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

Dedication.

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: Abstract.

Keywords: C# PHP ASP.NET PeachPie Symfony

Contents

Introduction

Present world of web development is divided into several nearly disjoint communities. Each of these communities focus primarily on different aspect of development, for instance security, reliability or simplicity. However, the emphasized aspect of development is actually not responsible for the separation of these communities. It is more of a consequence of the actual divider, which is the programming language selected for the implementation. Speaking of programming language chosen for web development we implicitly mean the server-side programming languages as the thesis is focuses mainly on them.

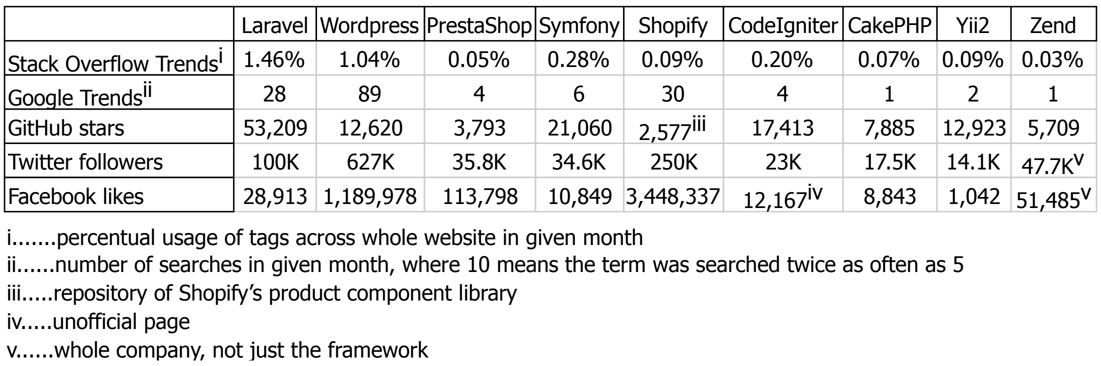
When it comes down to numbers, PHP: Hypertext Preprocessor (PHP) is the ultimate winner with the percentual usage between the websites surveyed of 60 to 70%. The second place holds ASP.NET with 10% - 20% of usage. The third and fourth places are occupied by Java and Ruby, both situated around the border 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 over the half of all surveyed, and presumably the half of all existing, web applications. Moreover, unifying the two technologies would benefit both communities. Where PHP community could make use of security and manageability advantages of pre-compiled code, .NET developers will be given means to use dynamic features of PHP language and a vast of community libraries. This is the ultimate goal of Peachpie, modern PHP to .NET compiler, that is currently in development.

However, because PHP is a general purpose programming language, many web oriented PHP frameworks have been developed, utilizing many of its core functionalities and providing them in a more abstract way, making the process of web application deployment faster and thus less expensive. Due to lack of any reliable statistics on 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 at least correspond to their actual usage. Gathered data are displayed in the Figure 0.1 and apply to 27/6/2019. 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. Therefore, each one requires special treatment during its compilation, making their automatic compilation impossible. Peachpie developers has successfully compiled Wordpress framework. Shopify is not open-source and thus its source code is not simply accessible. On the other hand, Laravel is open-source, and the same goes for 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 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 Symfony framework into .NET platform. However, the Peachpie compiler provides just the tools for 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 precompiled and supplied to projects in the quantity that they require, rather than to compile the project and all of its libraries for each project over and over again. For this task, NuGet package manager has been selected as it shares the same ideology with Composer, the package manager that is tightly entangled with the Symfony framework. The mentioned ideology is that is of focusing mainly on package and dependency management at the development-project level. Besides, its wide spread of usage, Visual Studio integration ability and fairly easy usability were also taken into accord.



	Laravel	Wordpress	PrestaShop	Symfony	Shopify	CodeIgniter	CakePHP	Yii2	Zend
Stack Overflow Trends[i]	1.46%	1.04%	0.05%	0.28%	0.09%	0.20%	0.07%	0.09%	0.03%
Google Trends[ii]	28	89	4	6	30	4	1	2	1
GitHub stars	53,209	12,620	3,793	21,060	2,577[iii]	17,413	7,885	12,923	5,709
Twitter followers	100K	627K	35.8K	34.6K	250K	23K	17.5K	14.1K	47.7K[v]
Facebook likes	28,913	1,189,978	113,798	10,849	3,448,337	12,167[iv]	8,843	1,042	51,485[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

An overview of the thesis

Chapter 1: PHP language The first chapter will provide an introduction to 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 Symfony framework, which will be discussed in the following part. The chapter ends with an explanation of TWIG engine's syntax and usage.

Chapter 2: C# language The goal of the second chapter is to introduce C# programming language and .NET platform. The following part of the chapter will focus on 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.

Chapter 3: PHP to .NET compilation The third chapter talks 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. Finally Peachpie, an actual implementation of previously profiled tool will be presented.

Chapter 4: Problem analysis Chapter 4 describes the general 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 possible alternatives to each selected technology and explains reasons for each choice.

Chapter 5: Peachpied Symfony

1. PHP language

PHP is a general-purpose scripting language. It originated as a collection of Common Gateway Interface (CGI) scripts, written in PERL by Rasmus Lerdorf [3]. 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 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 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, PHP community and the PHP Group [6, 7]. The PHP group is a comunity of developers that maintain both, PHP language repository on github and 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]. 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 being developed and its release has been announced for later half of November 2019 [9]. In the subsequent text, when speaking of PHP language without explicit version number mentioning, PHP version 7.3 is meant, as it is the latest stable released version.

