?

Autoloading scripts in Symfony are produced by Composer's `dump-autoload` or `install` command and updated each time new Symfony Component is installed. It is true that their content is determined by the set of used Components and their versions, however, we can see that providing them in a pre-compiled state as NuGet packages as well would prove very challenging. Choosing this approach would require to produce and store NuGet packages for autoloading scripts of all possible combinations of packages and their compatible versions. One, and probably the best, alternative to this approach is using Composer itself to produce these scripts during an early phase of the project's compilation and then simply compile them with the project. Time overhead of this method is negligible, compared to time needed for autoloading scripts' pre-compiling. Efforts were made attempting compilation of Composer or at least a required subset of its functionalities into .NET. However, Composer consists of more than 200 `.php` source files and uses over 10 additional libraries. When added to the number of compilation errors produced on an attempted compilation, this approach has proven to be too demanding for now and was left as one of the key points for the future work. For this reason, Composer itself is used to produce autoloading scripts. With this approach, we present a strict requirement for PHP to be installed on user's computer as Composer depends on it. Regarding the fact that we compile a PHP framework that heavily relies on Composer and PHP itself, we can assume that the user has it already installed, thus we decided to assign this requirement a low importance.

Symfony projects use caching mechanism in order to speed up future request handling. These application cache are by default stored in the `var` directory and produced after the first application start-up. In order for the application to work correctly under .NET, these files need to be compiled as well. For Peachpie however, this means that the application has to be started first, either from PHP or from .NET after previous compilation, and then recompiled again with generated cache files. These files are produced by the Symfony project's core and load user defined configuration shattered among several `.yaml` and `.php` files. An easy and efficient solution to this lays once again in producing these files manually during an early phase of the project's compilation. Sources responsible for cache producing in Symfony can be easily compiled into .NET and then used from the task on compilation. The `.php` configuration files however, yield some challenging issues. A well refined file parsing would have to be implemented in order to cope with this type of configuration files as they cannot be simply executed dynamically the way they are in PHP. Taking the complexity of such feat into account, we decided to defer solution of the problem to the PHP world once again.

On the contrary to previous conclusions, the solution to the final question of how to convert Composer's dependencies to package references might lay already within Peachpie. One of the targets provided by the Peachpie Software Development Kit (SDK) parses the `composer.json` file and converts every dependency into the package reference. Unfortunately, it does so only for packages not marked as development dependencies. In addition to, it does not check whether a NuGet package for listed dependency exist in any of registered repositories. This is prob-

lematic as during compilation, every Symfony Component mentioned somewhere in the compiled Symfony poject's code needs to be referred and thus cannot be considered a development dependency. Moreover, some non-development Components provide only additional CLI (Command Line Interface) functionalities for Composer and PHP, and therefore do not need to be referred nor compiled. Correct way to handle this issue would be to cooperate with the Peachpie developers and adjust current tool to befit our needs. We decided to avoid this way in sake of saving the time such task would consume and postpone it into possible future work.

Writing a custom tool handling this seems as a quick alternative, nevertheless, it would require delivering it before NuGet's `restore` target is executed which means loading it directly from a `.dll`. From the perspective of the UX (User Experience), this approach brings unpleasant complications as well and regarding mentioned ideal solution, it would be more of a temporal workaround that would be later discarded. We decided to settle for a manual package references set-up accompanied by a possibility to use preconfigured templates for projects and additional NuGet packages that group selected packages together.

Project usage After successful project compilation there still remains a question of how to actually use this `.dll` produced by proposed compilation process.

One way to integrate compiled Symfony project into an ASP.NET Core application would be to break it into functional units and integrate each one of them separately. This would certainly require a set of wrapper classes around Symfony's native functionalities that expose them via an API and enable programmers to use them in their own code. This approach provides high flexibility but when mapped to the actual way the Symfony framework works, it seems rather inefficient or even useless. The problem is that the Symfony framework as is provides only the basic routing functionality and the rest of the application logic is either defined by the user or imported via referencing additional Symfony Components. The routing mechanism of the Symfony framework cannot be simply separated from the rest of the application logic as there might exist another custom logic defined in the Symfony application.

An alternative we chose is to provide compiled Symfony project as a ASP.NET Core middleware and let the user put it in the application pipeline wherever he wants. This way, requests that reach the middleware are completely in the hands of Symfony, executing all the custom logic defined there while keeping the means of the user to modify them limited. However, regarding the fact that this middleware executes either Symfony's routing that can be simply overridden with another routing middleware placed before the Symfony's one in the pipeline, or custom functionalities that would have to be executed as they are nevertheless, drawbacks of this approach seem irrelevant. Furthermore, this way supports additional functionalities provided to the Symfony application through Symfony Components out of the box. This removes the need to write separate API for each Symfony Component while still giving an opportunity to create these APIs for interesting functionalities and use them from C#.

Steps `3.` and `4.` abstract the way such applications can be created. Following steps `5.`, `6.` and `7.` cover means of extending provided template applications. Finally, step `9.` depicts compilation of the whole ASP.NET Core application

using the `.dll` resulting from the previously compiled Symfony sub-application. In fact, the final two steps will be done at once in real applications. They are presented separately only for the presentational purposes.

It is is clear now that regarding the number of Symfony Components and continuing development of Peachpie, delivering our tool in a production grade quality would require tremendous amount of time or a greater team of developers. Because that exceeds both our capabilities and the scope of the thesis, we focused primarily on outlining how the tools should work and displaying some possibilities to provide further interoperability. The whole project is therefore to be regarded as a proof of concept.

5. Component compilation

This chapter looks closely into a phase of the problem solving that can be regarded as a preparation step. The way Symfony Components are packed into the NuGet packages and the tools that help to automatize the process are discussed here.

At first, a brief introduction into the MSBuild tool will be provided as it forms a required knowledge for further understanding. After that, we will present more detailed overview of the solution design. Description of various configuration files that accompany the build process will follow, ending the chapter with a section about implemented automatization of the whole Components compilation.

The source code for the NuGet packages of Symfony Components provided with the project can be found under the `Peachpied.Symfony/Components/4.2.3` directory. It was included in the project due to certain modifications that distinguish it from the one acquired by the `composer require "[name]:[version]"` command. More about these changes and their purpose will be discussed later on. For clarification, when speaking of a certain package folder or the `vendor` directory in the following text, those in this particular directory are meant unless stated otherwise. Moreover, custom notation {`* ... *`} will be used in examples on places where the idea behind the expression is of higher importance than the actual implementation.

5.1 MSBuild

When a .NET project is compiled, the MSBuild build system takes the configuration stored in the `.[projectType]proj` file that is associated with the project and performs the compilation [16]. The project type by convention reflects the actual extension of the file and is used in order to increase their understandability [16]. Some of generally used extensions are `.csproj` for C# projects, `.vbproj` for Visual Basic .NET projects, and `.msbuildproj` for projects not tied to any specific language. The `.msbuildproj` file extension is also used by the Peachpie SDK. In addition, `.targets` and `.props` extensions are recognized as reusable project files that can be imported to other project files [16]. Regardless the final extension and usage, all of the `.[projectType]proj` files are XML files that adhere to the MSBuild XML schema [16].

The basic syntax of the MSBuild XML schema uses properties and items as a way of passing information to tasks. Where properties are name-value pairs, items are collections of values and can contain attributes. There are many additional differences between properties and items such as their typical usage, the way they are passed to tasks or the way they are processed. MSBuild reserves some property names as a mean of storing information about the project file [16]. Those are for example `MSBuildThisFileDirectory` or `MSBuildProjectDirectory`. Properties can be referenced by using the `$([propertyName])` notation, items by `@([ItemName])`. Furthermore, new items and properties can be defined by using their name as a child element of either `PropertyGroup` for properties or `ItemGroup` for items.

Aside from that, MSBuild provides a way to modify the compilation process by hooking custom targets into the build order. Target is a set of tasks, functional units providing certain actions, represented by the `Target` element. On the .NET project compilation, a list of default targets is sequentially executed performing each target's actions or skipping them based on provided conditions, ultimately resulting in the project's compilation. MSBuild has a built-in collection of tasks providing basic actions, such as `Delete`, `Copy` or `Message`. In addition to, custom tasks can be registered from assemblies via the `UsingTask` element. We can include custom tasks written as .NET class libraries that will provide us with the required missing functionalities. Such class only needs to either implement the `Microsoft.Build.Framework.ITask` interface or inherit from the `Microsoft.Build.Framework.Task` class and define the `Execute` method returning a `boolean` value. Task's input parameters are supplied through attributes and correspond to the .NET properties in the class definition. Output parameters of the task are created by marking selected .NET properties with the `[Output]` attribute. These can be collected after task's execution with the `Output` subelements which perform mapping of a single .NET property to either MSBuild item or property.

5.2 Solution design

For each library, we need to create a project, more precisely, we need the `.msbuildproj` file that will be supplied to the MSBuild task. Initially, to include the Peachpie compiler we need to set the `Sdk` attribute of the root `Project` element to `Peachpie.NET.Sdk/[version]`. This way, the properties and the targets used by the Peachpie compiler will be imported into the project. The second step is to set the `GeneratePackageOnBuild` property to true in order for MSBuild to output a NuGet package. Then we need to specify files that are to be compiled, excluded and copied to the output package in a form of static content. For this purpose, items `Compile` and `Content`, and their attributes `Include`, `Exclude` and `Remove` can be used. When MSBuild copies static files to the output, by default, it preserves the structure the files are located in. As the NuGet packages will be used to supply functionality of the former Symfony Components, it is preferable to place all the `.msbuildproj` files for each project to parent directory of the `vendor` folder. This way the path to static files will remain unchanged, that is, beginning with `vendor`. The last step is to specify package's dependencies. This can be done with either `ProjectReference` for dependencies on another projects, or with `PackageReference` for dependencies on NuGet packages [16].

