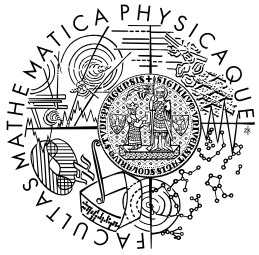

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**BACHELOR THESIS**

Tomáš Husák

# Client-side execution of PHP applications compiled to .NET

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Filip Zavoral, Ph.D.

Study programme: Computer Science (B1801)

Study branch: ISDI (1801R049)

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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

In ..... date .....  .....

Author's signature

Dedication.

Title: Client-side execution of PHP applications compiled to .NET

Author: Tomáš Husák

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract:

Write an abstract.

Keywords: PHP .NET Blazor Peachpie

# Contents

# Introduction

We can say a web application runs on two sides, that we call a server and client. The sides communicate with each other by Internet Protocols, where Hypertext Transfer Protocol (HTTP) is a fundamental communication standard. A user uses a web browser for requesting a server, which sends a response containing desired data back. The data can represent a web page or attachment like a file or raw data. A browser is responsible for interpreting and rendering a web page described by HyperText Markup Language (HTML). The Cascading Style Sheets (CSS) language accompanies HTML by enriching the web page with broad graphical content.

A server task is to process, collect and serve data requested by a client. The most popular language for server-side scripting is currently PHP.

Add a statistic for the popularity of PHP

The combination of CSS and HTML is sometimes sufficient for a web page. However, a modern web application needs to manipulate a web page structure depending on user behavior more sophisticatedly than the languages offer. This type of application utilizes a browser as an execution environment to change the web page structure, react to the events, save an application state, and control browser behavior. The scripting language Javascript became a browser standard for writing a client code inside most browsers like Google Chrome, Safari, Opera, and Mozzila.

Although Javascript is a powerful language, it is not appropriate for all scenarios and users. The reason can be dynamic typing or just a user practice with other languages. Despite the urge to write a client-side code in a different language, many technologies like Silverlight, which runs C# code in a browser, or Adobe Flash Player with Actionscript were deprecated due to insufficient support across the browsers. WebAssembly (WASM) was developed to offer a portable binary-code format for executing programs inside a browser [1] in 2015. WASM targets to enable secure and high-performance web applications. The advantage of WebAssembly is a being compilation target for many programming languages. Browsers support interoperability between WASM and Javascript to utilize both language advantages. Since December 2019, when World Wide Web Consortium (W3C) has begun recommending WebAssembly, it is easy to migrate other languages to the browsers supporting this recommendation.

Many projects use the WASM as a target of compilation. For example, the project PHP in browser [2] enables running a PHP script inside our browser using predefined Javascript API or standard HTML tag. Another project is an open-source framework Blazor [3] developed by Microsoft. It provides runtime, libraries, and interoperability with Javascript for creating dynamic web pages using C#.

Add a statistic for popularity of .NET

The .NET and PHP popularity led to the creation of the Peachpie compiler. Peachpie [4] compiles PHP to .NET and enables interoperability between the languages. Peachpie is usually used to connect a frontend written in PHP with a backend written in C# to utilize both aspects of languages on a server side.

Peachpie allows applying PHP to Blazor. Although Blazor can straightfor-

wardly reference compiled PHP by Peachpie, the collaboration between the code and Blazor seems complicated. Methods of how to utilize PHP scripts as a part of a Blazor website are not clear. This thesis targets to identify use-cases that will make use of the integration between Peachpie and Blazor and suggests a solution, which creates a library to execute and render compiled PHP scripts in a browser. Blazor is used as an execution environment for these scripts. The solution tries to achieve two goals. The first goal is to implement the support for using compiled PHP scripts with Blazor because no existing library supports the integration. The second goal is to enable web development on a client-side with PHP.

The integration between Peachie and Blazor can yield to following benefits. A community of PHP developers is significant. Thus, many PHP libraries apply to work with client's data, pdf, graphics and offer handy tools. The possibility to migrate the language PHP together with its conventions to a browser will impact developing dynamic web applications due to the PHP community and the libraries. It can join PHP and C# developers to collaborate with their programming languages using a minimum knowledge of the integration. Another interesting functionality of this idea is a full C#, PHP, and JavaScript interop which offers more options for developers and future extensions.

The first chapter addresses the analysis of related work, alongside descriptions of the technologies used in the integration. The second chapter analyses running PHP on the client-side and other problems related to used technologies. The third gives a detailed problem's solution. There are examples that demonstrate how to use all aspects of the created solution in chapter 4. In chapter 5, we can see benchmarks that explore the limits of the implementation and compare them with the already existing project. And the last chapter relates to a conclusion of this solution.

# 1. Existing technologies

In this chapter, we will give a short overview of web application functionality. We will explore server-side scripting using PHP and client-side scripting using Javascript in order to obtain observations of user conventions for interaction with web applications. Afterward, we will introduce WASM, followed by the existing project enabling to execute PHP in a browser. We will give short information about the .NET platform and C# language. In the last sections, we will introduce Blazor and Peachpie.

## 1.1 PHP

The basic principle of obtaining a web page is a request-response protocol, where a client sends a request for the web page using an HTTP protocol and receives a response with requested data. The protocol uses a dedicated message format for communication. Statelessness is a typical characteristic, meaning that a server has to retain information about clients and add additional information to the messages in order to distinguish between the clients.