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 integration with other programming languages without the need to be compiled. PHP scripts are executed by the Zend Engine, an open-source set of components than provide PHP language with tools for its procession and execution [10, 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 PHP version 3 [5]. Since then, 2 new versions of Zend Engine has been introduced, each time bringing new major version of PHP language [5, 9].

The Zend Engine Undoubtedly, the most important component of the Zend Engine is the Zend Virtual Machine (VM), responsible for interpretation of PHP script [10]. 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 resolving simple function calls with literal arguments, 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 Zend extension 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 PHP application's execution is also illustrated in the following Figure.

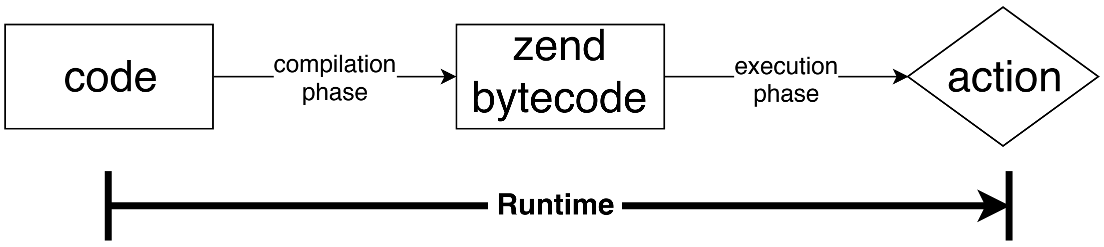


Figure 1.1: PHP's execution

1.1 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 PHP code, possibly bringing higher level of organization and reducing maintenance burden. PHP supports all expected constructs of OOP such as Classes, Interfaces, Constructors and Inheritance [5]. On a class instantiation, `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 PHP programming language is considered to be dynamically typed. This means that the language implicitly performs conversion between two types from different families, such as numbers and strings. This approach provides the language with additional 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 a `string` data type knowing, that the implicit conversion between the types will result in a `string 42`. On the other hand, an inexperienced developer might succeed 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 the 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 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 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 boundings, to be declared and checked.

Special place between PHP data structures hold arrays. Internally, arrays are represented as ordered maps [5]. When provided, arrays use custom string or integer values as a 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 a valid integer number are implicitly converted to integer keys [5].

Variables When a variable is declared by a standard syntactic mean of $ sign followed by the 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 requite 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 `unset` method for 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 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 data provided by the request, like submitted form fields and URL encoded parameters [5].

Futhermore, PHP provides means to easily create 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 setts it as one of 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].

1.2 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 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, version, 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]. Where some commands, like install or update, does 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 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 contents 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 a `readme.md`, as 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 vendor directory.

1.3 Symfony framework

Symfony is a PHP framework for web applications 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 made birth to many web applications such as Drupal or Laravel, 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]. At the time of writing, Symfony 4.2 is the latest stable version of the framework released, for this reason, further description would primarily focus on this version.

Installation Symfony application is created via composer, more precisely `create-project` command with either `symfony/website-skeleton` or `Symfony/skeleton` argument [13]. When executed, composer downloads predefined set of core libraries that handle base framework functionality [13]. After that, everything from routing set-up to additional libraries downloading with `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`, that 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 controllers 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 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].

1.4 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 TWIG code and three special delimiters.

Output delimiter The first one is `{{...}}`, which 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, where other are much more complex, like include, resulting in included TWIG template loading, generating and embedding the result in the current template [14]. Every expression can be suffixed with arbitrary number of `|[filterName]` expressions to further process 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 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 further 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 to build a template acting as a basis that child templates further extend. Such template may contain all the common elements of a site - header, footer and a navigation bar. Furthermore, it can define blocks with `{% block - %}...{% endblock %}` syntax, whose content can be either fully overwritten by child templates or combined with child's one via `{{ parent() }}` function [14]. Inheritance is enabled for selected templates by specifying a parent template with `{% extends - %}` construct in the beginning of each child template [14].

Environment Environment is a 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\LoaderIn` 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 HTML markup [14]. The `templateWrapper` class also provides a self-explanatory method `renderBlock` [14].

2. C# language

C# is a general-purpose programming language, under the management of Microsoft Corporation. It is a part of the 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 .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 7 major versions of C# programming language released [16]. Although some of C# version 8 features has been already published, this version has not been released yet [16]. Because of that, when speaking of C# we implicitly mean C# version 7, 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, EXE file is generated, DLL file is generated otherwise [17]. Exception to this behaviour brought .NET Core, which makes it possible to generate DLL files for both, 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 .NET APIs that are intended to be available on all .NET implementations [16]. Each version of 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 C# application's deployment from compilation to execution is summed up in the following Figure.