On the first look, this seems like a great amount of repetitive code. For the sake of the DRY principle, we will make use of the `.props` and `.targets` files and try to generalise as much of the configuration as possible. Either can be used, as MSBuild does not really pay much attention to the actual extension of a file, but to follow the conventional usage, we will store reusable properties in `.props` files and define reusable targets in a `.targets` file.

5.3 Props files

As was mentioned before, we have to set the `GeneratePackageOnBuild` property for every project in order to generate NuGet packages. Aside from this, there are several other significant and required properties that we need to set-up as well to produce the correct output. These properties specify the output type of the project, target framework, project description, name of both the output `.dll` and the final package, version of the package and necessary routes where to output the NuGet package. We also have to set up a path for intermediate output and a path to the output directory. All of these properties will be fairly similar for all of the packages, varying only in the names and parts of paths.

In order to fully isolate previously mentioned properties into a separate file, we denoted a naming convention for the `.msbuildproj` files. This way, we was able to use the project name to derive a name of the output package and a path to the actual library folder. For that, we used MSBuild's reserved property `MSBuildProjectName` that holds the file name of the project without the extension.

Now, we can proceed to configuring the two properties that will differ among the projects, libraries' names and their sources' locations. Hence, we created two properties, `LibraryName` and `LibraryPath`. The naming convention for the Symfony Components follows the rule `[providerName]\[componentName]`. For this reason, and also to comply the NuGet package naming rules, we defined a property that simply replaces each backslash delimiter on the route from the `vendor` folder to the library folder with dots. Together with the properties holding fixed values like `OutputType` or `TargetFramework` we have so far a fully automatable process. To detach background properties from the ones used for the project configuration, and compilation items configuration from all of them, we created 3 `.props` configuration files.

Directory.Build.props This file's presence is optional and should be located at the root folder of the project. Its content is imported automatically in an early stage of the compilation, making it capable of overwriting properties defined in most of the build logic [16]. The backslash to dot conversion logic will be defined there, introducing the `ProjectNameLocation` property as the result. Additionally, values for `BaseIntermediateOutputPath` and `BaseOutputPath` need to be set there as well, as they are later used for calculating routes used for storing NuGet related data [16]. Because we compile all the Symfony Components from the same working directory, we have to separate their output paths in order to avoid file overwriting. This is where we use the `ProjectNameLocation` property for the first time and it is also the reason this property must be defined here. The actual code that handles that can be seen in the subsequent Listing.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <ProjectNameLocation>
            $([MSBuild]::EnsureTrailingSlash('\$([System.String]::Copy('\
                $(MSBuildProjectName)').Replace('.','\'))'))
        </ProjectNameLocation>
        <BaseOutputPath>
            .\bin\$(Configuration)\$(ProjectNameLocation)
        </BaseOutputPath>
```

```
        <BaseIntermediateOutputPath>
            .\obj\$(Configuration)\$(ProjectNameLocation)
        </BaseIntermediateOutputPath>
    </PropertyGroup>

</Project>
```

Listing 5.1: Directory.Build.props

propertiesConfig.props Before proceeding to the build properties file, we need to mention that by default, the PeachPie compiler provides a feature of `composer.json` file parsing and producing package references as an output. Because this feature would collide with our own package references retrieval mechanism, we need to provide the PeachPie compiler with an invalid route to `composer.json`. This is done by setting the `ComposerJsonPath` property to any invalid value. For readability purposes, we set this property to `Ignore`. Now, with the use of defined `ProjectNameLocation` property, the build properties `.props` file will be organized as follows.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <OutputType>Library</OutputType>
        <TargetFramework>netstandard2.0</TargetFramework>
        <Description>.NET class library in PHP</Description>
        <LibraryName>$(MSBuildProjectName)</LibraryName>
        <LibraryPath>
            vendor\$(ProjectNameLocation.toLower())
        </LibraryPath>
        <AssemblyName>$(LibraryName)</AssemblyName>
        <PackageId>$(LibraryName)</PackageId>
        <GeneratePackageOnBuild>True</GeneratePackageOnBuild>
        <PackageOutputPath>
            ..\..\..\Nuget-Repository\Components\
        </PackageOutputPath>
        <IntermediateOutputPath>
            .\obj\$(Configuration)\$(ProjectNameLocation)
        </IntermediateOutputPath>
        <OutputPath>
            .\bin\$(Configuration)\$(ProjectNameLocation)
        </OutputPath>
        <OutDir>$(OutputPath)</OutDir>
        <ComposerJsonPath>Ignore</ComposerJsonPath>
    </PropertyGroup>

</Project>
```

Listing 5.2: propertiesConfig.props

compilationConfig.props Now, that we have successfully set up all the directories and the names used on the compilation, we can move on to the definition of what to be compiled and included in the final package. We can expect that all of the projects will share the same patterns in compile and content files' specification, that is in general, they will compile all of the `.php` files located in the library folder and include all the files as content. However, such assumptions proved to be much more naive that might seem.

We must not forget that we are compiling one language into another which in this case brought a problem of using features of one language, in our case PHP, that simply cannot be brought into the world of .NET. The most frequently

encountered error associated with this was an inclusion of a non-existing class or interface. In PHP's testing classes it is very common to include external libraries that are not mentioned in the `composer.json` file as dependencies. This is because test classes are not executed in typical program flow and require explicit actions for their execution. It does not change the fact that triggering such action manually will cause the original PHP project to crash. In some cases however, PHP does check presence of a class prior to its actual usage, protecting itself from executing a file that includes such class. Unfortunately, this is not supported in .NET as each file compiles independently and the fact that it is not executed does not change the fact that it must be compiled.

There are two possible solutions for this problem. Either we can look up the dependency tree of each such test class, compile them into separate NuGet packages and reference them, or we can exclude the tests. The later solutions proves to be better for most of users as a typical user does not perform downloaded libraries' testing. Therefore, their inclusion would cause unnecessary overhead in the the number of NuGet packages required for the project's compilation and also in the time of the actual project compilation. The test files are the only files we can generally exclude from every project, but in most cases, are also the only files required to be excluded in order to achieve a successful compilation. After we extended the naive configuration from the previous paragraph with the tests exclusion, we ended up with a specification shown in the Listing 5.3.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <Compile Include="$(LibraryPath)**\*.php"
            Exclude="   $(LibraryPath)**\test\**;
                        $(LibraryPath)**\Test\**;
                        $(LibraryPath)**\tests\**;
                        $(LibraryPath)**\Tests\**;"/>
        <Content Include="$(LibraryPath)**"
            Exclude="   **\*.msbuildproj;
                        **\*.csbuildproj;
                        **\*.props;
                        $(LibraryPath)obj\**;
                        $(LibraryPath)bin\**;"
            CopyToOutputDirectory="PreserveNewest">

            <BuildAction>Content</BuildAction>
            <PackagePath>
                contentFiles\any\netcoreapp2.0\$(LibraryPath)
            </PackagePath>
            <PackageCopyToOutput>true</PackageCopyToOutput>
            <Visible>true</Visible>
            <Pack>true</Pack>
            <Link>
                $(LibraryPath)%(RecursiveDir)%(Filename)%(Extension)
            </Link>
        </Content>
    </ItemGroup>

</Project>
```

Listing 5.3: compilationConfig.props

Required code edits Unfortunately, not all the compilation errors could be resolved by excluding the problematic source code. Sometimes, core parts of Symfony Components use language constructs that PHP supports and .NET

does not. For these files, the only remaining solution is to edit the code itself and refactor the functionally to a similar but compilable version.

Several times, Symfony Components used the `yield` keyword inside a `try -catch` block. Such operation cannot be performed under the .NET environment as it places strict restrictions on the usage of iterators inside those blocks. These cases have been rewritten into equivalent form with the use of arrays and booleans as flags. Furthermore, as such edits can be automatized, PeachPie developers have later implemented workarounds to handle these cases without compilation errors.

The second encountered compilation errors were caused by an inappropriate usage of the `__toString()` function. There are special rules applying to this function and its return values in PHP which the Peachpie compiler strictly complies with. Symfony on the other hand, occasionally throws an exception there which when compiled resulted in an invalid return type error. Such code was, for the sake of further progress, temporarily rewritten to a simple empty string returning and later resolved in another PeachPie compiler update.

The next problem encountered was of the same nature as the first one mentioned, that is, in the disruption of the program flow. Symfony's Flex Component used the `return` statement from the `finally` block of an exception handling construct. However, this Component is used solely as a supplement for the Composer tool, and thus, for the purpose of managing additional libraries. Compiled Symfony project has therefore no use for such functionality as requiring of additional packages from compiled projects via Composer or Flex is redundant with the introduction of NuGet packages. Thanks to that, the Flex Component can be completely omitted from the compilation.

The last type of compilation errors was caused by a missing implementation of a small set of features within the Peachpie compiler. Each of them was later fixed in a newer release of the software. This could not be helped as the Peachpie compiler is still in the phase of development and such problems were expected to appear. Along some of these missing features, we can name missing implementations of the `SessionUpdateTimestampHandlerInterface` interface or the `PCRE_ANCHORED(A)` flag in Peachpie's perlregex.

5.4 Targets file and ComponentTools

Even with the mentioned code edits and 3 `.props` files ready, we cannot proceed to the actual Symfony Components compilation yet. There have been several details that were intentionally left out or skipped in the previous sections and addressing them is a job of the `automatizationConfig.targets` file and the `Peachpied.Symfony.ComponentTools` project. These details skipped were packages' versions and packages' dependencies, and omitted was explanation what to do with those Symfony Components that require more precise compilation configuration than just an excluding of all the test sources.

automatizationConfig.targets This configuration file specifies custom targets and hooks them into each Component's compilation process. This can be achieved in multiple different ways but the most common is either by the `BeforeTargets` or the `AfterTargets` attribute of the `Target` element. Aside

basic `BeforeTargets` and `AfterTargets`, MSBuild specifies additional means of target execution flow modifications. One of them is the `DependsOnTargets` attribute, resulting in a sequential triggering of all the stated targets prior to the target's body execution. Each target can also define the `Inputs` and the `Outputs` attributes. When present, timestamps of specified input and output files are compared, possibly resulting in marking the target as up to date and skipping its execution. Finally, the `Condition` attribute can be used to control target's execution with a `boolean` expression (this attribute can also be used on tasks' elements, `itemGroups` and many other elements).

Now, we can proceed to the actual `automatizationConfig.targets` file content. Each of the 3 missing details mentioned before is provided by different task but all of them are located in `Peachpied.Symfony.ComponentTools.dll`. The implementation of these tasks will be discussed in the upcoming section. Naturally, the task responsible for retrieving Component's package references needs to be executed before the `Restore` target and all of its preprocessing during the Component's compilation. Unfortunately, this makes it impossible to distribute the `.dll` as a NuGet package. The ideal solution for this problem would be to include the `.dll` inside of the Peachpie SDK but due to a large extent of such operation we decided to distribute it only in a way of source code and the static `.dll` in this repository.

Each one of the custom tasks is dependant on existence of the `libsCache.json` file produced by the `TWarmLibsCache` target. This `.json` file contains necessary information that is further transformed by other 3 tasks into required properties and items. Caching target's execution is bound by the `composer.lock` and `libcCache.json` files' timestamps using the technique mentioned before, ensuring the cache file is regenerated only when necessary. Sadly, it is not possible to use a collection of Symfony Components' `composer.json` files instead of `composer.lock` because the `.lock` file contains the `version` key supplied by the Composer during package installation which is absent from Components' `composer.json` files.

Finally, parts of the `automatizationConfig.targets` file providing discussed functionality can be seen in the following Listing.

