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

Study branch: Software and Data Engineering

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

I would like to give thanks to my supervisor, RNDr. Filip Zavoral, Ph.D., and consultant, Mgr. Ing. Robert Husák, for their time and advice. I would like also to thank my family for their support.

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: Peachpie is a modern compiler enabling the compilation of PHP scripts into .NET. Blazor is a new part of the ASP.NET platform offering the usage of C# on a client side due to a new web standard, WebAssembly. This thesis explores a new approach of execution based on the integration between Peachpie and Blazor. PHP scripts can be referenced from Blazor and evaluated, but there are many steps to make it work. We analyzed these steps and designed a solution for inserting these scripts to Razor pages, navigating, and evaluating them. It results in a library containing helper classes to enable PHP programmers to move the PHP execution to a client side with the advantages of the Blazor environment. However, the difference of used technologies limits usage possibilities, which are shown by two benchmarks.

Keywords: PHP .NET Blazor Peachpie

Contents

1. Introduction

Web applications usually run on two sides that we call a server and a client. The sides communicate with each other by Internet Protocols, where Hypertext Transfer Protocol (HTTP) is the fundamental communication standard. Users use web browsers for requesting the server. The server sends back a response, containing the desired data. The data can represent a web page or an attachment like a file or raw data. The browser is responsible for interpreting and rendering the 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.

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

The combination of CSS and HTML can be sufficient for creating a standard web page. However, a modern web application needs to manipulate the web page structure, depending on user behavior, in a more sophisticated way than CSS and HTML currently offer. This type of application needs to use the browser as an execution environment. The environment should be able to change the web page structure, to react to the events, to save an application state, and to control the browser behavior. The scripting language called JavaScript became a browser standard for writing a client-side code inside most browsers as for example in 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) [1] was developed to offer a portable binary-code format for executing programs inside a browser in 2015. WASM targets to enable secure and high-performance web applications. The advantage of WebAssembly is that it is a compilation target for many programming languages. WASM and JavaScript interoperate and utilize in a browser both of the 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 can be compiled to the WASM. 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. Blazor provides a runtime environment, libraries, and interoperability between JavaScript and C# enabling creating dynamic web pages in C#.

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

Peachpie allows using PHP in Blazor. Although Blazor can straightforwardly 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 focuses on identifying use cases that will make use of the integration opportunity between Peachpie and Blazor. The thesis also suggests a solution by creating a library `Peachpie.Blazor` to execute and render compiled PHP scripts in a browser. Blazor is used as an execution environment for these scripts. `Peachpie.Blazor` tries to achieve two goals. The first goal is to implement the support for using compiled PHP scripts with Blazor because there is no existing library that supports the integration. The second goal is to enable the web development on a client side with PHP.

The integration between Peachpie and Blazor can yield the following benefits. The community of PHP developers is significant. Thus, many PHP libraries enable working with user's data, pdf, graphics and offer handy tools. The possibility to migrate the PHP language together with its conventions to the browser will impact developing dynamic web applications thanks to the PHP community and its 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 interoperability which offers more options for developers and future extensions.

The first chapter is about analysis of the related work, alongside with descriptions of the technologies used in the integration. The second chapter analyses running PHP on a client side and other problems related to used technologies. The third gives detailed description of the library's functionality. There are examples that demonstrate how to use all aspects of `Peachpie.Blazor` in chapter 4. In chapter 5, we can see benchmarks that show the limits of the implementation. The last chapter relates to a conclusion of this approach of executing PHP scripts in a browser.

2. Existing technologies

This chapter gives a short overview of the web application functionality. It explores server-side scripting using PHP and client-side scripting using JavaScript in order to obtain observations of user conventions for interaction with web applications. Afterwards it introduces WASM, followed by an existing project that enables the execution of PHP in a browser. There is short information about the .NET platform and C# language. Blazor and Peachpie are introduced in last sections of this chapter.

2.1 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. Its typical characteristics is the statelessness, meaning that a server has to retain information about clients and add additional information to the messages in order to distinguish between clients.

Since the server contains the business logic, the browser has to send the necessary data for the required actions via the HTTP message. The data are usually encoded as a part of Uniform Resource Locator (URL) or in the body of the HTTP message. HTML presents the tag `<form>` that enables interaction with the web application using a web form. The figure 2.1 contains an example of the tag. The `<form>` can contain other tags, which are displayed as various types of fields. The user fills these fields, and the browser sends the data as a new request to the server. The user can specify how the data are going to be encoded. The `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. Here 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` and is encoded in the request body, which does not appear in the URL.

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

The following language description uses Figure 2.1 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 that enables to send an information about the name and attaches a file to the message. The browser sends the message to the server via `POST` method when the form is submitted. The request is handled by *index.php* defined in the `action`. In the end, the script includes a proper ending of the document using the *footer.php*.

As we can see, the PHP code interleaves the HTML code, which has appeared to be a helpful method for data binding. This HTML interleaving allows inserting a 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 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 the type next to the variables. This is because the type system is dynamic. A variable represents just a reference to the heap. Its type is determined during the runtime.

PHP uses superglobals [6], which are a built-in variables accessible from all scopes of the script. Following superglobals are relevant for 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.

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 2.1. In the beginning, header rendering is delegated to *header.php* script. Then, the script renders the body and includes *footer.php*, which cares about the proper ending of the HTML page.

A PHP code can be split by several ways. Despite the fact that object-oriented programming is very wide-spread, the most notable characteristics of PHP are global functions. They are defined in the global scope and accessible from anywhere. The next option is an object inspired by object-oriented programming. A PHP script can include a PHP code from other scripts. They can be recursively

```
<?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 2.1: An example of PHP code in *index.php* file.

included during runtime, where variables remain across the inclusion. Scripts can be composed into a package, which another code can reuse.

2.2 JavaScript

The client-side code needs to control the rendered page and access a web interface providing additional services, which is usually accessible via JavaScript [7] in order to interact with a user. This section starts with the description of loading JavaScript in a browser. It introduces a page representation in a browser alongside with page events. In the end, it presents JavaScript as a scripting language for creating a responsive web page.

We can image a web page structure as a tree. Its nodes are tags or text fragments, and its edges connect nodes with their children. Representation of such a tree we can find on Document Object Model (DOM). Each node is represented as 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. The 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 for the manipulation with 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 a change of 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 mention the ones that are important for the purpose of this thesis. HTML tags are the most common entities which can trigger some events. For example, a button performs an `onclick` event which is triggered when a client clicks on the button as we can see in Figure 2.2. Other events can represent a state of a page like for example `onload` which is triggered when the whole HTML document is parsed.

The 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 [8].

ECMAScript (ES) is a JavaScript standard recommended across browsers. Later on, the text mentions 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 be also run on a desktop by Node.js, a JavaScript runtime environment running outside the browser. Figure 2.2 is used to show the language in a 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. The `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. The usage of the document property can be seen in the example, where it is used to get an element by a given id. 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

```
<!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 2.2: An example of a JavaScript code.

conveniently. There is a handler assigned to the click event in Figure 2.2.

JavaScript is single-threaded language, but allows an effective synchronous execution. This can be achieved by `Promise` object, which is a structure representing an unfinished processes. Large tasks can be seperated into smaller ones in order to improve 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 after an event has triggered it. The single thread is critical for blocking some operations, for example time demanding computatitons which cause thread freezing.

The `Worker` object represents a web worker [9], provided by a browser, enabling it 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.

2.3 WebAssembly

WASM [10] 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 codes can call each other. WASM code has a similar security to a code written in JavaScript because of the usage of the sandbox.

Thread [11] support is being discussed nowadays, and it will probably be added in future versions. After all, new versions of Google Chrome experiments with proper multi-threading support. As the replacement of using the multi-threading can serve using Web workers, mentioned in JavaScript section.

Despite the support of running 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.

2.4 PHP in Browser

The project PIB [2] aims to use compiled PHP interpreter in WebAssembly, which allows evaluating a PHP code. The page has to import a specialized module `php-wasm`. A PHP code is evaluated by using JavaScript API or by writing a specialized script block as we can see in Figure 2.3. PHP can afterward interact with JavaScript using a specialized API. In the figure, the code calls JavaScript `alert` function with the parameter.

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

Figure 2.3: An example of PHP script block executed by the project.

At first glance, it might be a good enough approach, but several parts can be problematic due to PHP semantics. The project 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 a 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.

2.5 C# and .NET 5

The section introduces the Common Language Infrastructure (CLI) [12] before diving into .NET, which is an overused name for several technologies. CLI is a specification describing executable code and a runtime environment for running it on different architectures. CLI contains descriptions of a type system, rules, and the virtual machine (runtime environtment), which executes specified Common Intermediate Language (CIL) by translating it to a machine code. The virtual machine is named Common Language Runtime (CLR). 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 [13] is the latest version of .NET Core, which is a cross-platform successor to .NET Framework. From now on, we 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 environment for executing CIL and many libraries that can represent whole frameworks like is ASP.NET, which aims for web development. A large collection of code is usually compiled into an assembly containing the code and additional metadata. An assembly can represent either a library or an executable program.