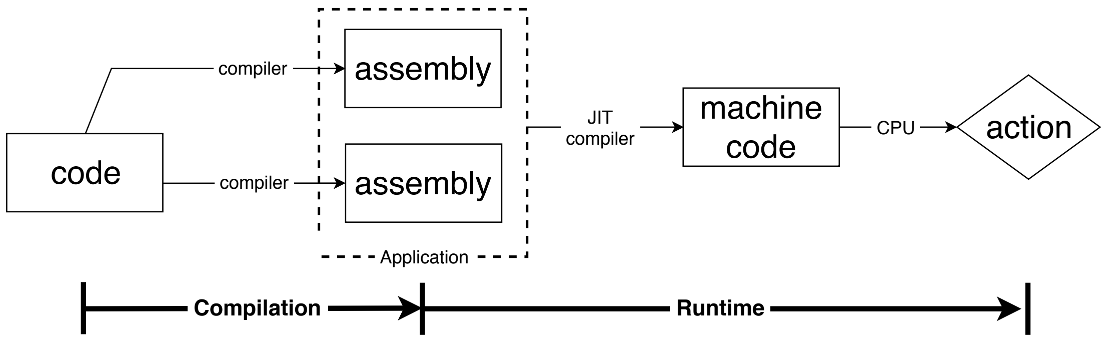


Figure 2.1: C# application's deployment

2.1 Language features

Object Oriented Programming Main characteristic of programming in C# language is that it is object-oriented. An object is basically a block of memory that has been allocated and configured according to the blueprint, also referred to as an instance [16]. These blueprints mentioned are nothing more than elements of C#'s type system.

Type system Types in C# are divided into two main categories, value types and reference types. In contrary to value types, that directly hold their data, reference types hold only references to objects. There is also a type called an object type. 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 base object type [16]. The point where dynamic type differs from base object type is in the amount of information compiler has about the type, and in the way type is processed on runtime [16]. An example illustrating both those differences is casting [16]. Dynamic variables does 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, 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 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].

2.2 NuGet

NuGet is Microsoft-supported mechanism for creating, sharing and consuming code under the .NET environment [16]. For this purpose NuGet defines a package, 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 a `.zip` archive of mentioned three groups of files with `.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 a 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, content of `.nupkg` archive is simply copied to user-specific package folder [16]. Finally, project files are updated and any missing package dependencies are installed [16].

2.3 ASP.NET Core

ASP.NET Core is a framework focused on building web applications, based on .NET Core [16]. Fundamentals of ASP.NET Core lies 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, placed in the order regarding the 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 a HTTP server implementation, middleware components, logging, dependency injection and configuration by a single object, in order to simplify application start up and shut down control [16]. ASP.NET Core provides the `WebHost` class with `CreateDefaultBuilder` method to set up a host with commonly used options, making most of the hard job in developer's stead [16].

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 Mentioned middleware component, also called request delegate, can be one of built-in components or custom one [16]. Each component is added to the pipeline in `Startup.Configure` method by invoking its `Use`, `Run` or `Map` extension method [16]. By convention, `Use[middlewareName]` method decides whether to invoke the next middleware component in the pipeline [16]. `Run[middlewareName]` methods runs at the end of the pipeline, terminating the pipeline after its processing [16]. Finally, `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 the SessionMiddleware to enable session state for the application [16]. The `UsePhp` method is extension of Kestrel provided by PeachPie software and will be spoken of later. Remaining `UseDefaultFiles` and `UseStaticFiles` handle file mappings and 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].

Concept of the Model-View-Controller pattern lays in separation of application logic into three parts, kept in different folders, 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 an 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 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]. User-defined services are registered in the `Startup.ConfigureServices` method [16]. New classes can be registered as services to the `IServiceCollection` provided as a parameter to the function to be further available along with predefined framework-provided 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 the `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 required services [16]. This process is called the Dependency Injection (DI) and it is a practise from a design pattern with a same name.

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

2.4 Razor

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

HTML rendering is controlled via @ 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 single space-less expression, called implicit expression, or spaces supporting explicit expression [16]. Implicit expressions are created as @ symbol directly followed by single space-less C# statement, usually returning desired output [16]. Explicit expressions can consist of several C# statements, combining their outputs into 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 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 @ symbol, @: directive or `<text>` tag [16].

In order to follow DRY 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, in 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 testability and code reusability.

Process responsible for determining which view file is used is called view discovery [16]. Actions initiate this process through calling 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` by calling either the `AddMvc` or the `AddMvcCore` extension method of the `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 a 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 used directive [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 connected with the `PageModel` classes via 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 `.cs` suffix [16].

3. PHP to .NET compilation

In recent days, web development, and software development in general, has evolved by a great bit in both the number of people involved and in a 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 of code [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 we share our knowledge and learn from knowledge of others. Whether some patterns and practices are applicable in general, with increasing complexity of problem their amount is smaller and their 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.

3.1 Motivation

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. Simply speaking, compilation means that we have a language A, that ordinary compiles into an intermediate language and is later executed via provided tools. 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 language B within our application written in language A. When 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 using features, knowledge and finally the developers of both languages in a single project [19].

For later usage, we would require a 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]. Finally, its assembly system allows for splitting 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 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 area of usage. When a problem in a certain language is encountered, solutions in other languages are mostly useless. This is 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 the process are mentioned means for achieving the compilation. Prior to learning how to operate with the compiler tool, we first need to 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 languages' type systems - one is statically typed, where the other one is dynamically typed, this difference will introduce no noticeable difficulties. PHP's engine is using types internally and those basic types are subset of the CTS [10, 15]. In addition to, PHP has no sense of topology in its type system [5]. The types simply exists without any base types or inherited functions [5]. This makes their mapping to .NET types and into its type hierarchy rather easy.