```
...
<UsingTask TaskName="WarmLibsCache" AssemblyFile="$(LibsAssebly)" />
<UsingTask TaskName="GetLibVersion" AssemblyFile="$(LibsAssebly)" />
<UsingTask TaskName="GetLibReferences" AssemblyFile="$(LibsAssebly)" />
<UsingTask TaskName="FilterLibExcludes" AssemblyFile="$(LibsAssebly)" />

<Target Name="TWarmLibsCache" Outputs="$(CacheDir)\libsCache.json"
        Inputs="$(MSBuildProjectDirectory)\composer.lock">
    <Delete Files="$(CacheDir)\libsCache.json"
            Condition="Exists('$(CacheDir)\libsCache.json')" />
    <WarmLibsCache ProjPath="$(ProjPath)" ConfigPath="$(ConfigPath)"
            CachePath="$(CacheDir)" />
</Target>
...
<Target Name="TGetLibReferences" BeforeTargets="CollectPackageReferences"
        DependsOnTargets="TWarmLibsCache">
    <GetLibReferences CachePath="$(CacheDir)" RepoPath="$(RepoPath)"
            LibName="$(MSBuildProjectName)"
            Condition="$(ManualReferences)!=true">
        <Output TaskParameter="References" ItemName="PackageReference" />
    </GetLibReferences>
</Target>
```

```
<Target Name="TSymfonyAutomatization" BeforeTargets="BeforeBuild"
        DependsOnTargets="TWarmLibsCache">
    <GetLibVersion CachePath="$(CacheDir)"
            LibName="$(MSBuildProjectName)"
            Condition="$(ManualVersion)!=true">
        <Output TaskParameter="Version" PropertyName="PackageVersion" />
    </GetLibVersion>

    <FilterLibExcludes ConfigPath="$(ConfigPath)"
            LibPath="$(LibraryPath)" LibVersion="$(PackageVersion)"
            LibName="$(MSBuildProjectName)"
            Compile="@(Compile)" Condition="$(ManualExcludes)!=true">
        <Output TaskParameter="NewCompile" ItemName="NewCompile" />
    </FilterLibExcludes>
    <ItemGroup>
        <Compile Remove="@(Compile)" />
        <Compile Include="@(NewCompile)" />
    </ItemGroup>
</Target>
```

Listing 5.4: automatizationConfig.targets

ComponentTools The first task that the project's `.dll` provides is called `WarmLibsCache`. As its title and input parameters from the previous Listing suggest, this task loads the `libsConfig.json` configuration file from `ConfigPath` and the `composer.lock` file from `ProjPath`, and creates a cache file at `CachePath`. This cache file lists all the installed Symfony Components, sorted in a cumulative manner, according to their dependencies. For each component, there is an object indexed by its name, holding the Component's version, indicator whether it was listed as a development dependency in the `composer.lock` file and a list of names of packages it depends on. Where the first two values are simply parsed out from the `composer.lock` file, the process of gathering each Component's references proved to be much more complicated and will be described in detail right ahead.

What might seem as a task that could be solved simply by loading an array of names from the `require` key in Component's record of the `composer.lock` file, turned out much more complicated after realising that for a successful compilation, Components do also require references to packages listed under the `require-dev` key. This does not come as a surprise because from the .NET's point of view it does not really matter whether some functionality is actually used or not during the compilation. However, joining these two arrays of package names would have not present a challenge, were it not for the circular dependency errors that appeared during compilation. Because of that, the whole process of package dependency tree building and optimizing had to be implemented.

In the first phase of the algorithm, a simple dependency tree is build. This is done using custom data structure for each node, holding its name, version and a list of children nodes, its dependencies. After the tree is incrementally built, each node is traversed recursively creating a list of unique cycles. At last, all edges in each cycle marked as development dependencies are selected and the one leading to the package with the longest name is removed. This "random" approach proved to be working for every circular dependency error containing at least one development edge encountered.

Unfortunately, some of the errors did not contain any development edge and thus required additional information on how to handle such cycle. This is provided via the `libsConfig.json` configuration file mentioned before.

Finally, a list of objects with versions, development indicators and valid dependencies is sorted in order to allow sequential build and saved to the `libsCache.json` file. This class grants access to this list via a public property along with a list of packages and development packages loaded from the `composer.lock` file. We will skip the reasons for this and return to it at the end of the chapter.

As we allready know, the `libsConfig.json` file contains additional configuration for the Symfony Components' compilation defined by the user. At the same time, it is also the only actual information that has to be provided manually. Aside from the mentioned hints for circular dependencies resolving, additional excludes or overriding includes of default excludes can be specified there. This can be done either separately for each version of each Component or globally for every version without its explicit configuration.

Thanks to this manual configuration and the cache file, the `GetLibVersion` and the `GetLibReference` tasks consist solely of Component's record loop-up in the cache file and storing of an appropriate value in either the `PackageVersion` property or the `PackageReference` item. The remaining `FilterLibExcludes` task is specific in its functionality because it does not provide any new configuration property or item. Rather, it modifies the value of the `Compile` item by applying additional excludes and includes from the `libsConfig.json` file. After that, content of the former `Compile` item is replaced with a new content in order to prevent duplicate file entries, which would result in compilation errors.

5.5 Automatization of the process

Thanks to all the tools and external configuration mentioned, our project file that will compile a single Symfony Component into the NuGet package reduces to a few lines displayed in the next Listing.