Since the server contains business logic, a browser has to send necessary data for required actions by an HTTP message. The data are usually encoded as a part of Uniform Resource Locator (URL) or in the HTTP message body. HTML presents a tag `<form>` that enables interaction with the web application by a web form. Figure 1.1 contains an example of the tag. `<form>` can contain other tags, which are displayed as various types of fields. A user fills these fields, and the browser sends the data as a new request to the server. We can specify how the data will be encoded. `get` method is one of the basic ways. It encodes the data as a pair of keys and its values to the query part of the URL. There is an example of URL `http://www.example.com/index.php?par1=hello&par2=world`. A query part begins with a question mark. We can see parameter keys `par1` and `par2` containing values `hello` and `world`. Another method is called `post`, which encodes it in the request body, which does not appear in the URL.

Although PHP [5] was originally designed for user page templating on a server side, it has been adjusted gradually to enable writing application logic. PHP is an interpreted language maintained by The PHP Group.

We will describe the language by using figure 1.1 as an example. The script includes a header, which adds a proper beginning of an HTML document. Then, it prints a `post` content together with a file content whenever the file `file` was obtained. There is a form enabling to send information about a name and attach a file to the massage. A browser sends the message to the server via `post` method when the form is submitted. The request is handled by `index.php` defined in `action`. In the end, the script includes a proper ending of the document by `footer.php`.

As we can see, the PHP code interleaves the HTML code, which has appeared to be a helpful method for data binding. We call the feature HTML interleaving, which allows inserting PHP code in `<?php ... ?>` tag. These fragments do not have to form individual independent blocks of code closed in curly brackets, as the example demonstrates by using the `foreach` cycle. An interpreter executes

```
<?php
    include("header.php");
?>

<h1>Superglobal POST</h1>
<?php
    foreach($_POST as $key => $value) { ?>
    <p><?php echo $key; ?> => <?php echo $value; ?></p>
<?php } ?>

<h1>File content:</h1>
<p>
<?php
    if($_FILES["file"])
    {
        echo file_get_contents($_FILES["file"]["tmp_name"]);
    }
?>
</p>

<form action="/index.php" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name"><br>
    <label for="file">File:</label>
    <input type="file" id="file" name="file"><br>
    <input type="submit" value="Submit">
</form>

<?php
    inlude("footer.php");
?>
```

Figure 1.1: An example of PHP code in file index.php.

a script from top to bottom. Everything outside the PHP tag is copied into the body of the request.

We do not see any specification of type next to variables. This is because the type system is dynamic. A variable represents just a reference to the heap. Its type is determined during runtime.

PHP has superglobals [6], which are built-in variables accessible from all scopes of the script. Following superglobals are relevant to the thesis. The `$_GET` variable stores parsed query part of the URL. The `$_POST` variable stores variables which are sent by post method. The `$_FILES` variable contains information about uploaded files sent by a client. The uploaded file is saved as a temporary file, and standard reading operations can obtain the content. This is demonstrated in the previously mentioned example by `file_get_contents` function.

Maybe add information about SESSION.

The nature of the request-response semantic usually results in a one-way pass of the application. After dealing with a request, the script is terminated, meaning that the request is sent, and variables are disposed. One of the well-known design patterns relating to PHP is the Front controller. Usually, the main script invokes other parts of the program, based on the request, to deal with it and send the response back. The idea of this pattern can be shown in figure 1.1. In the beginning, we delegate header rendering to `header.php` script. Then we render the body and include `footer.php`, which cares about the proper ending of the HTML page.

We can divide code in several ways. Global functions are the most notable characteristic of PHP despite wide-spread object-oriented programming. They are defined in the global scope and accessible from anywhere. The next option is an object inspired by object-oriented programming. We can include a PHP code from other scripts. They can be recursively included during runtime, where variables remain across the inclusion. Scripts can be composed into a package, which another code can reuse.

## 1.2 Javascript

A client-side code needs to control the rendered page and access a web interface providing additional services, which is usually accessible via Javascript, in order to interact with a user. We will start with a description of loading Javascript in a browser. We will introduce a page representation in a browser alongside page events. In the end, we will present Javascript as a scripting language for the creation of a responsive web page.

We can image a web page structure as an tree. Its nodes are tags or text fragments, and its edges connect nodes with their children. One representation of this tree is Document Object Model (DOM). Each node is represented by an object with special parameters relating to HTML and CSS. The nodes can contain other nodes representing their children. Afterwards, there is a document node representing the whole document together with its root node.

The process of generating a web page follows several steps. A browser parses the HTML page line by line. If a script occurs, the browser starts to execute the code, which can access already parsed tags. The order of processing is important

```
<!DOCTYPE html>
<html>
    <head>
    </head>
    <body>
        <button id="alert">Click to alert</button>
        <script>
            var handler = function (arg) {
                var timer = new Promise((resolve) => {
                    setTimeout(resolve, 1000);
                });

                timer.then(() => window.alert("Hello world."));
            };

            var button = window.document.getElementById("alert");
            button.addEventListener("click", handler);
        </script>
    </body>
</html>
```

Figure 1.2: A Javascript code.

for manipulation with an HTML structure. This limitation can be solved by web events mentioned later, but it is a convention to add scripts to the end of the body part after all HTML tags are parsed.

Events are the most common method of how to react to changing a web page state. Every event can have some handlers(listeners). Whenever an event occurs, it calls all its listeners. There are many event types, but we will mention the ones that are important for us. HTML tags are the most common entities which can have some events. For example, a button has an `onclick` event which triggers when a client clicks on the button as we can see in figure 1.2. Other events can represent a state of a page like `onload` which fires when the whole HTML document is parsed.