Mono aims at cross-platform execution of CIL. Recently, they started to support compilation [14] into WebAssembly. This support allows executing CIL inside browsers. The compilation has two modes. The first mode is compilation Mono runtime with using all assemblies. The second one only compiles 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 is a multi-paradigm language, but its common characteristic is the objected-oriented style. These features cause that C# is a good language for a huge projects that require discipline from developers and need to keep the code understandable and manageable.

2.6 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 [15] that 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 and a server for better functionality. This 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, the Blazor WebAssembly App is going to be refered as the Blazor App. The application can be hosted as 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 choice of a place for the implementation of business logic. Thus, the programmer can move the majority of business logic to the client and use the server for connection to a database,

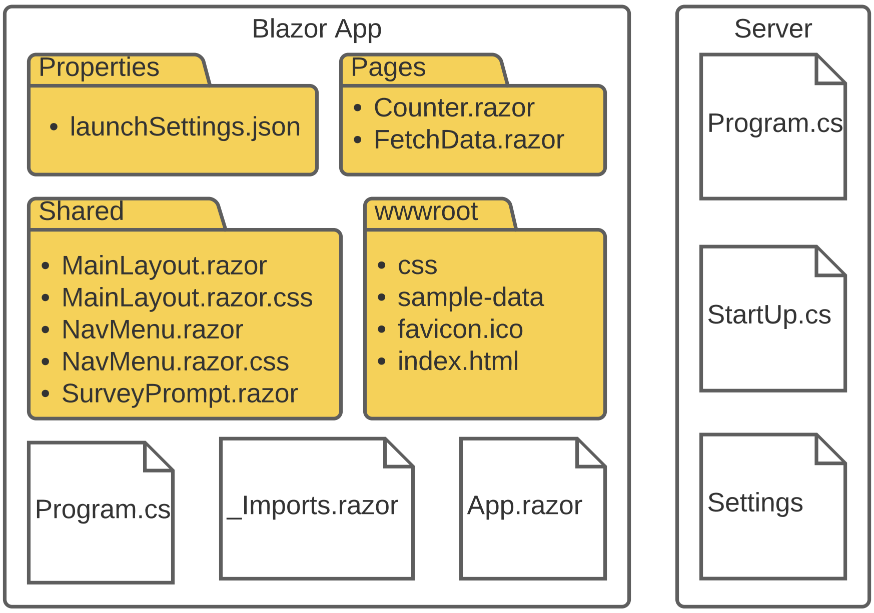

Figure 2.4: Server and WebAssembly App projects.

or he or she 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 2.4, there is a server, which provides the Blazor App to a client. The project contains a standard builder of a host using a `StartUp` class, which configures an HTTP request pipeline processing requests in *StartUp.cs* file. A middleware is a segment of an HTTP request pipeline, which handles some of the functionalities related to request processing. The pipeline contains a middleware providing the Blazor files.

The structure of the second project (Blazor App) in Figure 2.4 is shown to explain basic entities and their interaction with each other. There are few files with the `.razor` extension. Razor is a markup language interleaving HTML with C#. Razor uses a special sign `@` with keywords to identify C# code in HTML. Razor compilation results in a pure C# code representing the web page fragment. An example of Razor is shown in Figure 2.5.

Although the format can be considered as self-explaining, the keywords are going to be described. The first line begins with the `page` keyword determining a part of the URL page. The next keyword is the `inject`, representing a `HttpClient` service injection. The `if` keyword is an control structure interleaved by an HTML code. The `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

```
@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 2.5: An example of Razor page.

`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 2.5 in Figure 2.6.

We can assign the Razor keywords to parts of the code in the figure. The `page` keyword stands for the `Route` attribute. The `inject` keyword stands for a class property marked by the `Inject` attribute. The parameter is assigned by a dispatcher, during the component initialization. The `code` keyword is a part of the class content. Another markup is transformed into calling a specialized method to the `BuildRenderTree` function, which describes the page content for rendering.

The component has several stages [16], which can be used for initialization or action. Virtual methods of the `ComponentBase` represent these stages. We can see the the `OnInitializedAsync` method, which is invoked after setting the component parameters by the `SetParameters` method. The rendering is initiated by calling the `StateHasChanged` method. After that, the `RenderTreeBuilder` is invoked. When the rendering finishes, Blazor invokes the `AfterRender` method, which can manipulate with already rendered HTML tags.

Asynchronous processing should be mentioned because it helps with rendering a page containing 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.

Folders *Pages* and *Shared* contain parts of Blazor pages written in Razor. The *_Imposts.razor* contains namespaces, which are automatically included in other `.razor` files. The next folder is the *wwwroot*, containing static data of the application. The *index.html*, cares about loading parts of the Blazor application to the browser. We call all of the static files in the *wwwroot*, additional JavaScript scripts, and WASM runtime environment Static Web Assets.

The following paragraphs describe the loading of Blazor into the browser to fully understand the interaction between Blazor and the browser. There is the server, the Blazor App, and other optional user defined projects. When the server is started and a client 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 2.7. The first step is to load all resources, which are defined in a separated file. Blazor cuts all unnecessary `.dll` files to reduce the size. For this reason,

```
[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 2.6: Razor page generated to the C# class.

all of the `.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 of the `.dll` files containing the whole application (Blazor App with referenced libraries). The supporting scripts initiate the runtime and execute it. The runtime environment includes the `.dll` into the application and calls the `Main` method in the `Program.cs` defined in Blazor App project. We can see the process in Figure 2.8.