```
<Project Sdk="Peachpie.NET.Sdk/">
    <Import Project="....\Props\automatizationConfig.targets" />
    <Import Project="....\Props\propertiesConfig.props" />
    <Import Project="....\Props\compilationConfig.props" />
</Project>
```

Listing 5.5: compilationConfig.props

Such files could be produced automatically for every Symfony Component in a given directory or project. The tool providing this functionality was named `Peachpied.Symfony.BuildGen` and its process of execution is displayed in the Figure 5.1, which will be further explained in the rest of this chapter.

Peachpied.Symfony.BuildGen This utility is distributed as an executable file that generates `.msbuildproj` and other necessary files for every Symfony Component found in the target directory. These are marked red in the Figure 5.1. In addition to, the utility also outputs the `buildScript.ps1` script that can be run in order to execute `dotnet build` on each produced `msbuildproj` in correct order. The target directory for the build generator is supplied by the `-p, --project` parameter. Aside from that, the utility also accepts optional paths to the `libsConfig.json` and the `.props` files that are otherwise set to the default values regarding the repository structure. Flag parameter `-b, --build`

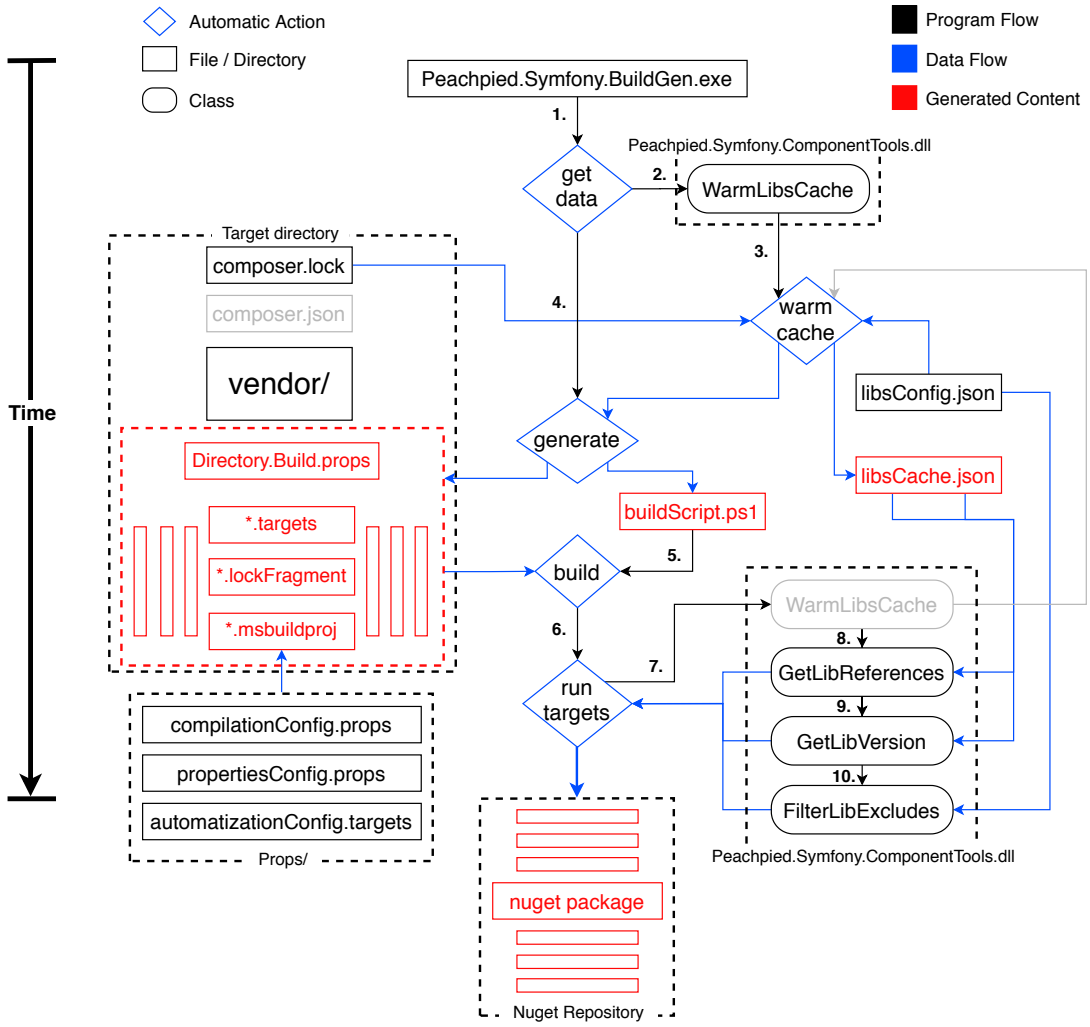

Figure 5.1: Compilation process of Symfony components

automatically runs generated `buildScript.ps1` on created files, and with the `-d, --debug` flag it is possible to skip the final clean-up phase that automatically removes all the generated files. By default, the clean-up is naturally performed only when the `-b` parameter is included.

List of Symfony Components present in the target directory is obtained using the `WarmLibsCache` class. This step is marked `1.` and `2.` in the Figure. In the step `3.` of the Figure, we prepared public properties of the class with the help of the `composer.lock` and the `libsConfig.json` files. Load-up of these files is marked with the blue arrows leading towards the `warm cache` action in the Figure. As a side effect of the `WarmLibsCache` class execution, the `libsCache.json` file is produced for the obtained list of Symfony Components.

Step `4.` of the Figure marks transfer to the generating phase of the compilation. Regarding the actual files provided by the build generator utility, it firstly creates the `Directory.Build.Props` file. After that, the necessary information like the list of package data and an ordered collection of package names are loaded from the `WarmLibsCache` class and one `[PackageName].msbuildproj`, one `[PackageName].lockFragment.json` or one `[PackageName].lockFragment`

`-dev.json`, and one `[PackageName].targets` file is generated for each Symfony Component. All generated files are placed in the target directory provided, in order to comply previously mentioned requirement for paths to NuGet packages' static files.

The `.lockFragment` files hold each Component's record parsed out from the `composer.lock` file. The `.targets` files are special as every NuGet package can include one such `.props` and one such `.targets` file that will be automatically imported into the project referring the NuGet. The requirements for this to work are to name the files according to the package name and place them inside the `build` directory of the package. The files can be also copied to the `buildTransitive` directory which will result in transitive behaviour of the functionality. However, this feature is only available in the NuGet version 5.0 and newer. Each `.targets` file provides a functionality to restore Symfony Component's folder inside the `vendor` directory of the project that uses the NuGet. In addition to, it also exposes the `LockFragments` item with appended path to its `.lockFragment.json` file. These functionalities are required for additional automation during actual Symfony project compilation and thus will be discussed in greater detail in the next chapter. A truncated example of one such `.targets` file can be found in the following Listing.

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         InitialTargets="Copy_Psr_Cache_Vendor">

    <ItemGroup>
        <LockFragments Condition="{* file exists *}"
                Include="{* package/build/ *}
                ../tools/any/any/Psr.Cache.lockFragment.json" />
        <LockFragments Condition="{* file exists *}"
                Include="{* package/build/ *}
                ../tools/any/any/Psr.Cache.lockFragment-dev.json" />
    </ItemGroup>

    <Target Name="Copy_Psr_Cache_Vendor"
            Condition="$(RestoreVendor)==true
            AND {* project/vendor/psr/cache does not exist *}">
        <Message Text="Copying Psr.Cache assets..." Importance="high" />
        <ItemGroup>
            <Copy_Psr_Cache_Vendor_Files Include="{* package/build/ *}
            ../contentFiles/any/netcoreapp2.0/vendor/**/*.*" />
        </ItemGroup>
        <Copy SourceFiles="@(Copy_Psr_Cache_Vendor_Files)"
                DestinationFolder="{* project/vendor *}"/>
    </Target>

</Project>
```

Listing 5.6: Psr.Cache.targets

Furthermore, with the use of the previously mentioned ordered collection of package names, the `buildScript.ps1` file is generated providing means of automatic compilation of Symfony Components in the order denoted by the dependency tree.

In case the `-b` flag was used, the utility next runs the `buildScript.ps1` script and proceeds to the compilation, marked with number `5.` in the Figure. This is performed as described in the previous sections. First, the `.msbuidproj` file is loaded. After that, MSBuild targets run in execution order and handle

importing of `.props` with `automatizationConfig.targets` and ultimately locating package specific `lockFragments` and `.targets` as well. Figure 5.1 depicts this phase separated from automatization targets execution in sake of increased readability. During the MSBuild targets execution named the `run targets` action in the Figure 5.1, custom targets are executed chronologically in displayed order providing missing information for the compilation and optionally skipping the `WarmLibsCache` target in case the `libsCache.json` file is already present in the target directory. This corresponds to the steps `6.` to `10.` in the Figure and on completion, passes control back to the MSBuild targets that finally produce NuGet packages.

We can now sum-up the actual compilation process of a set of Symfony Components from the earliest phase of PHP sources downloading to producing the actual NuGet packages into the following steps.

i: Clone the repository and initialize a directory under the `Peachpied.Symfony/Components` path with the `composer init` command (optionally skip the interactive `composer.json` configuration).

ii: Require a set of Symfony Components either by running `composer require [packageName:version]` or in the interactive configuration during the previous step.

iii: Compile the `Peachpied.Symfony.BuildGen` sources into an executable file with the `dotnet publish -r Runtime` command.

iv: Run the build generator tool supplying the path to the target directory.

v: Modify uncompilable sources and extend the `libsConfig.json` file with an additional configuration for problematic packages if needed.

vi: Find produced NuGet packages under the `Nuget-Repository/Components` path.

Finally, one can notice that to compile Symfony Components of an existing Symfony project, first two steps will not probably need to be executed. The third step is also a one time job, and the fifth step should not be needed when the Peachpie compiler is ready for the production and when the `libsConfig.json` file contains all the necessary configurations. Therefore, the whole compilation would in such case reduce to a single step of manual execution of the build generator tool. This can be observed on the Figure 5.1 where the execution of the tool is followed by automatic actions all the way to NuGet packages producing.

6. Project compilation

This chapter focuses on the way actual Symfony projects are compiled and deployed using he Peachpied Symfony toolkit. It introduces tools that make it possible and describes the process of their development. At the end of the chapter, an overview of provided feature of interoperability of template engines is revealed.

6.1 Compilation setup

Whether we want to embed our existing Symfony application within another .NET Core one or just to compile it for enhanced security, we always have to produce it in a form of a `.dll` and supply it to a .Net Core web server. In this case, we decided to choose Kestrel web server as it is both open-source and Peachpie already contains an extension that allows Kestrel to handle requests targeting `.php` files and process them with the `dll` library compiled from the PHP code [19]. This binding is in our case done by creating two .NET projects, one for the Symfony sources and the other for the server.

Binding with the web server For this, we can use an already made template called "Web App" provided by Peachpie. Upon usage, the template produces the `Website` and the `Server` directories. The `Server` one contains a simple project with a base configuration for Kestrel server and a logic for `.php` request handling. The other one is expected to contain a PHP code for compilation.

Apart from the configuration files `appsettings.json` and `launchSettings.json`, and project file `.csproj`, the server project directory by default contains only the `Program.cs` file representing an entry point of the application. It consists of two classes, the `Program` class with the `static void Main(string[] args)` function and the `Startup` class with the already discussed `ConfigureServices` and `Configure` functions. Furthermore, Listings 2.1, 2.2 and 2.3 combined represent the whole default `Program.cs`. The only difference is that the `Main` method must declare the actual Symfony project name in the Host object configuration.

Configuring Symfony's .msbuildproj For the majority of pure PHP web applications, the system described will already be executable, however, few further adjustments in Website's `.msbuildproj` file have to be done in order to run our application as expected.

First, we have to modify a value of the `Compile` item. The Website project compiles every `.php` file by default which is clearly more than we actually need. As we supply Symfony Components via package references, we can safely exclude content of the `vendor` directory which might in case of compiling an existing Symfony project occur to exist in the `Website` directory. We have to make an exception for the autoloading scripts in the exclusion rule as they are part of the `vendor` directory as well.