The only type-related complication is PHP's implicit conversion between the types. However, this problem can be solved easily by adding explicit conversions, implementing special functions that emulate PHP's implicit conversions between the types or for the sake of expensiveness using 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]. Based on each variable's usage, and for a sake of definitive assignment it can be simply initialized to null if needed. This operation will satisfy C#'s variable requirements and won't 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 as regarding PHP application's execution, such process would require compiler to transform C# code to native PHP code. On the other hand, in out case, we want to compile code of interpreted language to a code of compiled language. Although converting PHP code directly to C# code is also a solution, it is rather poor one as there would be two compilations in a process of deploying such application which would bring significant overhead.

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

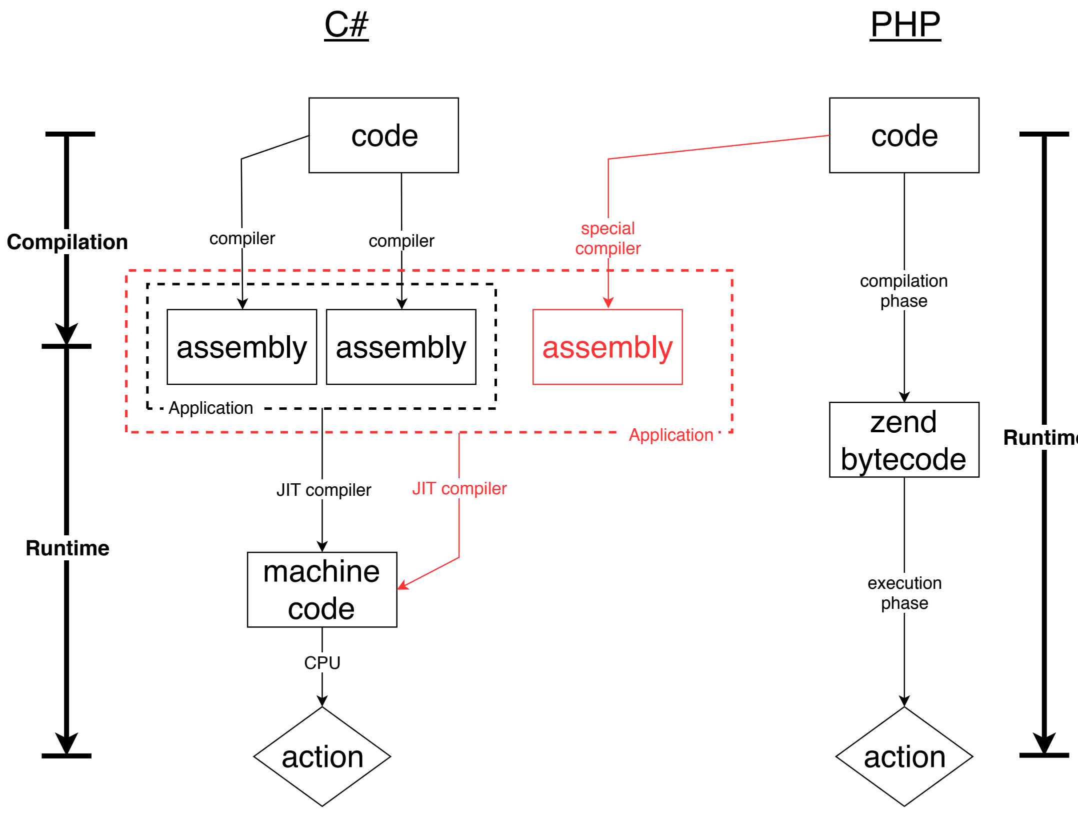


Figure 3.1: PHP to C# deployment comparison

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 runtime exception when executed. It is obvious that such feature can't 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.

The second consequence does not originate from anything mentioned but is more of an architectural nature. C# does require whole `finally` block to be executed and for this purpose forbids statements redirecting the execution flow inside [16]. On the other hand, PHP guarantees only `finally` block entering, 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 unsupported constructs, or to exclude them if the former is not possible.

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

Last but not least stands the problem of standardization. Because 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 compilation and presenting some of possible solutions for the each of them, we can finally proceed to actual compiler tool, Peachpie.

Peachpie is a modern PHP compiler based on Roslyn, a set of open-source compilers and code analyst APIs, also called .NET compiler platform [19, 16]. Peachpie took heavy inspiration in its predecessor Phalanger, the PHP language compiler for the .NET Framework targeting 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 allows for PHP applications to be developed directly in 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 legacy PHP code into .NET, at which it ultimately succeeded. However, its development slowed down through out the time and finally stopped at 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 compilation and 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 PHP compiler for .NET and Peachpie's development was initiated [19].

The compiler's development continues to 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 7.2, using it within .NET Core SDK 2.1 or newer and efforts are made towards compatibility with upcoming .NET Core 3.0 [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 PHP version 7.2.

Much more promising benefits of using Peachpie are 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 present 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

The goal of this thesis is to bring interoperability between selected PHP framework and .NET platform through compilation. Moreover, for the purpose of intuitive usage and simple further extensibility, compiled framework is to seamlessly utilize NuGet package manager.

- Jaky rozhodnuti se budou delat

5. Peachpied Symfony

In regards to the naming of the project, we decided to hold to the convention laid down by the previous project made by the PeachPie developers, where the objective was to compile Wordpress framework into .NET. The resulting project was named Peachpied Wordpress, therefore, this project was entitled Peachpied Symfony.