`Main` method uses the `WebAssemblyHostBuilder` to set the application. It defines services, that will be able through the dispatcher. It sets a root component,

```
<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 2.7: *index.html* provided by the server

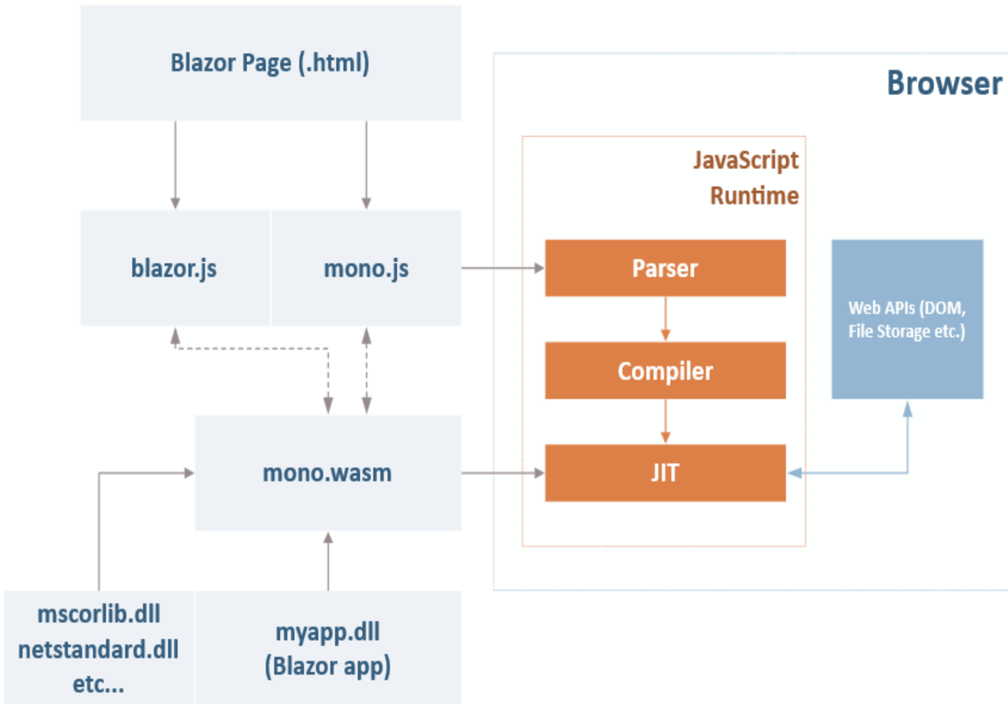

Figure 2.8: Running a Blazor WebAssembly App on a client side [17].

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 environment to offer interoperability with JavaScript.

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

The end of the section describes page navigation [18], rendering, and handling events. The navigation can be triggered by an anchor, a form, or by filling up the URL bar. The URL bar is handled separately by a browser. JavaScript can influence the remaining 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. The user can add listeners to the handler, but the Router implements default behavior for navigating. The `Router` finds all of the components which implement the `IComponent` interface and tries to render the page according to path matching the `RouteAttribute` of the component whenever the navigation is triggered. It creates an instance of the class and fills in its parameters. The `Router` calls the `BuildRenderTree`, which enables running the rendering process. Previously created intences of the components are disposed. The navigation can be redirected to the server if there is no match.

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 interoperability support. Blazor provides API for invoking JavaScript functions and vice-versa. `Renderer` provides the `RenderTreeBuilder` for describing page contents. The builder provides an API for adding various types of content to the `Batch`, which is a specialized structure for describing previous and present DOM. Although DOM changes are performance-expensive, a diff algorithm [19] recognizes and tries to reduce the updates in the `Batch`. The usage of the `RenderTreeBuilder` is complicated, because it has to be adapted to the algorithm [20]. 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 uses sequence numbers for parts of HTML to identify modified sections. Sequence numbers are generated in `RenderTreeBuilder` instructions during a compilation. The 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.

Blazor brings a new type of library called Razor Class Library (RCL), which differs from normal libraries by the *wwwroot* folder handling [21]. During the compilation, Blazor Software Development Kit (SDK) generates a configuration file comprising paths to *wwwroot* folders. The server provides static web assets of these folders to a client by default. The SDK only involves the folders contained in RCL libraries and WebAssembly projects into the configuration file.

2.7 Peachpie

Peachpie [4] 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 describe the basics.

Because different 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 the `Context` class as a state of the script while it runs. The `Context` class consists of superglobals, global variables, declared functions, declared and included scripts. It also manages input and output, where the resource can be chosen. The `Context` class can also be considered as a configuration of the incoming script execution. All of the information about a request can be arranged to mock every situation on the server side. The possibility of saving the `Context` class and using it later is a significant advantage. The class provides 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 its 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.

At the time of writing this thesis, there are certain limitations emerging from differences between the languages and the stage of Peachpie 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. C# constructs like structs and asynchronous methods are undefined in PHP.

3. Problem analysis

The chapter divides the analysis into three steps. In the first section, we think of potential users of the integration in order to define realistic use cases for them. Four use cases describe the user's intentions. Then, requirements are specified based on the use cases. In the last section, we propose a high-level architecture of the `Peachpie.Blazor` library aiming at utilizing Blazor and Peachpie to cover the requirements.

3.1 Use Cases

In this section we remind technologies of interest to introduce a context of our use cases. PHP is used for server scripting, where it is designed to process a request, create the website, and send it back. Blazor is a web framework for creating a client-side UI using C#. Peachpie is a PHP compiler, which compiles a collection of PHP scripts, representing a standalone project, to a .NET assembly.

A user persona [22] is a description of an imaginary user, which represents the needs of some group of users. In next paragraphs we introduce four user personas to cover use cases that help us to identify the requirements.

The first persona is a C# programmer, Blake, excited for Blazor. He has already got acquainted with the `Peachpie.Blazor` library.

The second persona is a PHP programmer, Alice, who has no experience with Blazor but knows Peachpie basics. Alice creates standard websites written in PHP, where she uses techniques introduced in the PHP section. Our first persona, Blake tells her about our library migrating the scripts to a browser using Blazor and Peachpie. She is excited by the approach and looks forward to using it. However, she does not want to learn the Blazor framework.

The third persona is a PHP programmer, Bob, who has already tried to write a simple website using Blazor and knows Peachpie basics. He creates standard PHP websites similar to Alice's. Our first persona, Blake, tells him about the `Peachpie.Blazor`, and Bob's wish is to use the library to help him inject his PHP scripts into Blazor websites. Occasional work with the Blazor framework does not bother him, but it should have appropriate difficulty to his skills.

The fourth persona is an enthusiastic PHP programmer, Chuck, who has advanced experience with Blazor and knows Peachpie basics. He does not avoid exploring new technology to utilize all of their aspects. Blake tells him about `Peachpie.Blazor`, and Chuck wishes that the library offers him to collaborate with Blazor by PHP.

These descriptions should help us determine the following use cases, which are realistic to them. The first use case called **Web** is aiming at Alice. We suppose she has a simple PHP website, which contains some information about her company. The website does not work with a database and consists of pages containing images and references interconnecting them. Some of the pages are adjustable by specifying the query part of the URL, and they include other scripts to add some basic layout. In a specific case the website notices many accesses, and Alice wants to migrate the website to a client side in order to save server resources. The migration should download most of the website to a browser. Afterward, navigation between scripts and script execution should be maintained on the client side. Even more, Alice does not want to adjust the website´s client side too much, and she wishes for simple instructions that are understandable by a novice.

The second use case is called **OneScript** and is aiming at Bob, who already has some experience with Blazor. He wants to contribute to an existing Blazor website. He has a great idea of adding a new widget, displaying the user's graph using HTML and CSS. Because he is used to PHP, he wants to implement it with a few PHP scripts, which use some supporting libraries. The idea consists of letting the user choose either to load the graph data from a file or to generate a predefined graph as a demo. After that, the widget renders HTML markup representing the graph. Bob uses forms to interact with a user, and he is not willing to learn JavaScript or interoperability between PHP and Blazor. Thus, he needs a mechanism, which offers interaction with a user and uses standard PHP conventions mentioned earlier.

The third use case is called **WebGame** and is aiming at Chuck. He wants to create a real-time web game similar to Asteroids written in PHP. He decides to target on a client side and utilizes Blazor, Peachpie, and the `Peachpie.Blazor` library. A client-side execution should prevent network latency by loading the game in the beginning. After that, the game will be independent of the network connection because of running and saving the game state by a browser. PHP programmers have not been used to saving variables or defined functions across scripts because of the HTTP policy mentioned in the PHP section. However, Chuck utilizes state persistency to save states of all game entities in variables. Because he has previous experience with Blazor infrastructure, he will appreciate utilizing all Blazor aspects to run this game.

The illustration of the fourth use case called **AllTogether**, is shown bellow in

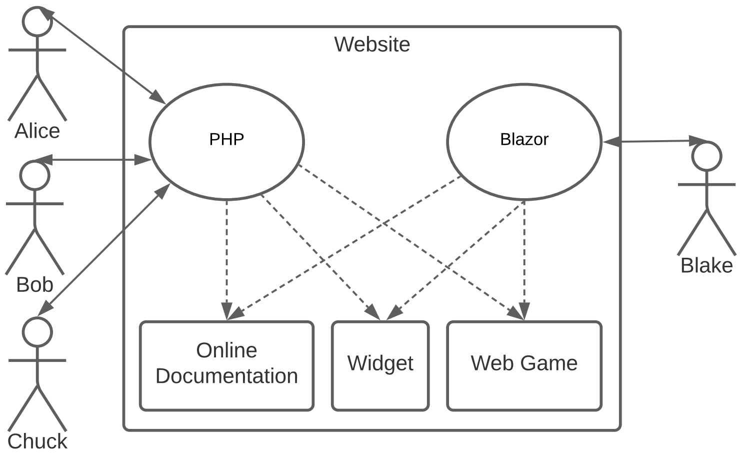

Figure 3.1: The `AllTogether` use case describing the combination of the all previous use cases.

Figure 3.1. A double-headed arrow represents the language the person had used. A dashed arrow represents a possible usage, where the head aims at an implementation of a website part written in the language connected to that part. The goal of the use case is to allow collaboration between PHP and Blazor programmers, where the difference of languages is not a barrier. Image two teams creating a web application. They agreed on developing a client-side web application, where both teams aim at different parts of the website. For example, one of the teams wants to create a fun zone where a user can play a web game, like Asteroids, and the second team wants to create an online documentation about the game and the widget for the graph representation displaying a user's score. Because Blazor targets client-side web applications, they want to utilize Blazor. Unfortunately, these teams use a different favorite language, where the first team uses PHP and the second team uses C#. Even more, these teams want to contribute to any part of the Blazor website, meaning that doing the fun zone can be handled by either the C# or PHP team. They need some environment where the PHP team can code alongside with the Blazor team, and they can focus on an arbitrary part of the web application. We can see the intention in Figure 3.1 where each team can create a part aiming at the web game, the online documentation, and widget. The PHP team consists of Alice, Bob, and Chuck, having different skills with Blazor, so the environment should reflect it. Even more, Blake should be able to manipulate their part of the application to customize it using C#. For example, he should be able to change the layout of the website without complex refactoring.

3.2 Requirements

The goal of this section is to describe requirements based on mentioned use cases. If the proposed library covers the requirements, then PHP scripts will become a valuable part of a Blazor website.

Navigation is the first requirement that our library should provide. Figure 3.2 shows navigation possibilities. Basic functionality should provide script routing, which finds a script by its name, executes it, and displays the output in a browser. This intention is illustrated by the first rectangle containing a collection of scripts in the figure. The library should offer a straightforward router making a PHP website accessible, as we can see in the figure. The simplicity is a necessary condition for the *Web* use case and should be reflected. Blazor should navigate components defined in *script.php* because of the the *WebGame* use case, which uses Blazor structures.

Reusability of a script is an important feature to make the *OneScript* use case more useful. Thus, Bob can insert the widget in different parts of the website, meaning that he can create a new web page containing some content and insert the widget into it, as we can see in Figure 3.2 where the Blazor component is generated from a PHP script and a Razor file.

Interactivity with a user is necessary in the *WebGame*, and the *OneScript* use cases. The library should enable using common conventions in PHP, like forms, and be able to utilize Blazor features providing the interaction as well.