Secondly, we need to include all the static content of the project like configuration `.yaml` files, but images and other assets as well. This is done by setting up the `Content` item.

Thirdly, package references to the Symfony Components used by the original project need to be manually set up. This is done by copying them from the project's `composer.json` file and rewriting them to the NuGet references. For this reason, the `Peachpied.Symfony.Skeleton` NuGet package was created that groups together dependencies of the Symfony Skeleton project with a matching version. As this thesis focuses solely on Symfony 4.2.3, this package is prepared only in this particular version. However, automatic version resolving target was implemented as a part of provided Skeleton's `.msbuildproj` file allowing their future arrangement into folders by version names.

Finally, one must not forget about the autoloading scripts and he cache directory discussed in the chapter 4. For now, we will settle for knowing, that all these problems will be resolved automatically by adding package reference to the `Peachpied.Symfony.ProjectTools` NuGet package. Implementation details of this set of tools will be discussed in the following section. Compilation configuration for the bare bones Symfony application will in the end look as shown in the following Listing.

```
<PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>netstandard2.0</TargetFramework>
    <AssemblyName>Empty-Page</AssemblyName>
</PropertyGroup>

<ItemGroup>
    <Compile Include="**\*.php"
             Exclude="vendor\**; config\bootstrap.php;
                      .\obj\**; .\bin\**" />
    <Compile Include="vendor\composer\**\*.php; vendor\autoload.php" />
    <Content Include=".\**"
             Exclude="{* all non configuration files *}"
             CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>

<ItemGroup>
    <PackageReference Include="PeachPied.Symfony.Skeleton"
             Version="4.2.3" />
    <PackageReference Include="Peachpied.Symfony.ProjectTools"
             Version="1.0.0" />
</ItemGroup>
```

Listing 6.1: Empty-Page.msbuildproj

In order to bring the UX even further, all the previously mentioned steps were compressed into two new templates provided by the `Peachpied.Symfony.Templates` NuGet package. The "Symfony Skeleton" or "syskeleton" template creates a doublet of base Kestrel server configuration and bare bones Symfony application version 4.2.3 with all the `.php` sources provided. The second template called "Symfony-Empty-Project" or "syemptyproj" creates one project with the same Kestrel configuration as the previous template, but the Symfony project produced contains only the `.msbuildproj` file without any package reference to Symfony Components, letting the user import own sources and setup required package references as required.

6.2 ProjectTools package

Just like the mentioned `ComponentTools` hooked the custom targets in order to provide necessary configuration settings during the Symfony Components compi-

lation, `ProjectTools` does the same thing during the Symfony project compilation in order to provide the compilation with additional `.php` files needed for actual application execution. Custom targets provided by this project are named by to the purpose they server: `RestoreComposerLock`, `GenerateSymfonyAutoload` and `GenerateSymfonyCache`. Because these tasks do not manipulate project's package references, whole project can be safely delivered in a form of a NuGet package. Furthermore, we can make use of discussed NuGet's feature to import special `.targets` file into out project file. This way, we prepare all the custom targets' load-up and usage in `Peachpied.Symfony.ProjectTools.targets` and let the package manager handle their embedding into compiled project's execution order, while effectively isolating user from all the background tasks.

RestoreComposerLock This task is responsible for collecting of all the `lockFragments` from NuGet packages of Symfony Components and joining them together into `composer.lock` file. The data from packages are in fact extended by the data taken from the project's `composer.json` file in order to provide a file that matches the one originally produced by Composer. This required to rewrite an implementation of the content hashing function used by Composer into C# in order to provide the similar value for the `content-hash` key.

This task's execution is bound by appropriate `Inputs` and `Outputs` values, ensuring that it runs only when there is no `composer.lock` file present or its content is outdated compared to `composer.json`.

GenerateSymfonyAutoload Things get slightly more complicated with this task due to requirements to use Composer itself to produce autoload files for us. The implementation we settled for includes the whole `composer.phar` file as a static asset of the NuGet package and calls it on task's execution with the `php composer.phar dump-autoload` command.

Unfortunately, quite severe problem emerged regarding Composer's `dump-autoload` requirements. For its proper execution, this task needs the `installed .json` file to be located under the `vendor/composer` path in the project directory. Thankfully, this file is nothing more than a concatenation of each package's `lockFragment` and thus can be easily created using previously generated `composer.lock`. On the other hand, it does also require for all the `.php` source codes of the Symfony Components that are autoloaded, to be physically present under the respective subpath of the `vendor` directory. We resolved this during the Symfony Components compilation as well by making Components' NuGet packages able to copy their source files for us just by setting the `RestoreVendor` property to `true`. This is done once again by default inside of the `Peachpied .Symfony.ProjectTools.targets` resulting in the background recovery of the `vendor` directory in case it is missing. We can see now, that this approach introduces serious time overhead during the first and each subsequent project compilation that results in the `vendor` folder recovery. For maximal efficiency, there are constraints ensuring this recovery gets triggered only for Symfony Components whose subdirectory of the `vendor` directory is missing. Were we to successfully compile Composer into .NET, we would be able to omit the whole recovery phase and supply compiled `.php` sources directly from NuGet packages.

RestoreSymfonyCache After autoloading scripts are produced, this task undirectly uses them with `src/Kernel.php` to produce Symfony cache. This is usually done by calling the `boot` method on an instance of the `Kernel` class. Additionally, routing cache can be produced with the `getMatcher` method of the `Router` class. The process is complicated due to the fact that these classes load configuration files stored under the `config` directory which are not only `.yaml` files but there are some configuration `.php` files as well. Because of their dynamic content, it is not possible to pre-compile them, and thus the only possible way to handle them from .NET would be to write a custom parser of these `.php` files. However, as we already depend on PHP being installed on the user's machine, nothing stops us from using these methods natively from PHP. For this reason, a simple PHP script handling cache generating was written, included with the `composer.phar` in the `ProjectTools` NuGet package and is triggered during the task's execution.

Furthermore, the constructor of the `Kernel` class accepts parameters identifying the current environment and thus influencing produced cache. These parameters are a `string` value for `$environment` and a `boolean` value for `$debug`. Values for these parameters are passed to the PHP script from the task where they are obtained by manual parsing either from the `.env` file or the `.appsettings.json` file. As is discussed in greater detail later on, Symfony uses the `.env` and the `.env.[environment]` files for environmental variables configuration, while .NET Core uses the `appsettings.json` and the `appsettings.[environment].json` files.

6.3 Peachpied.Symfony.AspNetCore

With the configuration as shown in the Listing 6.1, the application will successfully compile into a `.dll`. The `Program.cs` file in the Server project provided by one of the Symfony templates will then start a web server and upon accessing the application on default port, content of the `public/index.php` file will be displayed. In order for everything to work as expected, the Server project uses the `Peachpied.Symfony.AspNetCore` NuGet package and its `UseSymfony` method. Moreover, many additional efforts had to be made in misbehaviour errors identification, and by the PeachPie development team in particular, during Peachpie-related problems resolving. Significant errors encountered will be discussed later.

This package is intended as a helper class for the Server project, encapsulating the Symfony specific server configuration and an implementation of additional interoperability features. Both are provided via the API, following the standard ASP.NET Core `Use[midlewareName]` naming conventions. The public API is provided by the `RequestDelegateExtension` class and provides the `UseSymfony` method discussed in the following paragraph.

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

Aside from calling the `UsePhp` method provided by `Peachpie.AspNetCore.Web` which implements the actual middleware, the method configures static file

serving middleware, further discussed URL rewriting, environmental variables load-up, and optionally overrides Symfony's `bootstrap.php` which is responsible for environmental variables loading in Symfony.

The content of the `bootstrap.php` file can be found in the Listing 6.2 without original comments that have been omitted for the sake of increased readability. The fragment of the code performing the overriding is captured in the Listing 6.3. This is done by accessing the `Context` object's script table that performs mapping from `.php` files to compiled .NET classes, and overwriting path to the class stored for the `bootstrap.php` file. It is also worth of notice, that our `BootstrapMain` class effectively substitutes original file's additional functionality and overrides just the environmental configuration.

```
use Symfony\Component\Dotenv\Dotenv;

require dirname(__DIR__).'/vendor/autoload.php';

if (is_array($env = @include dirname(__DIR__).'/.env.local.php')) {
    $_SERVER += $env;
    $_ENV += $env;
} elseif (!class_exists(Dotenv::class)) {
    {* Runtime exception *}};
} else {
    (new Dotenv())->loadEnv(dirname(__DIR__).'/.env');
}

$_SERVER['APP_ENV'] = $_ENV['APP_ENV'] = ($_SERVER['APP_ENV']
    ?? $_ENV['APP_ENV'] ?? null) ?: 'dev';
$_SERVER['APP_DEBUG'] = $_SERVER['APP_DEBUG']
    ?? $_ENV['APP_DEBUG'] ?? 'prod' !== $_SERVER['APP_ENV'];
$_SERVER['APP_DEBUG'] = $ENV['APP_DEBUG'] = (int) $_SERVER['APP_DEBUG']
    || filter_var($_SERVER['APP_DEBUG'], FILTER_VALIDATE_BOOLEAN)
        ? '1' : '0';
```

Listing 6.2: bootstrap.php

```
static SymfonyConfig bootstrapConfig;

static PhpValue BootstrapMain(
    Context ctx,
    PhpArray locals,
    object @this,
    RuntimeTypeHandle self
) {
    ctx.Include("vendor", "autoload.php", true);
    Apply(ctx, bootstrapConfig);
    return 0;
}
...
// bootstrap.php env loading overriding
string bootstrapPath = "config\\bootstrap.php";
Context.MainDelegate md = newContext.MainDelegate(BootstrapMain);
Context.DeclareScript(bootstrapPath, md);
```

Listing 6.3: Context file mapping overriding

Environmental variables Because Peachpied Symfony performs project migration to the ASP.NET Core environment using the .NET based web server, it can be expected that users will want to set up environmental variables in a .NET-specific way with the use of previously mentioned `appsettings.json` configuration file. On the other hand, this way of configuring should not be compulsory

as there might also be users who would prefer to simply compile existing Symfony project without the need to worry about environmental variables letting the application use Symfony's configuration already present in the code. This functionality is delivered by the `SymfonyConfig` and the `SfConfigurationLoader` classes. The first one represents a definition for environmental variables and their default values. Therefore, it consists solely of properties. The second class contains methods for configuration values parsing, default configuration loading and `appsettings.json` files processing.