5.1 Architecture

Symfony strucure As was mentioned before, each Symfony project is made up from libraries, called Symfony components. All these libraries reside in the vendor folder of the project. The rest of the project contains mainly settings, such as routing definition, environmental variables set-up or database credentials, and user defined content, such as Controller classes that handle requests, images, definition for styling and front end scripts. Aside from this, every Symfony project also contains a Kernel class and an autoload file. Those two files are responsible for the proper load-up of all of the remaining libraries included in the vendor folder.

Therefore, one can perceive the Symfony project as two completely distinct parts. One, the vendor directory containing all of the libraries that the project uses, and two, excluding mentioned autoload file that is located in here, and two, basically the rest of the project folder. For the purpose of further referencing, we will name the second denoted part the kernel part. Ultimately, for set of Symfony components and fixed versions, the vendor folder will always look the same. Moreover, each of those Symfony components can be recognized as an independent unit as well, only holding references to another libraries that it requires for its proper functioning. For this purpose, Symfony components are making use of Composer tool. Each component contains `composer.json` file specifying required packages that it is dependant on. Thus, when a desired component is required via composer, the whole dependency tree of components will be actually installed.

Taken this inner composition into account, we decided that splitting each of the Symfony components into separate project and on the Symfony project compilation, actually compile just the kernel part, would bring the most desirable outcome. By this approach, we will be able to supply the project with pre-compiled libraries, the NuGet packages, of required versions while saving the compilation time. Furthermore, by storing the compiled packages in a publicly accessible repository, it would give us the possibility to supervise the compiled packages to make sure that every package is fully functional before it is supplied to the user.

One drawback of introduced approach is the necessity of significant number of packages to be pre-compiled in order to be able to comply the requirements of the minimal Symfony 4.2 project, created with the `composer create-project Symfony/Skeleton [projectName] [<version>]` command. As the command suggests, such project is called `Symfony/Skeleton`. It consists of 24 components and presents the first building block of all web applications running on Symfony

4.2. Nevertheless, this problem can be solved very efficiently when we take the advantage of each component's `composer.json` file that denotes its dependencies. When combined with common compilation settings isolation and reusing, those packages can be compiled fairly easily.

Extensibility Finally, further extensibility was granted, as bringing support to the new component requires nothing more, than to include its compiled NuGet package in the repository. This process could even be semi-automatized as the general compilation settings can be reused from already compiled libraries and dependencies can be taken from the `composer.json` file, leaving on the programmer only possible additional files exclusions or other PHP to .NET language compilation related problems.

5.2 Compilation

For the package pre-compilation task, we will prepare empty `Symfony/Skeleton` project with the `composer create-project Symfony/Skeleton empty-page 4.2` command.

MSBuild When a .NET project is compiled, MSBuild, the .NET compiler, takes the configuration stored in `.[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]. Basically used extensions are `.csproj` for C# projects, `.vbproj` for Visual Basic .NET projects, and `.msbuildproj` for general projects without bond to specific langauge [16]. The `.msbuildproj` file extension is used internally by peachpie Software Development Kit (SDK). In addition to, `.target` and `.props` extensions are recognized as a 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 MSBuild XML schema uses properties and items as a way of passing information to tasks. Where properties are name value pairs, items can contain attributes [16]. There are many additional differences between properties and tasks their typical usage, the way they are passed to tasks and the way they are processed [16]. However, these reasons are not relevant to further text and therefore will be omitted. MSBuild reserves some property names as a mean of storing information about the project file [16]. Those properties can be referenced by using the `$([propertyName])` notation. 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.

Solution design For each library, we need to create a project, more precisely, we need a `.proj` file that will be supplied to the MSBuild task. First, to include Peachpie compiler, we need to set the `Sdk` attribute of the root `Project` element to `Peachpie.NET.Sdk/[version]`. This way, properties and targets used by 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 statically to the output package. For this purpose items `Compile` and `Content` and their attributes `Include` and `Exclude` are used. When MSBuild copies the static files to the output, it preserves the structure the files are located in. As the NuGet packages will be used and referenced from the kernel project, it is preferable to place the `.msbuildproj` files for each project to the root of Symfony project, as 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 seem like a great amount of repetitive code. For the sake of DRY principle, we can make use of `.props` and `.target` files. Either can be used, as MSBuild does not really pay much attention to the actual extension of a file, but to address the conventional usage, we will create `.props` file, as those are primary used to define reusable properties, which is exactly what we are going to do `microsoft.docs`.

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 less significant, but still required, properties that we need to set-up as well to produce the correct output. These properties specify the output type of the project, targeted framework, project description, name of both the project and the final package, and necessary routes where to output NuGet package, intermediate path for configuration-specific intermediate output and the path to the output directory. All of these properties will be fairly similar for all of the packages, differing only in the names and paths.

To be able to fully isolate previously mentioned properties into a separate file, we denote a naming convention for the `.proj` files. This way, we will be able to use the project name to derive a path to the actual library folder. MSBuild provides one reserved property that fits out needs, that is `MSBuildProjectName`, holding the file name of the project without the extension. To detach custom properties creation from the actual compiler used properties set-up, we create another `.props` file, that will be used solely for the purpose of further reusable custom properties definitions. The two values that will differ between the projects are libraries' names and locations. Hence, we will create two variables `LibraryName` and `LibraryPath`. The first one will be used as the output package name and the second one as the path to the input library path. 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 simply replace each backslash delimiter on the route from vendor folder to the library folder with dots. Extended with additional property that provides a background conversion from the dot notation to the backslash route we have so far a fully automatable process.