Rendering should be maintained in two ways. The first way aims at the *Web* use case when a script output is transparently displayed as a web page or its fragment. The approach hides Blazor infrastructure for rendering a markup and makes creating a UI easier for PHP programmers. The second way aims at the *WebGame* use case when `Peachpie.Blazor` provides an interface for the interaction with Blazor. It is also necessary when we want to use already defined components in a PHP code. The rendering should be effective due to the high frame rate of the game.

State preservation should be available for creating a web application by a collection of scripts saving their variables after the execution. This feature is not typical for PHP because of PHP policy and conventions, where programmers are used to delete the variables and function definitions after the request termination. The state described by the variables needs to be preserved in order to interact with a user in a client-side application. For example, the *WebGame* use case uses variables to save the game state. However, we have to distinguish these situations where the feature is necessary.

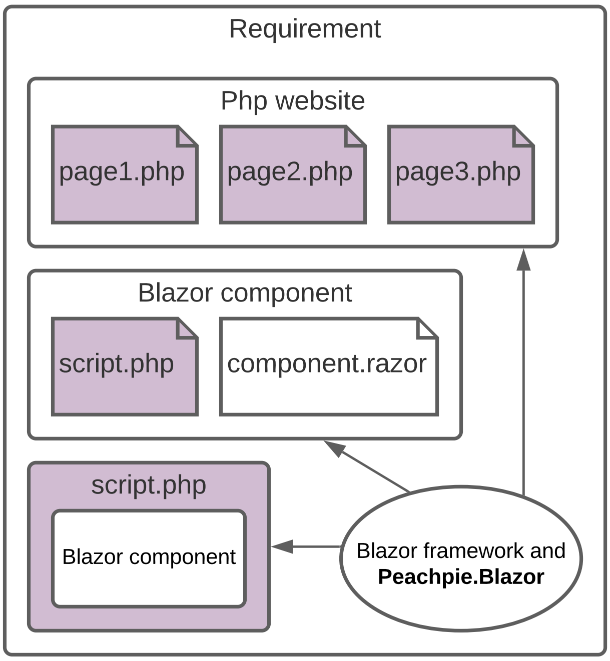

Figure 3.2: The requirement describing navigation between different types of entities. `Peachpie.Blazor` with Blazor connects these types into a single website, where they can live together.

Server simulation should be the main advantage of the library. Superglobals are commonly used methods used to obtain information about navigation or about submitted data. The library should support superglobals, for example as is the *Web* use case, where the website uses information about URL query part, via the `$_GET` variable, to make decisions.

Forms should be maintained by the library. The forms are usually sent to a server, but the library should handle them on a client side, where they should be provided to PHP scripts. After navigating to a script defined in the `action` attribute, the script should access the form data. A client can upload files by a form. Thus, the library should provide file management accessing and downloading them.

Interoperability between PHP, Blazor, and JavaScript should be supported in those situations where forms, the server abstraction, or Blazor are not sufficient. We need some representation of Blazor in PHP, which the *WebGame* use case will use for interacting with a Blazor. JavaScript is essential for client-side applications, and PHP should be able to use its features.

3.3 Architecture

The basic principle of our approach consists of PHP scripts compilation into a .NET assembly by Peachpie. After that, Blazor App references to `Peachpie.Blazor`, which is a support library providing a mechanism for navigating and executing the scripts. Then, a server provides the application to a browser, where Mono runtime executes it. We describe the architecture from the viewpoint of compilation time and runtime environment.

When we think about PHP script compilation, there are two possibilities. The scripts can be compiled ahead of time or can be referenced from a Blazor App. The second way is to regard the scripts as Static Web Assets and load them into a browser as separated files. Afterward, the Peachpie compiles and executes them. Both approaches have different advantages. Thus, there is no silver bullet. The first approach saves time by ahead compilation and compilation check. However, the second approach can save browser memory when the web application is larger and a client uses only a part of it. We are inclined to the first approach because the static compilation is a standard way in Peachpie. We think that the first approach is valuable for the use cases mentioned earlier. The *Web* use case wants to save additional requests. The rest of the use cases intends to utilize small amounts of PHP scripts as a part of the website, so we suppose that the smaller size of resulting scripts is insignificant in contrast with the compiled Peachpie assembly, containing all of them.

We have to figure out how to attach a PHP code, which is compiled into the assembly, to the Blazor App. Although Peachpie supports calling functions written in PHP from Blazor by default, 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, meaning that it should be easy to use it with the Blazor pages. A Blazor page consists of components, which can collaborate with each other. Thus, they can be utilized to represent PHP scripts. Components can be arbitrarily put together, which offers to place our PHP code in the desired place in the Razor code. Even more, a root component, `Router` by default, can be replaced by the component representing PHP scripts. Afterward, scripts will compose the whole Blazor website content. The component provides a sufficient Blazor interface for rendering control and interaction with a browser.

We can think about how to represent PHP scripts as components. We can consider a certain type of the component, that will provide the abstraction for all

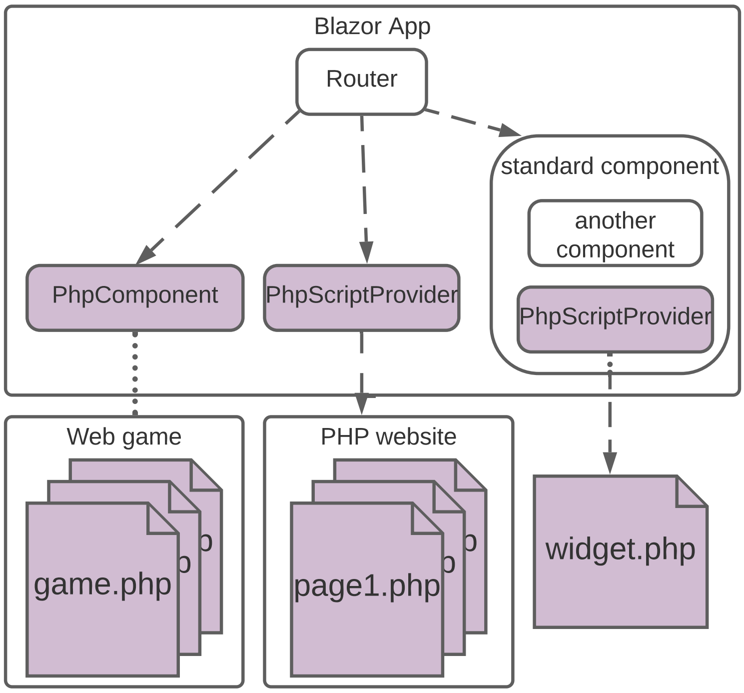

Figure 3.3: Components representing PHP scripts. Arrows represent navigation. Dot lines connect a runtime object with the implementation.

of the PHP code in use cases. A problem with this approach is that the use cases need different levels of abstraction. The *WebGame* use case wants to use the component for accesing the Blazor interface from PHP code. The access should contain identical or similar options, as are given in a C# code. The *OneScript* use case requires to free a programmer from Blazor. Thus, the component should be an adjustable provider finding and executing 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 the provider role in a Blazor App. The provider can behave either as default Blazor `Router` or as a routable component, which enables the navigation of PHP scripts. As a consequence, we need to create more types of components providing abstractions for the particular use cases. However, only one type of component can manage all provider roles due to their similar rendering transparency. The library will contain two types of components. The first one wants to bring Blazor to PHP in order to utilize the whole environment. The second one aims at presenting a transparent execution of standard PHP script without knowing about the connection between Blazor and PHP. Our intention is illustrated in Figure 3.3.

We focus on the first component, which we call `PhpComponent` due to the effort of moving the component concept to PHP. `PhpComponent` aims at the third use case. Despite language differences, we can utilize the common concept of classes and inheritance because 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 behavior from PHP. PHP class can inherit the `ComponentBase` class and use its methods in the same way as C# class. The inheritance offers the required interface in the *WebGame* use case. At the time of writing this thesis, there are also subproblems related to the differences between the languages. The current Peachpie version does not support some C# specifics fully. The reason can be hard or even an impossible representation of C# entities in PHP. The library should provide PHP support to allow using the parts of the Blazor interface, which can not be used in PHP directly.

The second type of the component is called `PhpScriptProvider` expressing an environment for executing standard PHP scripts. `PhpScriptProvider` solves the requirements of the remaining use cases *Web*, *OneScript*, and *AllTogether*. The provider should be able to navigate and execute PHP scripts. Because the remaining use cases try to hide the integration between PHP and Blazor, the provider should support the following features. It should pretend a server behavior, which copies everything in the output of PHP script to an HTTP response body rendered by a browser. Superglobals are often used for obtaining additional information given by the user. Thus, an ability to fill the `$_GET` variable with the URL query part is important. It should change a standard form functionality, which is sending the form to a server, to save the form information into superglobals, and executing the script again. We target to load and save files submitted by a form transparently in order to provide similar comfort to execute the script on a server side. A possibility of saving the script context to the next execution is a new opportunity how to keep an application state in PHP script. The following paragraphs describe the provider modes. These modes are intended to solve the *Web*, the *OneScript*, and the *AllTogether* use cases.

The first mode is called **Router**, and aims at the *Web* use case, where the implementation is inspired by a GitHub project [23]. It enables to set the provider as a root component. It handles all navigation events, determines the script name, finds it, and executes the script. Components defined in PHP code can also be navigation targets.

The second mode is called **Script**, and aims at the *OneScript* use case. It enables the provider insertion into a Razor page. Afterward, the provider executes the specified script.

The third mode is called **ScriptProvider**, and aims at the *AllTogether* use case. 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.

These observations lead us to make sure having two different components is a rational way how to separate the problems and offer an understandable difference between the components.

4. Solution

This chapter describes the complete process of `Peachpie.Blazor` functionality, covering the use cases. It starts with an overview of the library parts. Then, it gives a detailed description of each part.

Peachpie.Blazor is an RCL library containing components mentioned in the previous section with an additional support code. It is provided as a NuGet package containing an API for including PHP scripts to the website. It defines a collection of support classes representing HTML entities, that makes the rendering easier. A part of the support code is the JavaScript script, which is linked to the beginning of the HTML document, providing helping functions for form handling and securing interoperability with PHP. Then, the resulting Blazor website consists of three projects forming a .NET solution. In Figure 4.1, we can see these projects as green rectangles. The *Server* project references *Blazor App*, containing a part of the website, and *Peachpie.Blazor* library, containing the support code. The server cares about providing the Blazor website and its Static Web Assets. The *Blazor App* project becomes the environment for running PHP scripts in a browser. The rendering process and the website composition, where the scripts are evaluated, are meant by the environment. The project references the Peachpie project containing the programmer's PHP scripts and the *Peachpie.Blazor*,

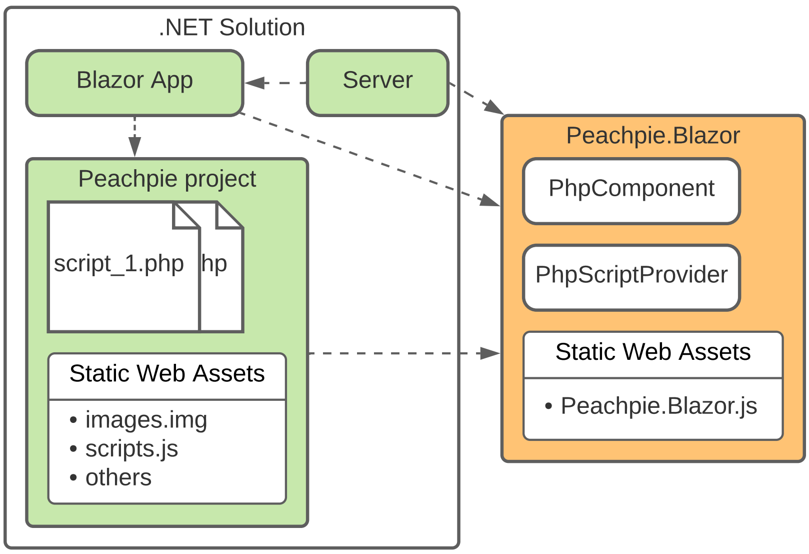

Figure 4.1: The .NET solution infrastructure. Green rectangles represent projects. The orange rectangle represents a NuGet package. Arrows represent references.

content of which is used to maintain the scripts. The *Blazor App* inserts the scripts using the components defined in the library.

The first section aims at the server settings providing a Blazor application using PHP scripts and the *Peachpie.Blazor* library. The second section talks about `PhpComponent`. It describes the resulting implementation and solves problems, which occurred during the implementation. The last section aims at `PhpScriptProvider`. It suggests a convenient way how to include the scripts into a browser, and presents the component design.

4.1 Server

The server has to be set in order to provide additional static resources of a Peachpie project containing PHP scripts because the WebAssembly SDK ignores the *wwwroot* folder of libraries except for RCL. Thus, we create the `UseAdditionalWebStaticAssets` extension method of `IApplicationBuilder`, which inserts middlewares providing the resources into the request pipeline. Its parameter is a configuration obtained from *appsettings.json*, which is a part of the `BlazorApp.Server` project. We can see an example of the configuration file in Figure 4.2, which defines a path to a folder and a base path used as a prefix of HTML document references. The path can be absolute or relative to the current working directory. Afterward, these resources can be referenced by URLs and downloaded from the server. For example, an HTML document can reference an image with the path `WebGame/PHPScripts/wwwroot/image.jpg` as `/Asteroids/image.jpg`. JavaScript helpers of our library can be found in the