A browser provides more APIs valuable for the application, like fetching extra data from a server or local storage. These APIs are mentioned as Web API [7].

ECMAScript (ES) is a Javascript standard recommending across browsers. We can see later an abbreviation ES2015 which determines the ECMAScript version. Javascript is a high-level language usually executed by a browser's dedicated Javascript engine. Browsers run scripts in a sandbox to prevent potential threats of harmful code. However, it can also be run on a desktop by Node.js, a Javascript runtime running outside a browser. Figure 1.2 will be used to show the language in the simple scenario. The page contains a button that invokes an alert with a second delay when a client clicks on it.

At first glance, we can see the type system is dynamic, which is similar to PHP. `window` is an essential global variable, which is an object representing the browser window of the running script. The window object consists of all defined global variables. It also contains a document property, which is an API for manipulating

the DOM tree. We can see the usage of the document property in the example. Javascript object is often used as a wrapper of Web APIs.

Functions are first-class citizens in Javascript. We can treat them as common variables. Javascript supports an event-driven style that helps to react to events conveniently. There is a handler assigned to the click event in figure 1.2.

Maybe add a section about async functions. Compare it with PHP

Javascript is single-threaded, but allows effective synchronous execution. This can be achieved by `Promise`, which is a structure representing an unfinished process. We can separate a large task into smaller ones in other to offer the processing time for other parts of the application. These processes can be chained. Although the structure can give an illusion of multi-threading, it uses the scheduler for planning the next task executed by the main thread after the previous task is completed or an event triggered it. The single thread is critical for blocking operations like time demanding computatitons which causes thread freezing.

`Worker` object represents a web worker [11], provided by a browser, enabling to run the script in the background. The worker limitation is communication with UI thread only by handling message events. Messages have to be serialized and deserialized.

Javascript can organize a code by function and objects similar to PHP. A module can gather a larger collection of code. Global entities of the code can be exported to another script. These exports make an API of the module. The module advantage is to define the API and to hide the internal code, which is not relevant for the user.

## 1.3 WebAssembly

WASM [9] is a new code format that can be run in current browsers. It has a compact byte format, and its performance is near to a native code. WASM is designed to be a compiling target of popular low-level languages like C or C++ due to its memory model. It results in the possibility to run other languages in a browser because its runtime is often written in C or C++. Browsers enable to run Javascript alongside WebAssembly, and even more, their code can call each other. WASM code is secure as same as a code written in Javascript because of the sandbox.

Thread [10] support is currently discussed nowadays, and it will probably be added in future browser versions. After all, new versions of Google Chrome experiments with proper multi-threading support. A replacement of multi-threading can be Web workers mentioned in Javascript section.

Despite supporting to run WASM in a browser, the browser cannot load it as a standard ES2015 module yet. WebAssembly JavaScript API was created in order to be able to load a WebAssembly to a browser using JavaScript.

## 1.4 PHP in Browser

The project [2] aims to use compiled PHP interpreter into WebAssembly, which allows evaluating a PHP code. The page has to import a specialized module

```
<script type = "text/php">
    <?php vrzno_run('alert', ['Hello, world!']);
</script>
```

Figure 1.3: An example of PHP script block.

`php-wasm`. A PHP code is evaluated by using JavaScript API or writing a specialized script block as we can see in figure 1.3. PHP can afterward interact with JavaScript using a specialized API. In the figure, the code calls Javascript `alert` function with the parameter.

At first glance, that might be a good enough solution, but several parts can be problematic due to PHP semantics. The solution does not offer additional support for using PHP scripts on the client side. For example, superglobals are unused due to a missing server. This issue is reasonable because this is the server task, but we cannot get information about a query part or handling forms without writing a JavaScript code. The next problem relates to how a script can navigate to another script without an additional support code, JavaScript.

## 1.5 C# and .NET 5

We will introduce the Common Language Infrastructure (abbreviated CLI) [18] before diving into .NET, which is an overused name for several technologies. CLI is a specification describing executable code and runtime for running it on different architectures. CLI contains descriptions of a type system, rules, and the virtual machine (runtime), which executes specified Common Intermediate Language (abbreviated CIL) by translating it to a machine code. The virtual machine is named CLR (Common Language Runtime). CIL's advantage is a compilation target of languages like C#, F#, and VisualBlasic, which gives us great interoperability. .NET Framework, .NET 5, and Mono are implementations of CLI. These implementations are usually uniformly called .NET.

.NET 5 [17] is the lastest version of .NET Core, which is a cross-platform successor to .NET Framework. From now on, we will refer .NET 5 as .NET, since it should be the only supported framework in the future. .NET is an open-source project primarily developed by Microsoft. It consists of runtime for executing CIL and many libraries that can represent whole frameworks like ASP.NET, which aims to web development. A large collection of code is usually compiled into an assembly containing the code and additional metadata. As assembly can represent either a library or an executable program.

Mono aims at cross platform exection of CIL. Recently, they started to support compilation [12] into WebAssembly. This support allows executing CIL inside browsers. The compilation has two modes. The first one is compilation Mono runtime with all using assemblies. The second one only compile Mono runtime, which then can execute .dll files without further compilation of them into WebAssembly. A consequence of these compilations into WebAssembly is enabling to call Javascript and WebAPI from C#.

.NET Standard represents API specifications of .NET libraries across different implementations. .NET Standard offers to specify minimum requirements for the

code.

C# is a high-level language using strong typing and a garbage collector. It has a multi-paradigm, but its common characteristic is the objected-oriented style. These features cause that C# is a good language for a huge project which needs discipline from developers to hold the code understandable and manageable.