Directory.Build.props The background conversion mentioned can be performed in `Directory.Build.props` file. This file is optional and should be located at the root folder of the project. Its content is imported automatically in an early stage of compilation, making it capable of overwriting properties defined in most of the build logic [16]. For the purpose of increased organisation we will define our conversion logic there, introducing `ProjectNameLocation` property as a result of given conversion from the dots to the backslashes. Additionally, values for the `BaseIntermediateOutputPath` and `BaseOutputPath` needs to be set there as they are later used for calculating routes used for storing NuGet related data [16]. The actual code that handles following can be seen in the subsequent Listing.

```xml
<?xml version="1.0" encoding="utf-8"?>
<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

With the use of defined `ProjectNameLocation` property, the `LibraryName` and `LibraryLocation` properties definitions will go as shown by the next Listing.

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

    <PropertyGroup>
        <LibraryName>$(MSBuildProjectName)</LibraryName>
        <LibraryPath>
            vendor\$(ProjectNameLocation.toLower())
        </LibraryPath>
    </PropertyGroup>

</Project>
```

Listing 5.2: LoadPropsConfig.props

Before proceeding to the build properties file, we need to mention that PeachPie compiler provides a feature of `composer.json` file parsing and does that automatically. Because this feature is unnecessary at the moment and would collide with our custom references set in regards to encountered complications, we need to provide PeachPie compiler with an invalid route to `composer.json`. This is done by setting the `ComposerJsonPath` property to any route not leading to Symfony project's `composer.json`. Now for the build properties `.props` file, it will be organized as follows.

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

    <PropertyGroup>
```

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

</Project>
```

Listing 5.3: BuildPropsConfig.props

Compilation content Now, that we have successfully set up all the directories and names used on compilation, we can move on to the definition of the content to be compiled. For that, we create another `.props` file that will hold together our compile and content elements. We do so, as it can be expected 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 remaining files as content. However, we can expect that some projects will require additional files exclusions that others won't, therefore, we will introduce the `AdditionalExcludes` property, defined in each project's `.msbuildproj` file, specifying its possible extra exclusions. For now, we will skip this `.props` file definition as it will be extended in reaction to encountered complications that will be further discussed.

Finally, we have to set up dependencies. Those are project specific, and thus will be located directly in each project's `.msbuildproj` file. This part can be effectively automated as well with sufficient `composer.json` file parsing, as every Symfony component includes this file when required. The `JSON` structure is an associative array that was developed in a way that it could be easily processed and parsed by machines. Hence, it is very simple for us to access the `"require"` key that contains object with names of libraries that the components is dependant on. For this reason, one can create simple mechanism that will convert content of this given object into a set of `PackageReference` elements and append it to the end of project's `.msbuildproj` file.

Complications With accordingly prepared `.props` files and `.msbuildproj` project file, we could theoretically proceed to the actual compilation, however, there are still two details we need to take into account before we can accomplish a compilation. The first one are discussed features of one language, in our case PHP, that simply cannot be brought into the world of the .NET. The most common and, when attempting for a compilation in the current state, also the most frequent, error is an inclusion of non existing class or interface. In PHP 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. For this reason, we can either 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 typical user does not perform downloaded libraries testing, therefore, their inclusion would cause unnecessary overhead in both the space required for storing those additional libraries and in time taken for the project compilation. Moreover, the test files are also the only files we can generally exclude from every project, but at the same time, in most cases the only files required to be excluded in order to achieve a successful compilation. For this reason, we set up the remaining third `.props` file containing compilation specifications as shown in the following Listing.

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

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

            <PackagePath>
                contentFiles\any\netcoreapp2.0\symfony\
                $(ProjectNameLocation)
            </PackagePath>
            <PackageCopyToOutput>true</PackageCopyToOutput>
            <Link>
                symfony\$(ProjectNameLocation)%(RecursiveDir)
                %(Filename)%(Extension)
            </Link>
        </Content>
    </ItemGroup>