Runtime problems Probably the most significant problem encountered was due to the differences in the server root directory between Symfony and .NET. In Symfony, it is expected that a server operates from the root directory set to `public/`, therefore, all paths to assets like styles, scripts and images are set relative to this folder. In our .NET application however, we set the server root to the parent directory of the Symfony project. We have to do it this way in order to be able to reach assets from both the applications. Peachpie's `UsePhp` midlleware simplifies this problem partially by locating Symfony projects' sources for us, however, it does not help us with the problem of assets that are expected to be relative to the `public` directory.

This problem was resolved by adding URL rewriting into the `UseSymfony` method. URL rewriting is a technique used to modify request URLs based on one or more predefined rules [16]. It creates an abstraction between resource locations and their addresses so that the locations and addresses are not tightly linked. [16]. In ASP.NET Core, the `UseRewriter` middleware serves this cause and for that accepts instance of the `RewriteOptions` class as an argument. The actual implementation of the URL rewriting in Symfony can be found in the following Listing.

```
var options = new RewriteOptions()
    .AddRewrite(
        @"((.*\/)*([^\/\?&]*\.[^\/\?&]*))",
        bootstrapConfig.PublicDir + "/$1",
        skipRemainingRules: true
    )
    .AddRewrite(
        @"^(?!"+ bootstrapConfig.PublicDir + "/)(.*)$",
        bootstrapConfig.PublicDir,
        skipRemainingRules: true
    );

app.UseRewriter(options);
```

Listing 6.4: Symfony URL rewritting middleware

Regarding the Peachpie-related errors, we will mention invalid handling of the `new ReflectionProperty("Exception", "trace")` expression. This was done due to lack of formal documentation of the private `$trace` property of the `Exception` class. For the moment, this enabled the application to run only with the `APP_DEBUG` environmental variable set to false. However, the error was later resolved by an update of the PeachPie compiler and therefore the debug mode is fully supported at the moment. Along some less complicated errors, we can name an incorrect `realpath` function return value implementation, where the function returned `null` instead of `false` on failure.

6.4 Additional interoperability

In addition to the already mentioned `UseSymfony` method, the `Peachpied .Symfony.AspNetCore` API provides two classes that display some of the interoperability possibilities brought by the Symfony compilation. These classes are named `TwigRenderer` and `RenderRazorService`, and both are stored in the `Templating` namespace of the `Peachpied.Symfony.AspNetCore` NuGet package.

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 `Peachpied.Symfony.Skeleton` NuGet package, but only with the help of two additional packages, the `Twig.Twig` package and its single dependency, the `Symfony.Polyfill-mbstring` package.

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 the 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 `RazorViewEngine` or the `ServiceProvider`. To take the advantage of the Dependency Injection process of Asp.Net Core, the class had to be registered as a service in the server's `Startup` class, within the `ConfigureServices` method while implementing custom `IRenderRazorService` interface. This way, the application provided the necessary objects for us, creating an instance of the `RenderRazorService` that can be 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 is therefore directly accessible from the 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, a 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 was entitled 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 Peachpie's PHP `Context` provided on the instantiation, then uses it to access the `RazorRenderService` and finally 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.

7. Examples of usage

Firstly, we will provide an introduction into deployment of both new and existing Symfony applications. Following with an example of merging the ASP.NET Core application using Twig + Razor with the Symfony application also using Twig + Razor. Finally, an example of the pure ASP.NET Core application with both Twig and Razor template engines is described.

Both examples further mentioned can be found in the `Examples` directory of the project repository. For their execution, we used the .NET Core SDK version 2.2.301 and the .NET Core Runtime version 2.2.6, but any mutually compatible 2.x versions should work. They also require the NuGet package manager version 5.0 or newer and PHP 5.3.2 or newer. For the proper functioning, PHP also requires the `openSSL` extension to be either correctly configured or disabled with the `composer config -g -- disable-tls true` command, which in turn requires previous installation of Composer. Examples can be also executed with the Visual Studio IDE but due to the mentioned version of the NuGet package manager required, this might be only possible in Visual Studio 2019.

7.1 Compiling Symfony Components

The process of Symfony Components compilation can be easily tested on a small set of Symfony Components that are required by the very basic Symfony application called Symfony Skeleton. Additionally, Twig template engine Components with their dependencies can be counted in as well. These Components with required source code edits and a Component-specific build configuration prepared are included as a part of the repository and can be found in the `Peachpied .Symfony/Components/4.2.3/vendor` directory.

Pre-compiled tools that take care of the compilation are included as a part of the project inside the `Libs` directory but they might require recompilation due to the possible differences in the operating systems and the CPU architectures. To test this, simply run the

```
./Peachpied.Symfony.BuildGen.exe
--project ../Peachpied.Symfony/Components/4.2.3
```

command from the `Libs` directory. In case of success, a set of `.msbuildproj` files will be generated inside the provided working directory, one for each Symfony Component installed there. The script will also produce the `buildScript.ps1` script inside the `Libs` directory. In case the script execution fails, the tools might require recompilation because of differences in architecture of the computer it was compiled on and the one it is now executed on. To do so, clear the content of the `Libs` directory and execute `dotnet publish -r [Runtime]` from the `Tools/BuildGen` directory of the repository. The `[Runtime]` parameter is to be replaced with custom .NET Core Runtime Identifier. For the table of available values refer to the Microsoft .NET Core guide. The Runtime Identifier used for the tools compilation is `win10-x64`.

To perform the actual compilation execute `Libs/buildScript.ps1` generated before. Alternatively, the whole compilation process can be achieved by running:

```
./Peachpied.Symfony.BuildGen.exe
--project ../Peachpied.Symfony/Components/4.2.3
--build
```

from the `Libs` directory. The additional `--build` flag tells the tool to automatically execute `buildScript.ps1` after build files are generated, making it possible to compile all the Components with a single command. Running this command should output 35 NuGet packages into the `Nuget-Repository/Components` directory, corresponding to the 35 Symfony Components compiled. We have precompiled these packages and included them as a part of the project repository.

Custom Components Compilation Following instruction focuses on describing how to compile custom Symfony Components of different versions than those provided with the repository from downloaded sources. This process will most probably require to manually modify downloaded sources and might even require further cooperation with the Peachpie development team in order to resolve all the compilation errors. This guide is therefore suited for users with a deeper understanding of MSBuild and a greater passion for the subject as we can provide no manual that describes how to resolve all possible compilation errors.

First of all, it is necessary to get the actual Symfony Components. This is done by using Composer, a PHP package manager, and thus requires having both PHP and Composer installed. Prior to any Symfony Component downloading, we have to initialize the working directory as a project. This is done by running the

```
composer init
```

command and following the interactive process, and results in creation of the `composer.json` file. Values set for the project name, authors, etc. during the interactive mode are of no concern as our compilation process does not pay any attention to them. Alternatively, an empty `composer.json` file with just {} as its content will work the same. Finally, each Component that is to be compiled can be downloaded using the

```
composer require [Component name](:[version])
```

command. This will further update the `composer.json` file as well as produce the `composer.lock` file with detailed information about each Component downloaded. The presence of both these files within the working directory is mandatory for any further progress.

When Symfony Components are in place, we can perform compilation in a way mentioned in the previous paragraph. The only modification required is to provide custom relative path to the working directory as an argument to the `--project` parameter.

The next step consists of repetitive attempts to compile the Components and of an application of required fixes. To trigger Components compilation, run the previously mentioned `buildScript.ps1`. This will perform sequential compilation of all the Symfony Components while adhering to the required order denoted by the dependency tree. Each compilation error encountered has to be

resolved manually either by excluding the whole file from the compilation or by modifying the PHP code to compilable form. To exclude the `.php` source file, it is necessary to head to the `libsConfig.json` file and create or modify the entry for the Symfony Component and its precise version. The `excludes` key of each entry holds a list of files that are to be excluded from the compilation. The `includes` collection can be then used to override default file exclusions from generated `.msbuildproj` and enforce their inclusion. Finally, with `ignoredDependencies` it is possible to resolve circular dependency errors that the dependency resolving mechanism of the `BuildGen` tool was not able to figure out.

After resolving each and every error, the `buildScript.ps1` successfully produces NuGet packages on execution and stores them in the `Nuget-Repository/Components` directory.

7.2 Deploying Symfony application

Using the tools and templates provided, Symfony application deployment on the .NET environment in general was simplified into the following summary.

At this point, we expect that all the Symfony Components required are already accessible as NuGet packages, either from included package repository that can be found under the `Nuget-Repository` directory of the repository, or via the custom Components compilation as discussed in the section before. We also expect the project templates from the `Peachpied.Symfony.Templates` NuGet package to be installed in advance with the following command:

```
dotnet new -i
Nuget-Repository/Peachpied.Symfony.Templates.4.2.3.nupkg
```

New Symfony application First of all, we have to set up two projects, one for .NET Core and the other for Symfony. Thanks to the `Peachpied.Symfony.Templates` NuGet package, this is as easy as running the

```
dotnet new syskeleton
```

command. A new solution titled `Symfony-Skeleton` with two projects - `Server` and `Skeleton`, will be generated in the working directory.