## 1.6 Blazor

Blazor is a part of the open-source ASP.NET Core framework. Blazor allows creating client-side web applications written in C# language. Blazor framework offers two hosting models [13] which have different approaches to creating web applications. The first one is referred to as Blazor Server App and represents a server-side web application using specific communication between a client for better functionality. The thesis uses the second model, which Microsoft refers to as Blazor WebAssembly App, enabling to move business logic to a client-side without using Javascript.

From now on, we will use Blazor App to refer Blazor WebAssembly App. The application can be hosted by a standalone project representing a standard ASP.NET Core web server. The hosting consists of serving application .dlls and static files like HTML, CSS. The division enables a choice of a place for the implementation of business logic. Thus, we can move the majority of business logic to the client and use the server for connection to a database or, we can use the client only for rendering the page. When we chose the template, there are two main projects to describe.

As we can see in figure 1.4, there is a server, which serves the Blazor App to a client. The project contains a standard builder of a host using a StartUp class,

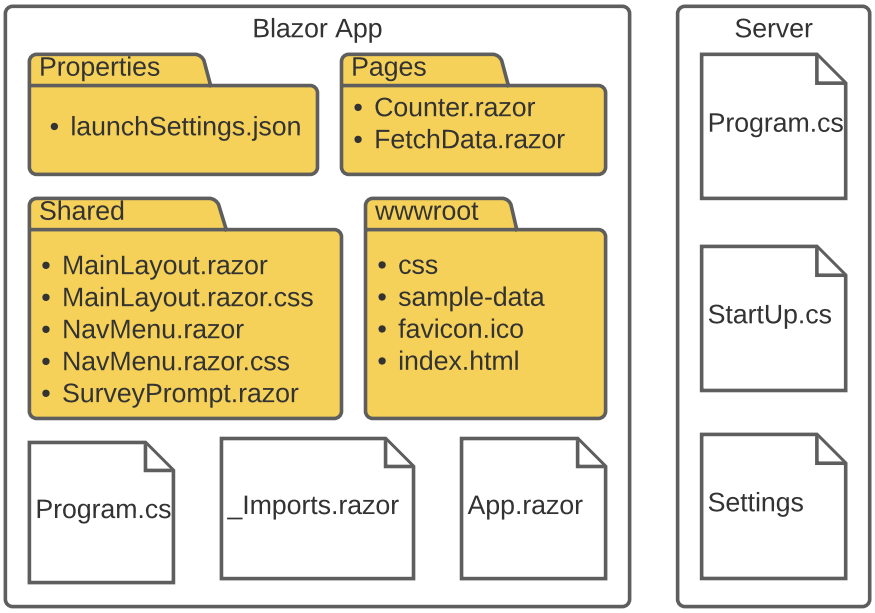


Figure 1.4: Server and WebAssembly App projects.

```
@page "/example"
@inject HttpClient Http

<h1>Example</h1>
@if (!loaded)
{
    <p>Loading...</p>
}
else
{
    <p>Ticks: @ticks</p>
}

@code {
    private bool loaded = false;
    private int ticks = 0;

    protected override async Task OnInitializedAsync() {
        ticks = await Http.GetFromJsonAsync<int>("ticks.json");
        loaded = true;
    }
}
```

Figure 1.5: Example of Razor page.

which configures an HTTP request pipeline processing requests in `StartUp.cs` file. A middleware is a segment of an HTTP request pipeline, which cares about some functionality related to request processing. The pipeline contains a middleware providing the Blazor files.

We will describe the second project (Blazor App) in figure 1.4 to explain basic entities and their interaction with each other. We will start with a new format Razor to get familiar with it. Razor is a markup language interleaving HTML with C#. Razor uses special sign at with keywords to identify C# code in HTML. Razor compilation results in a pure C# code representing the web page fragment. We can see an example of Razor in figure 1.5.

Although the format is self-explaining, we describe the keywords. The first line begins with a `page` keyword determining a part of the page URL. The next keyword is `inject`, representing a `HttpClient` service injection. An `if` keyword is an control structure iterleaved by an HTML code. A `code` keyword contains a regular C# code, which can be used in the whole `razor.` file.

A Razor file is compiled into a C# dedicated class. The class inherits from `ComponentBase` or implements `IComponent`, which provides necessary methods for rendering the page. Components can be arbitrarily put together in order to form the desired page. We can see the generated component from figure 1.5 in figure 1.6.

We can assign the Razor keywords to parts of the code in the figure. `page` keyword stands for `Route` attribute. `inject` keyword stands for parameter attribute. The parameter is assigned by a dispatcher, during the component initialization.

```
[Route("/example")]
public class Index : ComponentBase {
    private bool loaded = false;
    private int ticks = 0;

    [Inject] private HttpClient Http { get; set; }

    protected override void BuildRenderTree(
                                RenderTreeBuilder __builder) {
        __builder.AddMarkupContent(0, "<h1>Example</h1>");
        if (!loaded)
        {
            __builder.AddMarkupContent(1, "<p>Loading...</p>");
            return;
        }
        __builder.OpenElement(2, "p");
        __builder.AddContent(3, "Ticks: ");
        __builder.AddContent(4, ticks);
        __builder.CloseElement();
    }

    protected override async Task OnInitializedAsync() {
        ticks = await Http.GetFromJsonAsync<int>("ticks.json");
        loaded = true;
    }
}
```

Figure 1.6: Razor page generated to the C# class.