</Project>
```

Listing 5.4: CompileContentConfig.props

Another much less frequently occurring cause for this is forgotten dependency. In practise, this shouldn't happen as such will cause the original PHP project to crash as well. However, in some cases, PHP might 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 can't be compiled. Again, there are two solutions to this problem and the exclusion has yet again proved to be better one. In order to follow the way the application is executed in PHP, we shouldn't be bringing in additional explicitly unrequested functionality.

The second group of details that are yet to be addressed are language constructs that PHP supports and .NET does not. Because these will usually appear in a not excludable parts of the code it is not possible to avoid using these files and the only remaining option is to edit the code to functionally similar but compilable version. In addition to, as PeachPie compiler is still work in progress, a missing functionality can be expected to be encountered, therefore, resulting in the need for additional code edits. Mentioned edits that had to be performed in order to achieve compilable code are as described in the following paragraph.

Required code edits Several times, Symfony components used `yield` keyword inside a `try-catch` block. Such operation cannot be performed under the .NET environment as it places further restrictions on usage of iterators inside these blocks. These cases have been rewritten into equivalent form with the use of arrays and booleans used for signalization. Furthermore, as such edits can be automatized, PeachPie developers have implemented features that handle these compile errors automatically.

The second encountered compilation errors were caused by inappropriate usage of _`toString()` function. Symfony classes occasionally displayed a possibility to throw an exception in this function, which is from .NET's point of view invalid. This case was for the sake of further progress temporarily rewritten to empty string returning and later resolved in another PeachPie compiler update.

Another 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 `return` statement from the `finally` block of exception handling construct. However, this component is used solely as a enhancement of Composer tool, and thus, for the purpose of requiring additional libraries. Compiled Symfony project has no use for such functionality as requiring additional packages from compiled project via composer and its hooked flex is redundant with the introduction of NuGet packages.

The last type of compilation errors were caused by missing implementation of small set of features. Unfortunately, this couldn't be helped as the Peachpie compiler is still in the phase of development. As some of these missing features we can name missing implementation of`SessionUpdateTimestampHandlerInterface` and
.

Result Final empty-page project structure consists of more than 20 `.msbuildproj` project files located in the `Symfony/Skeleton` project's root. Each of them implicitly uses default properties from `Directory.Build.props` and includes three additional `.props` files located in the `Props` folder. Furthermore, each of these `.msbuildproj` files optionally sets the `AdditionalExcludes` property that is further used in the `.props` file handling specification of files to be compiled and included. Ultimately, each of the project files can contain references to other compiled packages in a form of arbitrary number of the `PackageReference` elements. Hence, the final `.msbuildproj` file for the `framework-bundle` library, located in the `vendor\symfony\framework-bundle` folder does look as depicted in the next Listing.

```
<Project Sdk="Peachpie.NET.Sdk/1.0.0-appv2570">

  <!-- Sets package-specific properties based on name and path -->
  <Import Project="..\props\loadPropsConfig.props" />

  <!-- Configurates build properties packageId and AssemblyName -->
  <Import Project="..\props\buildPropsConfig.props" />

  <PropertyGroup>
    <AdditionalExcludes>
      $(LibraryPath)Client.php;
      $(LibraryPath)command\XliffLintCommand.php;
    </AdditionalExcludes>
  </PropertyGroup>

  <!-- Adds compile and content items node.
       Adds additional excludes to content. -->
  <Import Project="..\props\compileContentConfig.props" />

  <ItemGroup>
    <PackageReference Include="Symfony.Cache" Version="1.0.0" />
    <PackageReference Include="Symfony.Console" Version="1.0.0" />
    <PackageReference Include="Symfony.Routing" Version="1.0.0" />
    <PackageReference Include="Symfony.Templating" Version="1.0.0" />
    <PackageReference Include="Symfony.Translation" Version="1.0.0" />
    <PackageReference Include="Symfony.Yaml" Version="1.0.0" />
  </ItemGroup>

</Project>
```

Listing 5.5: Symfony.Framework-bundle.msbuildproj

The output of Peachpied Symfony project compilation is a local NuGet repository with more than 20 NuGet packages, located in the `Peachpied- Symfony-Nuget-Reposito` folder. This repository can be easily further expanded by adding new compiled libraries. In this respect, possible way of semi-automation of additional libraries compilation and inclusion has been discussed, and build files' structure and naming has been denoted in order to provide seamless future transition to semi-automatic solution.

5.3 Execution

Unfortunately, by successful compilation did our work not end. For that, it is required to preform another step, that is, to connect the uncompiled kernel part of `Symfony/skeleton` with the NuGet packages, link both to the Kestrel server and to execute the project.

Project file First, we need to create yet another `.msbuildproj` project, this time for the kernel part. Its content will resemble libraries' `.msbuildproj` files after imported `.props` files' content embedding, however, there are some significant differences. The first one is that we do not want to output this project as a NuGet package, thus, package relevant properties such as `PackageId` and `PackageOutputPath` will be omitted. Secondly, we know exactly what files we need to include for the compilation, include as the content and what to exclude. From the definition of kernel part we denoted before, it is the rest of the `Symfony/Skeleton` project apart from libraries. Therefore, every `.php` file located elsewhere than inside a `venor/[providerName]` folder will be included and vice versa. The content files files will be set-up accordingly. That leaves only the configuration of references. This was taken from the `composer.json` file just like when we were resolving dependencies for libraries. The only exception is that we intentionally excluded the flex components for the reason already explained before.

Binding with web server To be actually able to run our application, we need to bind it with 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 `dll` library compiled from PHP code [19]. This binding is done by creating two .NET projects, one for the website, and the second for the server. We could either create both of these projects separately and than bind them together, or, we can use some already made template that will handle this for us.

In addition to Kestrel server extension, PeachPie also provides several templates for project creating in Visual Studio. The PeachPie ASP.NET Core Web App template suited our intentions perfectly, therefore, we decided to use it. Alternatively, this can be done in .NET Core Command-Line Interface with `dotnet new -i Peachpie.Templates::*` command followed by `dotnet new web -lang PHP` command, where the first expression installs PeachPie .NET Templates and the second one actually creates the projects [19]. When this template is used, two projects are created. The Server project, containing the base configuration for Kestrel server and logic for `.php` request handling, and the website project, that is expected to contain PHP code for compilation. The project for the website has already been described in the previous chapter, and thus, we will directly proceed to the Server project.

Apart from configuration files `appsettings.json` and `launchSettings.json`, and project file `.csproj`, the Server project folder by default contains only `Program.cs` file. This file represents the entry point of the application. It consists of two classes, the `Program` class with the `static void Main(string[] args)` function and `Startup` class with 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 our `Main` method does in the `Host` object configuration declare the actual project name empty-page.

Complications At the moment, the application described will be executable, and will, upon localhost on port 5004 accessing, display content of the `index.php` file, located in the public directory of the Symfony project. However, in order to achieve this, many efforts were made in PeachPie misbehaviour errors identifications, but especially by the PeachPie development team during their actual resolving. Significant errors encountered, of which the solution had consequences for actual code shape, will be explained ahead.

The first error originated from the invalid handling of `new ReflectionProperty("Exceptio` `"trace")` expression. This was done due to lack of formal documentation of private `$trace` property of the `Exception` class. The error was later on resolved by update of PeachPie compiler and therefore is fully supported at the moment.

Much less fortunate was a problem with generated content. On start up, Symfony framework compiles container files, translations and similar heavyweight components and stores them in the `var\cache` folder as a long term cache. From the .NET point of view, such process is slightly problematic as it would mean two-phased compilation. The first phase to compile the code, run it and generate the long term cache files, and the second phase to recompile those additional files in order to be able to execute them. The actual implementation of the problem solving mechanism belongs into the administration of the PeachPie developers, hence, we will not include its description in the thesis. Nonetheless, there was a fatal error with functionality that performs the generated code load up. This error was caused by the difference in the interpretation of the following fragment of code in PHP and in .NET after the compilation.