```
{
    ...
  "AdditionalStaticWebAssets": [
    {
      "Path": "A//Path//To//Resources//Directory",
      "BasePath": "/Asteroids"
    }
  ]
  ...
}
```

Figure 4.2: Fragment of configuration file defining additional resources.

wwwroot folder, which is transparently provided to a client side due to an RCL library type and the SDK.

4.2 PhpComponent

This section introduces several issues caused by many factors, for example by the difference between the PHP and C# language. We analyze these issues and suggest helping concepts, which form the resulting `PhpComponent` class. The main objective is to utilize the Peachpie feature allowing to inherit C# classes, like `ComponentBase` representing a Blazor component, to provide the full Blazor API to PHP.

4.2.1 PhpTreeBuilder

The first issue is caused by the difference between the PHP and C# language where Peachpie tries to compensate it, but it is not its main target. For example, C# structs have not a representation in PHP. Structs are necessary to work with `RenderTreeBuilder`, which contains API for adding callbacks handling element events, as we can see in Figure 4.3. This API uses method overloading in many methods. However, PHP does not support method overloading. `AddAttribute` is an example where various types of the attribute value can be put in. One of the values can be `EventCallback` struct representing an event handler. The struct contains a static property, `Factory`, which is a class containing methods for creating callbacks.

Peachpie enables using structs in PHP code. However, there are limitations at the time of writing this thesis, which force us to make workarounds. We try to rewrite the previous example in PHP code using Peachpie. We create a component, which inherits `ComponentBase`. Afterward, we override the method for building a render tree and implement the body. Peachpie does not allow us