`code` keyword is a part of class content. Another markup is transformed into calling a specialized method in the `BuildRenderTree` function, which describes the page content for rendering.

A component has several stages, which can be used for initialization or action. Virtual methods of `ComponentBase` represent these stages. We can see the textttOnInitializedAsync method, which is invoked after setting the component parameters.

We should mention asynchronous processing because it helps render a page with long-loading content. Blazor allows using Tasks and async methods, separating the code into smaller tasks planned by a scheduler. Blocking operations in Blazor are projected into UI because it is single-threaded due to Javascript and WASM.

We return to the project description. Folders `Pages` and `Shared` contains parts of Blazor pages written in Razor. _Imposts.razor contains namespaces, which are automatically included in others `.razor` files. The next folder is `wwwroot`, containing static data of the application. We can see `index.html`, which cares about loading parts of the Blazor application to the browser. We will call all static files in `wwwroot`, additional Javascript scripts and WASM runtime Static Web Assets.

```
<body>
    <div id="app">Loading...</div>
    <div id="blazor-error-ui">
        An unhandled error has occurred.
        ...
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>
```

Figure 1.7: index.html

We will describe the loading of Blazor into the browser to fully understand the interaction between Blazor and the browser. We have the server, the Blazor App, and other optional user's defined projects. When we start the server and tries to navigate the web application, the following process is done. The server maps the navigation to `index.html` and sends it back.

The `index.html` contains a script initializing Blazor as we can see in figure 1.7. The first step is to load all resources, which are defined in a separate file. Blazor cuts all unnecessary `.dll` files to reduce the size. For this reason, all `.dll` files have to be used in the Blazor App code in order to be contained in the file. These resources comprise Mono runtime compiled into WASM, additional supporting scripts, and all `.dll` files containing the whole application (Blazor App with referenced libraries). The supporting scripts initiate the runtime and execute it. The runtime includes the `.dll` into the application and calls the `Main` method in `Program.cs` defined in Blazor App project. We can see the process in figure 1.8.

`Main` method uses `WebAssemblyHostBuilder` to set the application. It defines services, which will be able through the dispatcher. It sets a root component, which will be rendered as the first. The host is run. Afterward, the application provides the dispatcher, cares about rendering, and communicates with the runtime to offer interop with Javascript.

`App.razor` is the last file for clarification. It is the root component in default. It contains a specialized component, `Router`, enabling to navigate the pages.

In the end, we will describe page navigation and rendering and handling events. The navigation [14] can be triggered by an anchor, form, or filling up the URL bar. The URL bar is handled separatly by a browser. JavaScript can influence the remainings elements by adding a listener to their events. Blazor App handles only an anchor by default. After clicking on an anchor, a navigation event is fired. One of the handlers is a javascript function, which invokes C# method through Mono WASM and prevents a browser navigation, when Blazor App handles it. The method represents a navigation handler in Blazor App. A user can add listeners to the handler, but the Router implements default behavior for navigating. `Router` finds out all components which implement an `IComponent` interface and tries to render the page according to path matching `RouteAttribute` of a component whenever the navigation is triggered. It creates an instance of the class and fills in its parameters. `Router` calls `BuildRenderTree`, which enables running the rendering process. Previous created intences of components are disposed. The navigation can be redirected to the server if there is no match.

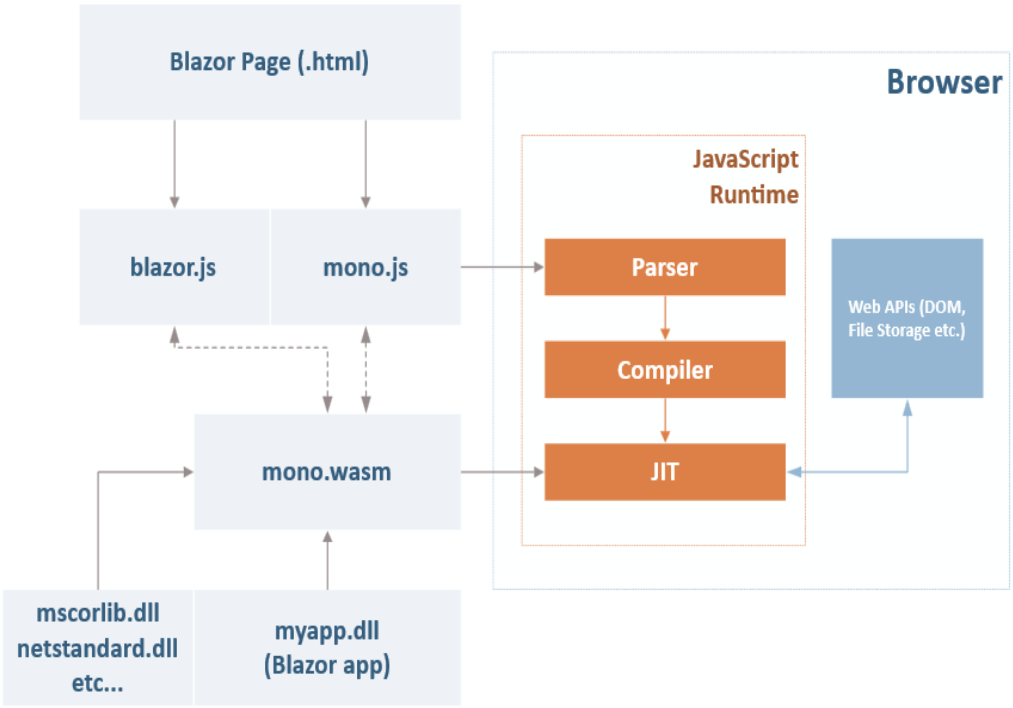


Figure 1.8: Running a Blazor WebAssembly App on client-side.