The `Server` directory consists only of a simple set-up for a Peachpied Symfony application running on Kestrel. Aside from the build configuration, the `Symfony` directory contains `.php` sources equal to the Symfony Skeleton 4.2.3 project with a basic routing and a style-sheets configuration.

Existing Symfony application Project creation for an existing Symfony application is the same as for a new one with the only difference in running the

```
dotnet new syemptyproj
```

command instead of the previously mentioned one. After that, Symfony sources of an existing application can be placed into the `Symfony` directory of generated solution.

This project template comes with no `PackageReference` configuration of Symfony Components, leaving the actual configuration up to the user. The best way to do this is to add references to all the Components listed in the `require` and `require-dev` fields of Symfony project's `composer.json`. Although, not all of the dependencies listed might be in fact required for a successful compilation. One such example that can be safely omitted from compilation is the `Symfony /Flex` Component.

Extending the application In order to use additional Symfony Components to the ones included by the `Peachpied.Symfony.Skeleton` NuGet package, it is necessary to add a `PackageRefference` to the particular NuGet package to the `Symfony` project's `.msbuildproj`. When installing a NuGet package to the project with Visual Studio IDE, this is done automatically. It is also necessary to run the

```
composer require [Component name](:[version])
```

command in order to update `composer.json` file that is parsed on project's compilation. As a side effect of this, the Symfony Component will be installed into the `vendor` directory. Having sources of Symfony Components inside of the `vendor` directory significantly speeds up compilations that perform Symfony cache generating. Symfony cache regenerating can be forced by deleting the `cache` directory, autoload scripts regenerating by deleting `vendor/autoload.php` and the `vendor/composer` folder. It might be necessary to do both after new Symfony Components are included.

In order to further extend the application, arbitrary number of custom routes, controllers and other Symfony ecosystem `.php` code can be included without the need to modify Symfony's `.msbuildproj` in any way. Symfony cache generating mechanism also keeps track of changes to Symfony application configuration and handles cache regenerating on build automatically.

7.3 Example 1: An application using Symfony and both Twig and Razor template engines

Usage of the Twig-Razor interoperability features will be demonstrated on the `Examples/Twig-Razor-Full-Page` application. In order to run this example, simply execute the

```
dotnet run
```

command from the `Examples/Twig-Razor-Full-Page/Server` directory. Doing this will start the application on `http://localhost:5004`. The first application start-up may take several minutes because all the Symfony Components need to be unpacked from the NuGet packages and the autoload scripts with cache have to be produced. Any subsequent application start-up completes within several seconds.

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 due to the absolute migration from PHP to .NET, because of merging of an existing Symfony project into an existing .NET one, or just to reuse existing Razor and Twig templates within your project, these features can save both time and money in all of the cases mentioned.

The application consists of five pages. The index page is defined within the Symfony project's `GeneralController` class and returns a static `.html` file that is used as a signpost of the application. This can be seen on the following Figure.

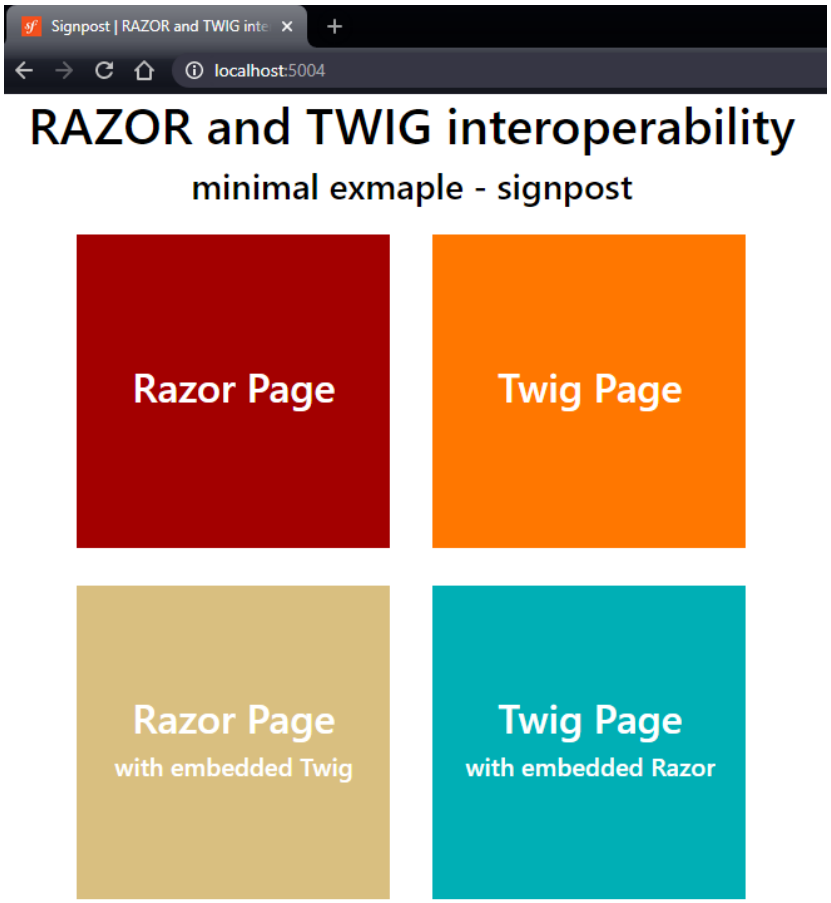

Figure 7.1: Example application's signpost

There are four locations being exposed inside of 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 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. Pictures of those two pages can be seen on Figures 7.2 and 7.3. For each element within the retrieved content, they provide a small calendar that is embedded using either Partial View, or via including additional Twig template. The remaining two pages take the advantage of the Razor-Twig interoperability features and embed Twig calendars within razor page and vice versa. Finally, Figure 7.4 shows both possible calendars in the layout used for the twig page.