```
return $this->services[$id]
    ?? $this->services[$id = $this->aliases[$id] ?? $id]
    ?? ('service_container' === $id
        ? $this
        : ($this->factories[$id]
            ?? [$this, 'make'])($id, $invalidBehavior));
```

Listing 5.6: Expression interpretation difference

When reaching the final part of the expression, the last line of PHP code is interpreted as the `$this->make($id, $invalidBehaviour)` method call. Compiled code under .NET, however, is not able to interpret it this way, resulting in runtime error.

Between other, less complicated errors, we can mention false `realpath` function return value implementation, where the function returned `null` instead of `false` on failure.

Result Empty-page project is now bound to the Server project and is fully operable. Moreover, previously compiled NuGet packages have been proven functional through successful execution with this minimal `Symfony/Skeleton` project. Ultimately, several PeachPie compiler shortcomings has been identified during the project execution attempts, hence, the possible compiler enhancements discovery can be considered a side effect of the execution phase as well.

5.4 Provided interoperability

On the one hand, Peachpied Symfony's NuGet repository now enables for some minimal Symfony projects to be compiled, which might by sufficient as a proof of concept, on the other hand, its usage is still unwieldy, and it does not manifest any of the possible interoperability options between Symfony and .NET. For the purpose of additional functionalities providing, another project has been crated. Again, following the naming conventions used during the development of Peachpied Wordpress, the project has been entitled Peachpied Symfony AspNetCore.

Peachpied.Symfony.AspNetCore This project is intended as a helper class for the Server project, encapsulating the Peachpied Symfony specific server configurations and additional interoperability integration and providing them via Application programming interface (API), following the standard ASP.NET Core `Use[midlewareName]` naming conventions. The public API is provided by the `RequestDelegateExtension` class.

As Peachpied Symfony performs project migration to the ASP.NET Core environment using .NET based web server, it can be expected that users will expect to be able to set up environmental variables in a .NET specific way with the use of mentioned `appsettings.json` configuration file. On the other hand, this functionality should not be compulsory as there might be users who would welcome the possibility to simply compile existing Symfony project without the need to worry about environmental variables as the application will use Symfony's environmental variables configuration that is already located 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 strings parsing, default configuration loading and `appsettings.json` configuration file processing. Finally, the `RequestDelegateExtension` class optionally overrides the Symfony's `bootstrap.php` class, that is responsible for environmental variables load up. Its content can be seen in the Listing 5.6. For the sake of readability, comments have been erased and long strings truncated with three dots in this Listing. The fragment of the code performing the overriding is captured in the Listing 5.7. This is done by accessing the `Context` object's script table that performs mapping from `.php` files to compiled .NET classes, and overwriting the script value stored for `bootstrap.php` file. In 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)) {
    throw new RuntimeException('Please run "composer require...');
} 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 5.7: 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 = new Context.MainDelegate(BootstrapMain);
Context.DeclareScript(bootstrapPath, md);
```

Listing 5.8: Context file mapping overriding

Compilation process semi-automation

Twig component

Twig-Razor interoperability

6. Real-life usage

- Oduvodneni naming konvenci skrz Peachpied Wordpress.

6.1 Example 1: Symfony project deployment

- Showing and explaining all the steps in complete symfony application to .NET porting

6.2 Example 2:Template engines interoperability

- Aplikace ktera v ramci sebe pouziva obe template engines - twig i razor a ukazuje schopnost renderovani sablon z twigu i razoru

6.3 Example 3: Full-fledged applications union

- Nastin merge 2 aplikaci - 1 napsane ciste v Symfony, druhe ciste v .NETu tak aby se nejak hezky doplnovaly (napriklad jedna zazemi a druha front)

Conclusion

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

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

[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] Micorosoft. *.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
CGI	Common Gateway Interface
VM	Virtual Machine
OOP	Object Oriented Programming
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
API	Application programming interface

A. Attachments

A.1 First Attachment