```
__builder.OpenElement(5, "button");
__builder.AddAttribute(7, "onclick",
        EventCallback.Factory.Create<MouseEventArgs>((object)this,
                (Action)IncrementCount));
__builder.AddContent(8, "Click me");
__builder.CloseElement();
```

Figure 4.3: Fragment of code adding a button element with an event handler.

to access a static property of struct, which is necessary for obtaining an object providing callbacks representing event callbacks of HTML elements. Another issue is using a method for creating the callback, which has many overrides. Peachpie can not choose the correct version, which results in a runtime error. We tried to use many workarounds, which used helper classes trying to avoid these issues, but it is impossible to use some Blazor API directly in PHP code.

To make the example work, we can hide the struct from PHP code by implementing a C# helper method using the struct. The method should have only parameters compatible with PHP types. The overloading can be replaced by a different method name for each overload. Afterward, Peachpie allows us to call the methods from PHP code. This approach can be used in the `AddAtribute` method. Defining a new method for each overload is a reasonable approach due to a small number of overloads. The `RenderTreeBuilder` does not allow us to inherit it because it is sealed. For this reason, a wrapper is created containing the builder and defining method for each overload, which calls the original method in the C# code. This decision leads us to make `PhpTreeBuilder` wrapping the original builder.

4.2.2 Collection of helper classes

The next issue relates to the rendering time. `RenderTreeBuilder` provides a method for adding arbitrary markup text. The text can contain `<script>`, but its content is not executed. At first glance, one can see the method as a convenient way to render the whole content, avoiding using other dedicated methods for building the tree. These methods accept a sequence number used by the diff algorithm. Although using the one method for rendering, the whole component causes slow rendering, which is critical in some applications, as are for example games. The diff algorithm relies on marking the blocks of markup by sequence numbers for optimization in page updates. When there is only one big block, the diff algorithm can not do anything better than generate an update, which renders the whole page. This issue can be seen in the Benchmark section comparing the difference between using one method and utilizing all methods.

Because the builder usage can be complex, `Peachpie.Blazor` introduces a collection of classes for representing tags, helping implement the code using the builder for rendering. The class diagram is shown in Figure 4.4. The main idea is to implement the `BlazorWritable` interface, which writes the class content into the builder. An example of the class is `Tag`, which represents an arbitrary tag. The tag contains its name, attributes represented by `AttributeCollection`,

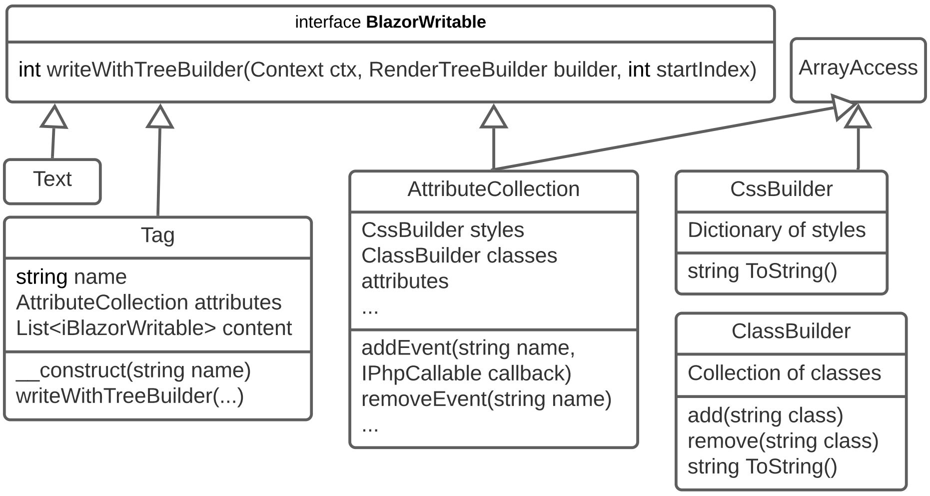

Figure 4.4: Class diagram of supporting library for writting tags.

and inner objects implementing the `BlazorWritable` interface. Because a tag can contain other tags using sequence numbers, we have to keep the currently used sequence number used in the diff algorithm. For this purpose, the `writeTreeBuilder` method gets the actual sequence number and returns the last unused number. This API should hide separated class logics for rendering. The helper classes give a basic implementation of this method, which renders the content with a dynamic sequence numbering. However, a programmer can override the method because sequence numbering is impossible to predefine in advance to make the most effective updates. The next object implementing the interface is the `Text` class representing a text. `AttributeCollection` offers a convenient interface for working with attributes by implementing the PHP `ArrayAccess` interface enabling indexer. Because string values of `style` and `class` attributes are more complicated the library provides dedicated classes for working with these attributes, used by `AttribureCollection`. `CssBuilder` and `ClassBuilder` provide API for creating these values and then format them into the HTML style.

The next barrier is assigning handlers to C# events in PHP code. Peachpie does not either support accessing the events. Thus, a PHP code can not directly use a class like `Timer`, which is useful in the *WebGame* use case for updating the screen every defined period of time. The issue can be solved by helper methods defined in a C# static class, `EventHelper`. They accept the object providing some events, handler, and event name. Afterward, the reflection is used for obtaining the desired event by name from the object. Then it assigns the `IPhpCallable` handler to it. Because `Timer` is a common object, the library contains an additional PHP wrapper class, which uses the timer. Then a programmer avoids using the workaround defined above.

4.2.3 Interoperability

The last feature to discuss in this section is interoperability between JavaScript and PHP. Blazor allows calling JavaScript functions from C# [24] and vice-versa [25]. Thus, a Blazor service, `IJSRuntime` injected by the dispatcher, can be utilized to call JavaScript functions. The service offers a specialized API for that. Calling PHP from JavaScript is more complex. Peachpie enables calling PHP function by the `Call` method of `Context`. The context finds already defined methods contained in the included PHP script and executes it with the current context. When we want to call a C# instance method from JavaScript, we have to have the reference, supplied by the framework, for calling it. Additionally, the called C# method has to be marked with a `JSInvokableAttribute` during compilation. The reference can be assigned from C# by a method, which creates it and uses it as a parameter of a JavaScript function. These conditions lead us to create a new Peachpie context, `BlazorContext`, which inherits the original Peachpie context and provides the `CallPHP` method marked by the attribute and calling the PHP function based on parameters. The context requests the mentioned services provided by the dispatcher and sets the reference by calling predefined code in `Peachpie.Blazor.js`. The context will be the Peachpie context of the component. Thus, it enables to call PHP methods defined in this context from JavaScript by using the reference. The advantage of this approach is that there can be two components inheriting `PhpComponent` with the same context when their context is set to the same instance. Thus, we can call their functions only by one reference.

4.2.4 Summary

This section summarizes the architecture of `PhpComponent` and main implementation idea of the *WebGame* use case. The component hides the original function for rendering and replaces it with our version of the builder, as shown in Figure 4.5. It results in transparent usage of the builder in the inherited class. The builder is just a wrapper, so the programmer can use the original builder by accessing its property, the `Builder`. Additionally, there is a collection of classes for creating tags, making builder usage easier. For assigning PHP handlers to C# events, there is a universal helper class, `EventHelper`. Furthermore, the last feature is

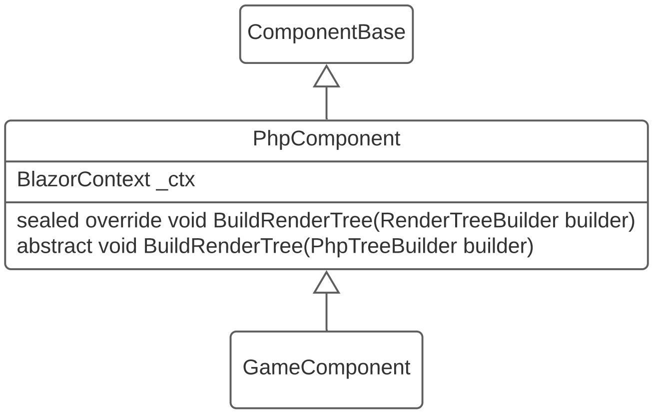

Figure 4.5: Class diagram of the use case implementation.

a timer wrapper, which uses the C# timer, offering a convenient API. Interoperability with JavaScript is provided by our predefined API using `BlazorContext`. These features are sufficient for implementing a game described in the *WebGame* use case. The game screen is updated by calling `StateHasChanged` in a PHP function, which is an `Elapsed` event handler of the timer. The update consists of evaluating positions of game entities, which use the helper classes representing HTML entities. Lastly, context preservation is used to keep the game state.

The last necessary thing is to pass assembly references containing the components into `Router`, which is a standard duty in Blazor.

4.3 PhpScriptProvider

The beginning of this section introduces the main component parts, which give us an overview of the component composition. Component duties like navigation or script execution are divided into subsections because the component consists of many processes, which are complex to describe at once in the structure. The component functionality will be explained in these sections.

Figure 4.6 describes the connections between the main parts. `PhpScriptProvider` is a class representing a Blazor component. The component manages the following features.

- It handles the navigation.
- It finds the script by name based on provider mode.
- It creates and keeps a PHP context, which is used for script execution.

- It executes the script.

These duties contain several steps, which are maintained by helper classes. As we can see, there is `PhpComponentRouteManager`, which finds the components, inheriting `PhpComponent`. It enables navigation of Blazor components defined in PHP scripts. The next part is `BlazorContext`, already mentioned in the PhpComponent section, which is Peachpie `Context` designed for Blazor environment. The context constructor accepts several Blazor services like `IJSRuntime`, enabling interoperability with JavaScript. The context initializes superglobals based on URL and on submitted forms, manages files uploaded by a form, and controls `BlazorWriter`, which redirects the script output to the render tree. Lastly, `FileManager` reads submitted files, downloads them, or deletes them from browser memory.

The following paragraph zooms in the `PhpScriptProvider` structure in order to better understand processes maintaining the features. The provider consists of many properties. Some of them are injected by the dispatcher like `NavigationManager` or `IJSRuntime`, which is a service providing interoperability with JavaScript. Others can be parametrized, like the `Type` determining the mode of provider, `ContextLifetime` determining the persistency of the script context, or `ScriptName` determining the executing script when the mode *Script* is set. These properties influence the component methods. The first method is `Attach`, which assigns a render handle providing `RenderTreeBuilder`, and registers a navigation handler creating the context and calling `Refresh`. The second method, `SetParameters`, handles calling `Refresh`, and creates the context as well. `Refresh` finds a script or a component based on properties, assigns the superglobals, and calls the `Render`, which renders the component or executes the script and its output is rendered. `OnAfterRender` cares about enabling the forms to send data back to Blazor. Some of these methods are called by Blazor framework itself. The way of calling them provides the component a certain

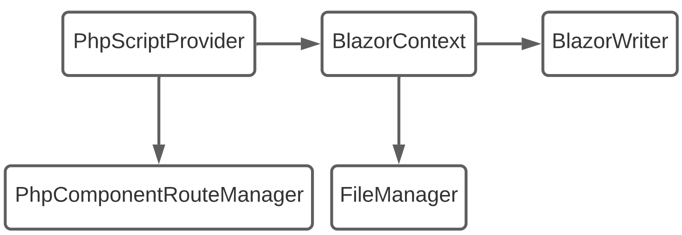

Figure 4.6: Diagram illustrating usage of PhpScriptProvider main parts.

lifecycle.

4.3.1 Navigation