The rendering process begins with the Renderer initialized in the application builder. Renderer keeps a copy of DOM in memory, creates page updates, and calls Javascript API for changing the web page DOM in a browser, using the runtime interop support. Blazor provides API for invoking Javascript functions and vice-versa. Renderer provides `RenderTreeBuilder` for describing page contents. The builder provides an API for adding various types of content to `Batch`, which is a specialized structure for describing previous and present DOM. Although DOM changes is performance-expensive, a diff algorithm recognizes and tries to reduce the updates in `Batch`. The usage of `RenderTreeBuilder` is complecated, because it has to be adapted to the algorithm. The purpose of Razor is to make the usage more effortless when the compilation implements the `RenderTreeBuilder` for us. When the Renderer prepares `Batch`, it calls specialized Javascript API for changing the page through Mono runtime.

The diff algorithm is used to minimize the browser DOM update after all components used `RenderTreeBuilder` to render their content. This algorithm used sequence numbers for parts of HTML to identify modified sections. Sequence numbers are generated in `RenderTreeBuilder` instructions during a compilation. A benefit of this information is detecting loops and conditional statements to generating smaller updates of DOM.

Event handling is just clever usage of the Renderer with dedicated Javascript API for updating, where the API registers the listener. When the event is fired, the listener invokes C# method representing the handler through the WASM runtime.

## 1.7 Peachpie

Peachpie [16] is a modern compiler based on Roslyn and Phalanger project. It allows compiling PHP scripts into a .NET assembly, which can be executed alongside standard .NET libraries. All information about the scripts are saved in the assembly. We will describe the basics.

Because the languages have a different type system, Peachpie brings dedicated types for representing PHP variables in .NET. Some of these types are `PhpValue` representing a standard PHP variable, `PhpArray`, or `PhpAlias` which is a reference to `PhpValue`.

Another abstraction is the `Context` class. We can imagine `Context` as a state of the script while it runs. `Context` consists of superglobals, global variables, declared functions, declared and included scripts. It also manages an input and output, where we can choose a resource. `Context` can also be considered as a configuration of the incoming script's execution. All information about a request can be arranged to mock every situation on the server-side. The possibility of saving `Context` and using it later is a significant advantage. We can use the class API to obtain information about compiled scripts.

Because some PHP extensions are written in C or C++, Peachpie implements them using .NET libraries, which can add additional functions providing an extra nonstandard functionality such as an interaction with a browser.

The main advantage of the compiler is the great interoperability between PHP and .NET. An option to work with C# objects, attributes, and calling methods will become crucial for achieving advanced interaction between Blazor and PHP.

The compiler successfully compiled well-known web frameworks like WordPress or Laravel. Thus many companies use it for combining the existing frameworks with a C# backend.

However, there are limitations following from differences in the languages and the stage of development. Availability of PHP extensions depends on binding these functions to C# code which gives equivalent results. The .NET libraries can be executed in an independent environment. However, the code can have performance issues in WebAssembly. The previously mentioned interoperability has limits as well. Csharp constructs like structs and asynchronous methods are undefined in PHP.

V dobe psani prace tady byly nejake komplikace

# 2. Problem analysis

describes misto relate

aim at

We will divide the analysis into two sections. The first section relates to defining requirements, which the proposed solution will solve. Four scenarios will describe these requirements, and they will point to the resulting benefits achieved by them. These scenarios will aim to use the integration in different ways. The second section will observe available architectures in view of used technologies, and it will describe the solution's architecture.

## 2.1 Scenarios

Use cases pouzit

describes misto relate

The first scenario moves a website written in PHP to a client side.

majority

It can help saving the server's resources by loading the website's major to a client using one request.

Front controller neni

We can imagine a PHP website using a Front controller pattern. We want to handle all navigation by the Main script, distributing an additional workload to other scripts.

Pozadavky co to ma umet a pro jake prilezitosti

zminit pro jake uzivate cilim, knim priradit usecasy

The URL information should be accessible in the super global $_GET alongside the query. Scripts should render a page by the interleaving or echo. Rendering should be triggered once per navigation, which means clicking at an anchor tag. We should choose a Context duration If we want to use the same context for a whole component life or change it after navigation. This extension brings a new look at PHP programing when we can utilize saving the context among navigations. A problem comes with external resources like images. These resources are needed, while a particular part of the page references them. This complication has to make another request to the server for obtaining the resource.

The second scenario aims to inject a PHP code to the Blazor page. The page will do some data processing. There exist a dedicated PHP library solving this processing, and we are familiar with it. We should write a PHP script using the library and inject it as a part of the page. The PHP script should interact with a client by a Form tag to avoid Javascript and advanced interaction with Blazor. Get and post methods should be enabled and should use the correct superglobals. There should be file support that will enable loading and saving files from the script. A PHP script should be rendered as in the previous example.

The third scenario aims to fully utilize aspects of Blazor and move it into the PHP world. We should be able to use all Blazor interfaces from PHP The scenario is intended for users, which have a notion about Blazor functionality and want to make the rendering time faster for demanding web applications. The solution

should offer constructs to improve interaction with Blazor in PHP. In the end, we should be able to place the web application into the desired place in the Blazor application.

The fourth scenario combines previous scenarios. We should be able to add a PHP website as a part of the Blazor website. The PHP website should be able to navigate the component created in the third scenario.