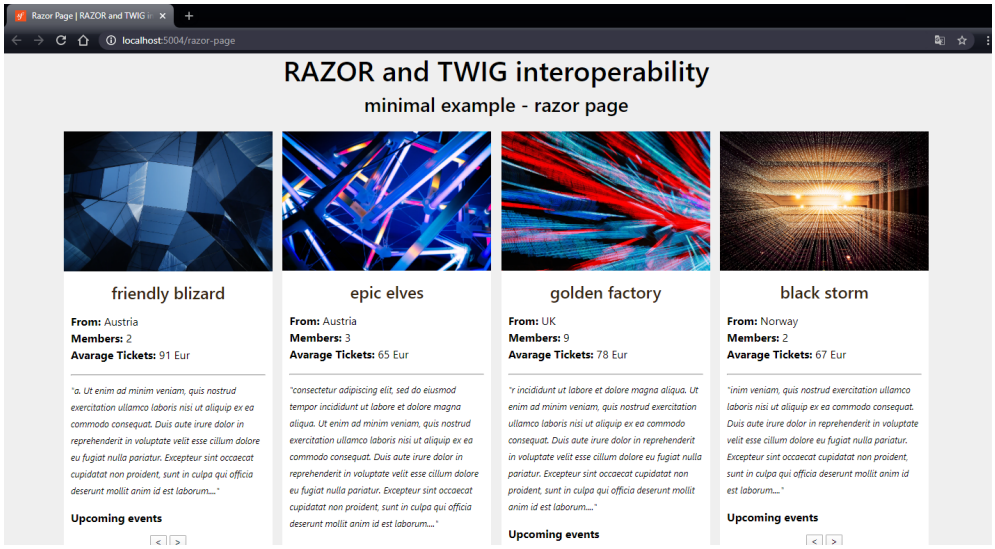

Figure 7.2: Example application's razor page

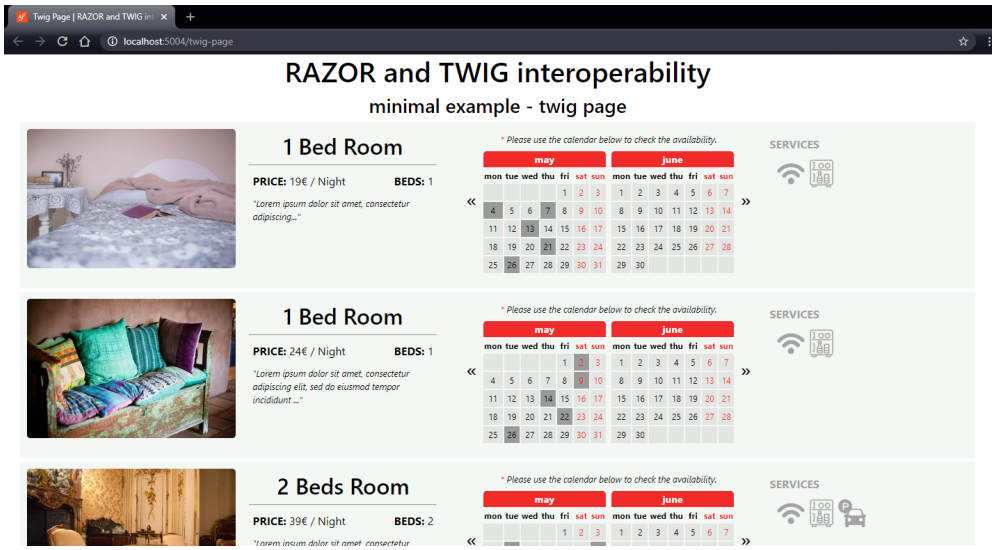

Figure 7.3: Example application's twig page

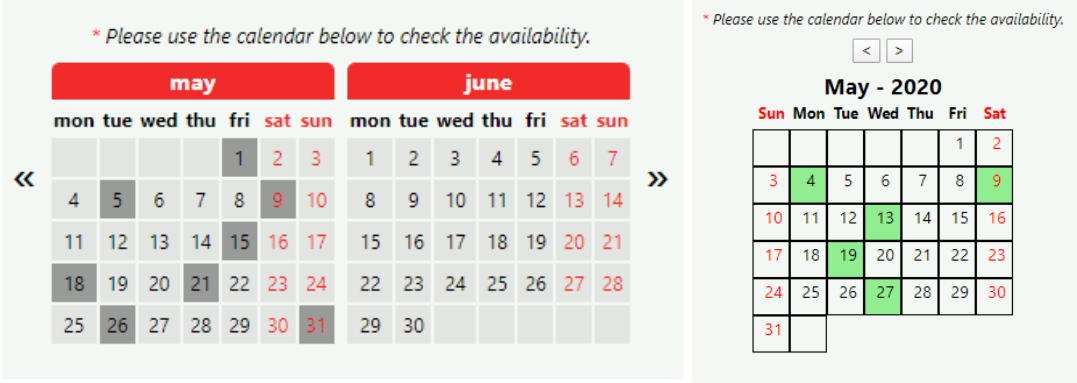

Figure 7.4: Variations of the example application's twig page calendars

Twig rendering Rendering of a Twig template from a 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 responsible for handling 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 a Peachpied Symfony project, the user must first add the `RenderRazorService` service. 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 or her 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 Symfony's `Environment` class and its only difference lays in additional `render_razor` function that is accessible within Twig templates rendered with it.

Now, 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 form and thus are expected to be supplied in the PHP native form.

7.4 Example 2: An application using Twig and Razor template engines without Symfony

Usage of the Twig template engine within Razor without using any actual Symfony project can be found in the `Examples/Twig-Razor-Only` application. Once again, the application is stared by running the

```
dotnet run
```

command from the `Examples/Twig-Razor-Only` directory. This will start the application again on `localhost:5004`. The first start-up takes considerably less amount of time than in the previous example because there is no need for any Symfony Components unpacking as there is no Symfony application included.

This example illustrates how easily can be the Twig rendering feature used from the plain ASP.NET Core application, without any unnecessary bonds to any particular Symfony project. This application uses `Razor Page` and `Twig In Razor Page` routes from the previous application, with no further modifications, revealing this feature's actual independence on Symfony. All the Twig templates are stored inside the `Templates` directory and referenced on the `RenderTwig` function invocation just as they would if they were stored inside the Symfony project. The previous example's `index.html` was moved into the `Server` project

and modified in order to satisfy this application's needs. Furthermore, Twig calendar styles were brought into the `Server` project as well.

Conclusion

In this thesis, we examined possibilities of enhancing the process of compilation of Symfony application into the .NET environment with the Peachpie compiler. Then we proposed custom and more efficient strategy of compiling Symfony applications and introduced several tools that represent a possible way of how can these ideas be implemented. After that, we denoted an approach of simple and user-friendly embedding of the product of the previous process into an ASP.NET Core application with the `Peachpied.Symfony.AspNetCore` NuGet package. Finally, we presented possible interoperability features that the whole process of Symfony application compilation yields, with Twig and Razor templating functionalities provided by the `Templating` namespace of the `Peachpied.Symfony.AspNetCore` NuGet package.

Future work

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 non-optimal way. Therefore, possible future enhancements and refactorizations will be discussed ahead.

Package references One of the biggest opportunities for improvement of the current solution lays in the way Symfony dependencies are converted to the package references. At the moment, it is necessary to manually set-up the configuration which still slightly spoils the user experience.

PHP dependence The second flaw of the solution is its partial dependence on PHP and the Composer package manager. The best way to address the issue with the Composer would be compiling the whole Composer software into .NET using the Peachpie compiler. This would also enable Composer to use the PHP classes directly from the packages, therefore, save considerable amount of time and space consumed by their current copying from packages to projects. Furthermore, the cache generating system is based on the PHP as well. The Peachpie compiler provides a way for the PHP cache generating in general, however, this mechanism is not fully usable with Symfony projects. It would either require additional work possibly from both the Peachpie dev team and us, or we might be forced to implement our own parser of Symfony's `.php` config files and use that from within the compiled `Kernel.php`.

Interoperability usability 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 who would want to use only the feature to embed the Twig template within their Razor pages. In addition to, the former purpose of the `PeachPied.Symfony.AspNetCore` NuGet package 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.

Packages availability Finally, the NuGet packages are currently stored locally and thus require each user to either download them from project's repository or to compile them manually prior to their usage. Even though they can be generated automatically with the `Peachpied.Symfony.BuildGen` tool, this process is still far from optimal solution. Moving the pre-compiled packages into a remote repository would be a step into the right direction, possibly enabling community to supply missing packages and thus ensure faster advance.

Bibliography

[1] W3Techs. *Usage of server-side programming languages for websites*. https://w3techs.com/technologies/overview/programming_language/all.

[2] BuiltWith Pty Ltd. *Framework Usage Distribution in the Top 1 Million Sites*. https://trends.builtwith.com/framework.

[3] Josh Lockhart. *Modern PHP - New Features and Good Practices*. O'Reilly, 2015.

[4] Scott Guelich, Shishir Gundavaram, and Gunther Birznieks. *CGI programming with Perl - creating dynamic web pages (2. ed.)*. O'Reilly, 2000.

[5] The PHP Group. *PHP Manual*. https://www.php.net/manual/en.

[6] Paul Tarjan and Sara Golemon. *OSCON - Open Source Convention (Include Hack - HHVM - PHP++)*. O'Reilly, Portland, OR, June 2014. https://www.youtube.com/watch?v=JrPGa1JDX38.

[7] The PHP Group. *PHP Language Specification*. https://github.com/php/php-langspec.

[8] Andrei Zmievski. *PHPCOMCON - PHP Community Conference (The Good, The Bad, And The Urly: What Happened to Unicode and PHP 6)*. Nashville, TN, April 2011. https://www.slideshare.net/andreizm/the-good-the-bad-and-the-ugly-what-happened-to-unicode-and-php-6.

[9] Julien Pauli, Nikita Popov, and Anthony Ferrara. *PHP Internals Book*. http://www.phpinternalsbook.com/.

[10] The PHP Group. *The PHP.net wiki*. https://wiki.php.net/start.

[11] George Schlossnagle. *Advanced PHP Programming*. Sams Publishing, 2004.

[12] Nils Adermann and Jordi Boggiano. *Composer Documentation*. https://getcomposer.org/doc/.

[13] SensioLabs. *Symfoy - The Reference Book*. https://symfony.com/pdf/Symfony_reference_4.2.pdf.

[14] Fabien Potencier. *Twig Documentation*. https://twig.symfony.com/doc/2.x/.

[15] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 6 edition, June 2012.

[16] Microsoft. *.NET Documentation*. https://docs.microsoft.com/en-us/dotnet/.

[17] Andrew Troelsen and Philip Japikse. *Pro C# 7: With .NET and .NET Core*. Apress, 8th edition, 2017.

[18] David McCandless. *Knowledge is Beautiful*. HarperCollins Publishers, 2014.

[19] iolevel. *Peachpie, Your bridge between PHP and .NET*. `https://www.peachpie.io/`.

[20] Jan Benda, Tomas Matousek, and Ladislav Prosek. *.NET Technologies 2006 - Phalanger: Compiling and Running PHP Applications on the Microsoft .NET Platform*. University of West Bohemia, Plzen, Czech Republic, May-June 2006. `http://oot.zcu.cz/NET_2006/Papers_2006/full/A37-full.pdf`.

[21] iolevel. *PeachPie Compiler Platform*. `https://github.com/peachpiecompiler`.

List of Figures

List of Listings

List of Abbreviations

PHP	PHP: Hypertext Preprocessor
CMS	Content Management System
CGI	Common Gateway Interface
VM	Virtual Machine
API	Application programming interface
OOP	Object Oriented Programming
SDK	Software Development Kit
CLI	Common Language Infrastructure
CLS	Common Language Specification
VES	Virtual Execution System
CTS	Common Type System
CIL	Common Intermediate Language
IL	Intermediate Language
DLL	Dynamic-link library
EXE	Executable
JIT	Just-In-Time
MVC	Model-View-Controller
DI	Dependency Injection
DRY	Don't Repeat Yourself
IDE	Integrated Development Environment
UX	User Experience

Attachments

A. Project repository

Content and structure of project repository attached to this thesis:

- **Nuget.config** — Configures local NuGet repository.
- **global.json** — Globally controls used Peachpie SDK version.
- **libsConfig.json** — Holds per-Component build configuration.
- **favicon.png** — Icon used as example applications' favicon.
- **README.md** — Repository's readme file.
- **Docs/** — Contains electronic version of the thesis's text.
- **Examples/** — Contains projects demonstrating possible usage.
- **Libs/** — Contains `.dll`s used for Components building.
- **Nuget-Repository/** — Local NuGet repository.
- **Peachpied.Symfony/**
  - **README.md** — Folder's readme file.
  - **AspNetCore/** — Provides API for the whole functionality.
  - **Components/4.2.3/** — Folder with Symfony 4.2.3 Components.
  - **Empty-Page/** — Simple .NET Core application with Symfony.
  - **Skeletons/4.2.3/** — Project bundling Symfony 4.2.3 NuGets.
  - **Templates/** — Provides project templates.
- **Props/** — Holds collective build settings for Components.
- **Tools/** — Contains sources of tools responsible for automatization of compilation processes.