The section explains navigation in `PhpScriptProvider` for each of the provider modes. We have to clarify how the component is instantiated and maintained by Blazor. There are two ways of using the component. The first of them is to set it in `WebAssemblyBuilder` as a root component, which is rendered after the first application launch in Blazor. The component is alive for the whole life period of the application because there is no `Router`, which disposes components representing a previous page. It results in calling the `Attach` method and the `SetParameters` method only once. The second way is to use a Razor page containing the component and let the navigation to the page on `Router`, which is a root component by default. When the page is navigated, the component is instantiated as well. The difference is the possibility of calling the `SetParameters` method multiple times when the page is parameterized. Then, when the parameters are changed, Blazor automatically calls the inner component `SetParameters` methods if the components have at least one complex type. This action is caused because the Blazor framework can not decide if the parameters, which are complex types, were changed.

The *Router* mode is designed to be used when the component is a root component. Then, it is certain that `Attach` and `SetParameters` methods are called only once. Thus, the navigation handler is registered in the `Attach` method, which should handle further navigation. When the navigation occurs, a new context is created, if the `ContextLifetime` property is set to `Persistent`. Then, the `Refresh` method is called. `SetParameters` creates a new context and calls the `Refresh` method as well, but the method is called only once at the beginning of the application. Then the `Refresh` method parses the query part of URL [26], obtained from `NavigationManager`, and gets the script name from the URL. When the script name is determined, the method calls the `Render` method, where it decides if it is a script or a component defined in a script based on the `.php` extension. If the extension is missing, it tries to find a component defined in scripts, which has the correct `RouteAttribute` by `PhpComponentRouteManager`. Otherwise, it asks the Peachpie context for obtaining the script representation as `ScriptInfo`. Additional manager duty is to assign assembly references containing scripts, to the context, at the beginning of the application. The context does not have to know the script name, or the component does not have to exist. Then, the `Render` method renders a predefined *not found* page, which can be set by `PhpScriptProvider` parameters. Otherwise, it renders the script or the component. Additionally, when the component is navigated, the method sets its context by the current provider context. It results in using the interoperability by either provider or the component because there is the same context reference in JavaScript code.

Next modes, *ScriptProvider* and *Script*, are similar. They are defined in a Razor page and initialized when the page is navigated. The difference is finding the script by name, where the *Script* always uses the name defined in the component parameter and *ScriptProvider* finds script based on URL. As we said, the `SetParameters` can be called multiple times, so the method calls the `Refresh` only the first time. Additional rendering is initiated by the navigation handler. Thus, when the navigation occurs, it finds and updates the page, or the component is disposed if the `Router` matches another Razor page. A new context is created based on the `ContextLifetime` mode. It differs from common PHP behavior, where the context is disposed after scritps handles the request. This unusual approach allows calling functions defined in the context after the component is rendered, which is a part of JavaScript interoperability where PHP functions can by called by JavaScript. We should check if the component has not already been disposed by `Router` before calling the `Refresh`. Obtaining the script is similar to *Router* mode.

4.3.2 Script Execution

We start with `BlazorWriter`, which inherits `TextWriter`. The inheritance allows using the writer as a `BlazorContext` output writer, which manipulates with a script output. The writer consists of a buffer and `RenderTreeBuilder`. The main usage is to write any string to the writer, which adds it to the buffer. In the end, the writer flushes it into the builder by `AddMarkUpContent`. It results in treating the whole script output as one modification by the diff algorithm, which causes the whole page update instead of smaller necessary updates. This approach is chosen due to the following limitations. `AddMarkUpContent` does not allow to add incomplete markup text, meaning that tags are not properly closed. Thus, we can not divide the text into smaller parts because of the HTML nature, where tags are coupled by other tags. The second possibility is to parse the output and recognize the types of HTML entities, which can use specialized builder API. However, it is not suitable because of parser complexity. It is important to dispose the writer after the rendering because the same builder can not be repeatedly used.

`BlazorContext` maintains the writer and the interoperability with JavaScript.

It provides methods for initializing and ending the rendering. The script represented by `ScriptInfo` is executed by its method `Evaluate`. This method accepts the context, which maintains the script output by redirecting it to the render tree. The context also allows setting superglobals like `$_GET` to provide the query part of URL and turns forms to client-side handling, which is described in the following section.

4.3.3 Forms

Typically, web forms are not handled by Blazor but they are sent to the server. JavaScript interoperability is used to evaluate them on a client side. It starts in the `AfterRender` method, where it calls our JavaScript function, which finds all already rendered forms and assigns them an event handler for submitting. When submit occurs, the handler collects all data from the form, performs ordinary navigation to the page defined in the `action` attribute, and prevents default behavior, which is sending the form to the server. When the navigation is handled by `PhpScriptProvider`, it gets all collected data and assigns the context superglobals by them. Afterward, the script is executed, and it can access the superglobals.

The previous paragraph did not explain the file management due to its complexity. We describe it now. When a user loads files by a form, JavaScript obtains only the list of files. When a programmer wants to read the content, he or she has to use a reading operation, which is done asynchronously by `Promise` mentioned in the JavaScript section. Thus, when we get the data during navigation, the page rendering has to wait until the content is read. This operation could take a long time, so the page shows old content. For this reason, we provide an additional parameter for defining the content, which is shown during navigation. An alternative is initializing reading by a PHP script when it is executed. It uses interoperability between PHP, C#, and JavaScript in order to call desired reading methods. Unfortunately, Blazor does not allow us to wait until the reading operation is done, and we have to provide callbacks, which handle the end of the reading. We suppose that it is confusing for potential PHP programmers who will use `Peachpie.Blazor` to define PHP callbacks. We decide to prefetch the file content before the execution to provide the data synchronously in a PHP code.

Interoperability in the provider can be achieved in a similar manner as in `PhpComponent`. The difference is calling the JavaScript functions, which use prefined API using our context, which uses the `IJSRuntime` service.

4.4 Templates

A Visual Studio (VS) template is a prepared project or collection of projects, that is adjusted by a programmer. This approach of creating new software is comfortable for the user because he or she does not have to start with an empty folder. We created templates for VS 2019 in order to make website creation easier. The templates are copied and adjusted from Peachpie templates [27] and comprise the first and the third example, presented in the following section, packed into a NuGet package. Usage and compilation of templates can be found in the attachments.

5. Examples

This chapter demonstrates the usage of `Peachpie.Blazor` by four examples, which are inspired by the use cases mentioned earlier. We describe example of structures, show important blocks of code, which have to be added to projects. Source code and binaries can be found in the attachments with build and run instructions. The library offers debugging logs, which can be helpful to get a better insight into the architecture mechanisms. Instructions on how to turn the logs on are shown in the second example.

5.1 WebGame

The example aims at the third use case. It contains a game where a rocket has to destroy falling asteroids with bullets, as we can see in Figure 5.1. The current Frames Per Second (FPS) is displayed in the left corner of the screen. The rocket can be controlled by buttons in the bottom, or we can use a keyboard, where arrows determine the rocket movement and the `F` key fires a bullet. We will discuss the rendering time in the benchmark section.

The example is implemented as a .NET solution consisting of three projects. `BlazorApp.Client` and `BlazorApp.Server` are generated projects by the Blazor App template. The game is implemented in the Peachpie `PHPScripts` project. These projects reference the `Peachpie.Blazor` library as a NuGet package containing all the necessary helper classes. The following paragraph consists of steps, which were necessary for making the game work.

The game implementation is divided into three PHP scripts and a CSS file defining game styles. The *settings.php* script comprises default values of delay between game refreshing, an asteroids frequency, and additional settings. The *asteroids.php* script defines game entities like the rocket or an asteroid. These entities utilize the `Peachpie.Blazor` library, which provides helper classes representing HTML elements. At the bottom of the script is the `Application` class connecting all parts together. The class uses HTML element events for interacting with a user due to the `PhpTreeBuilder` class providing an API targeting PHP usage. The last script is *main.php*, which contains the `AsteroidsComponent` class. A user can navigate this class by `/Asteroids` path. It initializes the game and uses `Timer` provided by `Peachpie.Blazor`, which enables to fire tick events updating the game and the screen. Because the `Application` class inherits the helper class `Tag`, `AsteroidsComponent` uses the `BlazorWritable` interface to

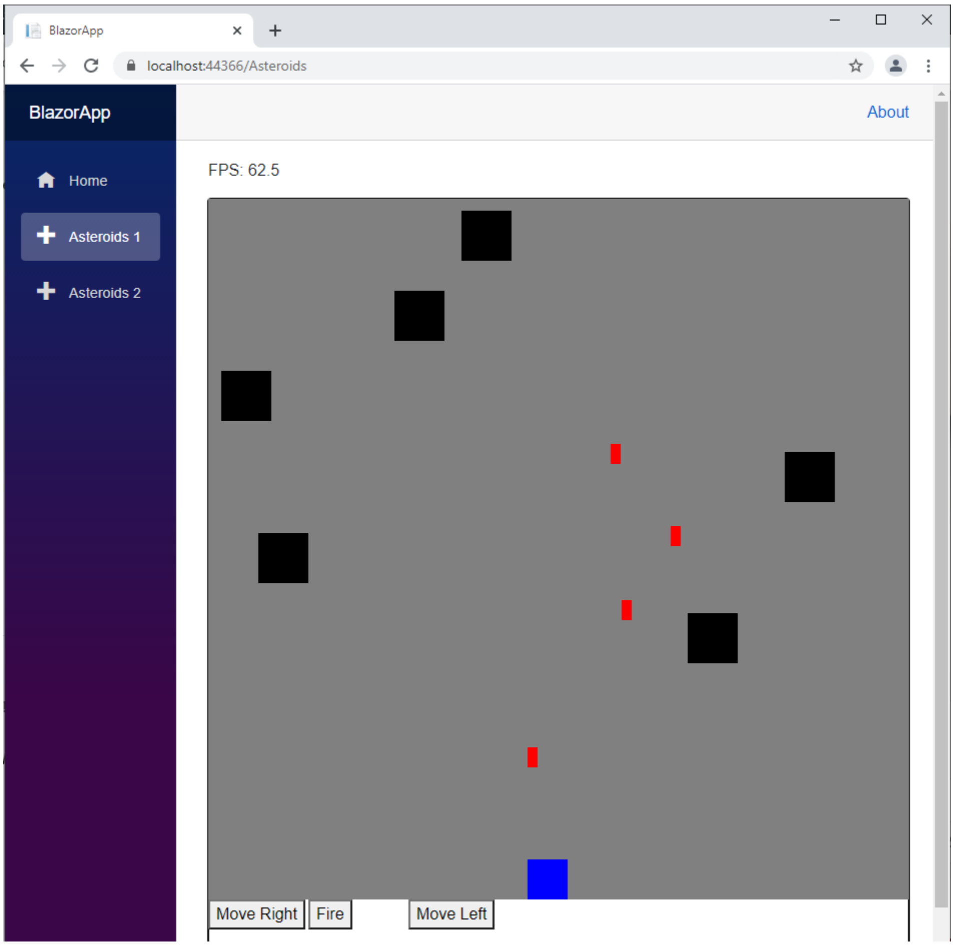

Figure 5.1: The game is written in PHP. We can use control buttons or a keyboard to let the rocket move and fire at the asteroids.

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
...
app.UseAdditionalWebStaticAssets(Configuration);
...
}
```