## 2.2 Architecture analysis

Blazor App is used as a cornerstone for our application. Blazor provides the C# migration to a browser and afterward interop with Javascript. Peachpie transforms PHP scripts into the assembly. The assembly can be used in Blazor App without any limitation. Peachpie's libraries can be referenced from Blazor due to the compatibility of their target frameworks.

Where to compile the scripts is the first question. They can be regarded as static resources of the Blazor App and loaded after the Blazor's initialization. Afterward, the Peachpie can compile them and execute them. This approach allows us to add a PHP script during the runtime. The second option is to compile the scripts ahead of time and reference the assembly from Blazor App. It saves the compilation time during the runtime. A disadvantage could be the larger size of the initial response. The solution selects the second option. It benefits from the compilation check on the server-side. We suppose that the benefit is more helpful than the smaller size of the initial response. The assembly contains the same information about the scripts, and Peachpie is intended for a static compilation.

We have to figure out how to attach a PHP code, which is compiled into the assembly, to the Blazor App. Although, we can now call functions written in PHP from Blazor. We want to create an abstraction over the Blazor environment in order to simplify the interface. The abstraction should offer a representation of PHP scripts in Blazor. It should allow an option for accessing the Blazor interface for advanced features. It should be compatible with the Blazor environment in other to allowing a smooth collaboration between the abstraction and the Blazor pages. A Blazor page consists of components. We can achieve collaboration by utilizing the component to represent PHP scripts. We can benefit from a component's architecture. Componets can be arbitrarily put together, which offers to place our PHP section in the desired place in the Razor code. Even more, we can replace the Router with the component representing PHP scripts. Afterward, scripts will care about the whole Blazor website's content. The component provides a sufficient Blazor interface for rendering control and interaction with a browser. We can illustrate the options of usage in figure 2.1.

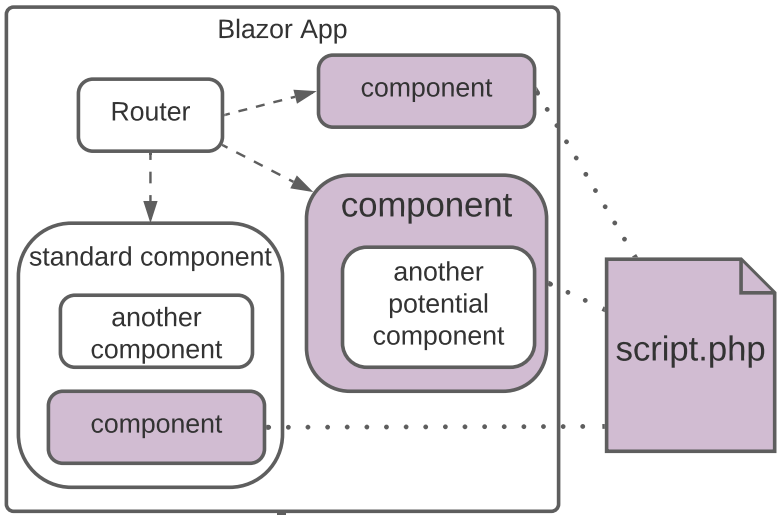


Figure 2.1: The component representing a PHP script.

We can think about how to represent PHP scripts as components. There can be one type of component, which will provide the abstraction for all the PHP code in scenarios. A problem with this approach is that the scenarios claim different levels of abstraction. The third scenario wants to use the component for offering the Blazor interface accessible from PHP code. The offer should contain identical or similar options, which are given in a C# code. The second scenario wants to use the component as an adjustable provider. The provider finds and executes PHP scripts. Its purpose is to keep the user away from knowing about the detailed structure of Blazor and the integration. Another important thing is a provider's role in a Blazor App. The provider can behave either as the Router or as a navigatable component, which enables the navigation of PHP scripts. The conflict yields to create more types of components. These types will provide the abstraction for the particular scenarios. The solution will reach two types of components. The first one wants to bring Blazor to PHP in order to utilize the whole environment. The second one aims to present transparent executing of standard PHP script without strangeness of connection between Blazor and PHP.

We will focus on the first component. We will call the component PhpComponent due to the effort of moving the component concept to PHP. PhpComponent aims to the third scenario. Despite language's differences, we can utilize the common concept of classes and inheritance. Peachpie allows inheriting C# class in a PHP code. This feature results in full support of component interface without creating new structures for managing component's behavior from PHP. We can inherit ComponentBase class in PHP and use its methods in the same way as C# class. The inheritance offers the required interface for creating effective rendering in scenario 3. There are also subproblems with the differences. The current Peachpie version does not support some C# specifics fully. The reason can be a hard or impossible representation of C# entities in PHP. It should be developed some PHP support for making the usage of the interface easier. The support will replace the missing usage of the interface.

We will call the second type of component PhpScriptProvider expressing an environment for executing standard PHP scripts. PhpScriptProvider aggregates the requirements of the rest scenarios by a single component. Although, the provider has more than one purpose. The main idea of serving a PHP code is the same. The provider should be able to navigate and execute PHP scripts. Because the rest scenarios try to hide the integration, the provider should support the following features. It should pretend a server's behavior. The behavior contains rendering everything, which is outside the PHP section or written by echo. Superglobals are often used for obtaining additional information given by the user. An ability to fill $_GET variable with the URL's query part should be presented. It should change a standard Form functionality to saving the Form's information into superglobals and execute the script again. Loading and saving files submitted by Form is essential for avoiding using Javascript. There is an interesting thing about saving the script's context to the next execution. These abilities are the same for the rest scenarios. We will describe the provider's modes. These modes are intended to solve the rest scenarios.

The first mode relates to the first scenario. It enables to set the provider as a root component. It handles all navigation events, determines the script's name, finds it, and executes the script. PhpComponents can also be a navigation's target.

The second mode relates to the second scenario. It enables the provider's insertion into a Razor page. Afterward, the provider executes the specified script.

The third mode relates to the last scenario. It enables to navigate the set of scripts with respect to URL. The navigation is generally maintained by the default Router. The component only provides navigation to scripts.

We can see that two different components are rational ways how to separate the problems and offer an understandable difference between the components.

# 3. Solution

describes misto relate

aim at

The section describes the complete solution to solve the scenarios. We will start with an overview of the solution's parts. Then, we will give a detailed description of each part.

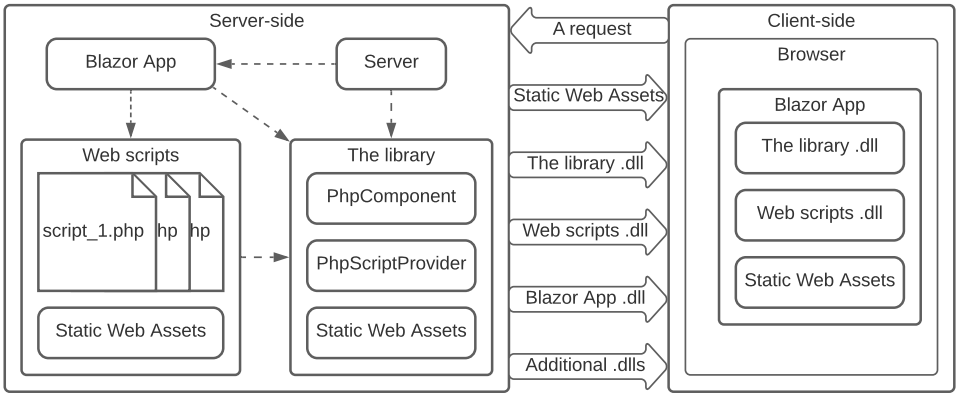


Figure 3.1: The solution's infrastructure.

Figure 3.1 illustrates the composition of projects forming a Blazor website, which will use PHP scripts. There are four projects on the server-side. We can see the user's defined PHP scripts in the Web scripts project. Peachpie will compile the project to the .NET assembly. The next project is a library containing an API for including PHP scripts to the website. There are PhpComponent and PhpScriptProvider, mentioned earlier, together with additional code support necessary for the correct functionality. There is the Blazor App project, which becomes the environment for running PHP scripts in a browser. The server serves these projects. The server has to provide additional static web assets contained in Web scripts and The library projects. Dot lines represent references between projects. We can see the library provides functionality to the server, Blazor App, and the scripts. Blazor App injects the scripts using the components. We can see an instance of the Blazor App on the client-side. Arrows indicate an interaction between the sides.

The first part will aim at PhpComponent. It will introduce the implementation problems connected to creating render demanding applications, and it will present the proposed solution. The second part will talk about PhpScriptProvider. It will suggest a convenient way how to include the scripts into a browser, and it will present the component's design. The last part of the description will relate to the server's settings.

## 3.1 PhpComponent

Problem struct creation in BuildRenderTree

PhpComponent structure

Additional API for tag, attributes, + rendering + timer

Javascript interop

## 3.2 PhpScriptProvider

Division into functionalities (Router, ScriptProvider, Script)

Common things(Script finding,rendering, context, Context save, Using forms, files, parsing url)

Router navigating

ScriptProvider navigating

Script navigating

## 3.3 Server

serving web static assets

# 4. Examples

Scenario 1,2,3,4

# 5. Benchmarks

Rendering speed(Asteroids first version vs. the current)

Problem with gd library

# Conclusion

# Bibliography


Webassembly. URL https://en.wikipedia.org/wiki/WebAssembly.

Threads. URL https://developers.google.com/web/updates/2018/10/wasm-threads.

Web workers. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

Compilation. URL https://www.mono-project.com/news/2017/08/09/hello-webassembly/.

Hosting models. URL https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0.

Navigation. URL https://chrissainty.com/an-in-depth-look-at-routing-in-blazor/.

Running blazor. URL https://daveaglick.com/posts/blazor-razor-webassembly-and-mono.

Peachpie. URL https://docs.peachpie.io.

.net core. URL https://en.wikipedia.org/wiki/.NET_Core.

Cli. URL https://en.wikipedia.org/wiki/Common_Language_Infrastructure.

The project php in browser. URL https://github.com/oraoto/pib?fbclid=IwAR3KZKXWCC3tlgQf886PF3GT_Hc8pmfCMI1-43gdQEdE5wYgpv070bRwXqI.

Blazor's homepage. URL https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor.

Peachpie's homapage. URL https://docs.peachpie.io.

Php. URL https://en.wikipedia.org/wiki/PHP.

Php manual. URL https://www.php.net/manual/en/.

Webapi. URL https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction.

Webassebmly. URL https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

# List of Figures

# List of Tables

# List of Abbreviations

**HTTP** Hypertext Transfer Protocol

**HTML** HyperText Markup Language

**CSS** Cascading Style Sheets

**WASM** WebAssembly

**W3C** World Wide Web Consortium

**URL** Uniform Resource Locator

**DOM** Document Object Model

**ES** ECMAScript

# A. Attachments

## A.1 First Attachment