Figure 5.2: A part of the `Configure` method contained in the `StartUp` class, which is defined in *Startup.cs*

format and render the game into HTML instead of formating and rendering the game by exposing the `Application` structure.

`BlazorApp.Client` references the game and uses the default `Router` component to navigate `AsteroidsComponent`. The `index.html` contains links to the game styles and a supporting JavaScript defined in the `Peachpie.Blazor` library. We can see two examples of `AsteroidsComponent` usage in Figure 5.1. *Asteroids 1* utilizes the router for navigating it. *Asteroids 2* utilizes a Razor page, which contains additional content of the game.

`BlazorApp.Server` provides the Static Web Assets to a client by inserting additional middlewares, which handles their requests. This insertion can be seen in the *Startup.cs* file and in Figure 5.2, where we utilize the extension method for mapping resources defined in the configuration file, *appsettings.json*, into the pipeline.

5.2 Web

The *Web* example is inspired by the first use case, which moves the website to a client side. The website contains a simple layout consisting of references to its parts. We can see the default page in Figure 5.3. The website contains images, which are downloaded from the server when they are required.

The whole application is implemented as .NET solution consisting of three projects. `BlazorApp.Client` represents the client application containing `Program.cs`, which sets `WebAssemblyBuilder` and a root component, `PhpScriptProvider`, as we can see in Figure 5.4. Debug logs can be turned on by setting the minimum level of logging to Debug. `Blazor.Server` has the same role as in the previous example.

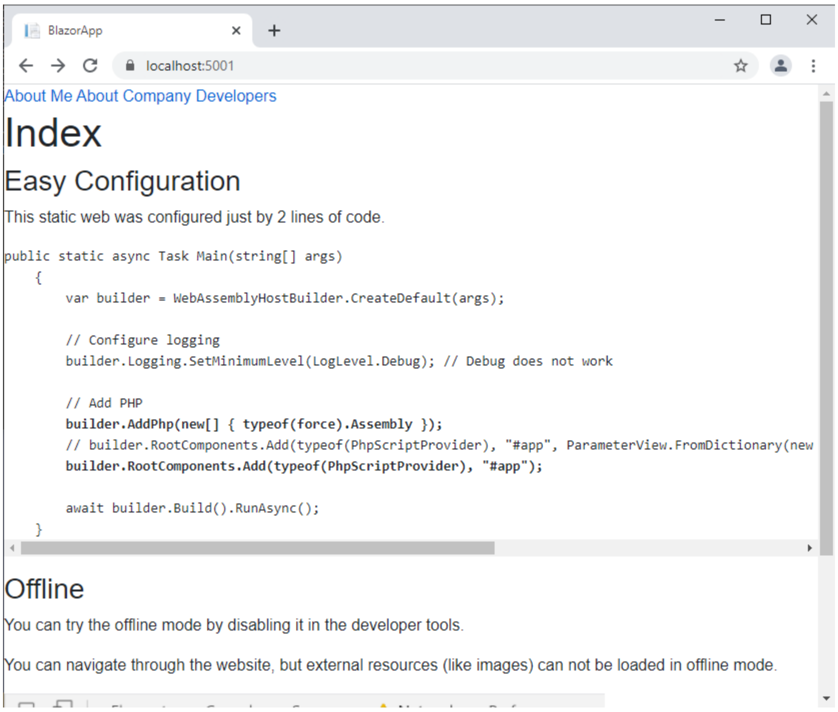

Figure 5.3: The default page of the website.

```
public static async Task Main(string[] args)
{
    var builder = WebAssemblyHostBuilder.CreateDefault(args);

    // Configure logging
    builder.Logging.SetMinimumLevel(LogLevel.Debug);

    // Add PHP
    builder.AddPhp(new[] { typeof(force).Assembly });
    builder.RootComponents.Add(typeof(PhpScriptProvider), "#app");

    await builder.Build().RunAsync();
}
```

Figure 5.4: The Main method in *Program.cs*, which can be found in the BlazorApp.Client project.

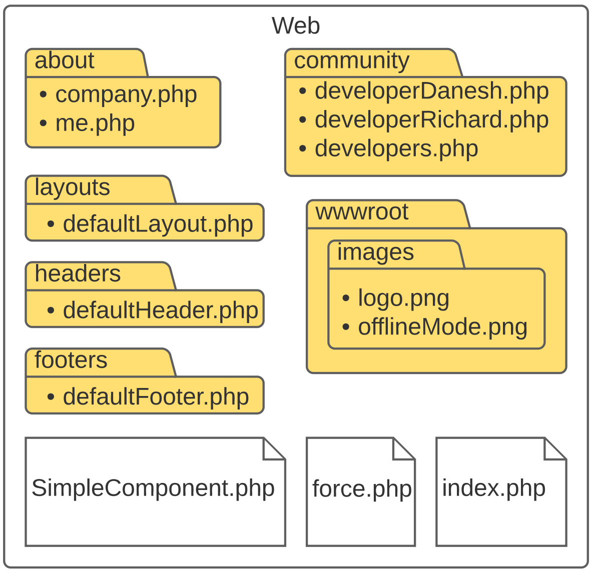

Figure 5.5: The file structure of the `PHPScripts` project.

The project references the `Peachpie.Blazor` library and a Peachpie project, `PHPScripts`, containing PHP scripts representing the website. The provider has default settings, which contains the `Router` type and the `OnNavigationChanged` mode of `BlazorContext`. Furthermore, we have to link the JavaScript script from the `Peachpie.Blazor` library to *index.html* in order to use it during the runtime.

The `PHPScripts` project contains the programmer's defined PHP scripts forming the web of a software company. We can see the project content in Figure 5.5. The project uses Peachpie SDK for compiling the scripts. The website has a simple layout defined in *defaultLayout.php* referencing pages about the founder, the company, and the community. The *me.php* page contains an image, *logo.png*, which is loaded by a common tag, `<img alt="Logo" src="Web/images/logo-.png"/>`. We can see the *force.php* script containing an empty `force` class, which is used in `BlazorApp.Client` to force loading of this assembly to a client.

An interesting page is *developers.php*, which displays information about developers working in the company. We can see the script in Figure 5.6 It uses script inclusion to add the head section. Then, there is a JavaScript call using our predefined API, which causes the alert with the message when the page loads. The whole page uses HTML interleaving. The `$_GET` superglobal is used to make a decision on which content to display based on the URL query. The default mode for the context is `OnNavigationChanged`. When a user refreshes the page after navigation to a developer, he or she can see the anchors to developers. It

```
<?php
    require("/headers/defaultHeader.php");
    CallJsVoid("window.alert", "Hello from PHP script.");
?>
<?php
if (isset($_GET["developer"])) {
    $name = $_GET["developer"];
    require("/community/developer$name.php");
} else {
?>
...
<p>Get more info about
<a href="/community/developers.php?developer=Richard">Richard</a>.
</p>
...
<?php } ?>
<?php
    require("/footers/defaultFooter.php");
?>
```

Figure 5.6: *developers.php* demonstrates using superglobals in the Blazor environment.

is caused by creating a new context between navigation, so the variables are disposed. If we want to change the mode to `Persistent`, we just set the component parameter, `ContextLifetime`, to `Persistent`. It is shown in the next example. This page is transparently rendered by our `PhpScriptProvider`, which evaluates the whole script and adds the output as a markup text to the builder.

5.3 OneScript

This example aims at the second use case. The website contains several pages, which demonstrate an insertion of page fragments, written in PHP, to the Blazor website. When we navigate the page, there is a button, which utilizes interoperability between PHP and JavaScript provided by `Peachpie.Blazor`. When we click on it, the JavaScript code calls PHP code, which writes a message to a browser console. The next examples referred as *Example 1*, *Example 2*, *Example*

3 show working with forms. The first example uses a simple form with the `GET` method. When we submit the form, we are navigated to a page written in PHP, displaying the content of superglobals. The same process is done with the `POST` method. We can also try to load a file to the form in the last example. After the submit, the page displays its file content encoded into *base64* encoding. *Example 4* uses previously mentioned features to enable displaying defined graphs by the user. We can upload a file containing the graph, or the application will generate it. Then the graph is displayed, and the points defining it can be downloaded, as we can see in Figure 5.7.

The .NET solution consists of three projects. `BlazorApp.Server` is the same as equivalents from the previous examples. `BlazorApp.Client` contains common Razor pages, and has default `Router` as a root component. We create several scripts in the Peachpie `PHPScripts` project to enrich the website with all the presented content except the layout provided by Blazor. The website contains three Razor pages: *Index.razor*, *PhpGateway.razor*, and *PhpScript.razor*. The first page uses `PhpScriptProvider` to navigate `index.php`. Using the provider is straightforward.

Figure 5.8 shows the call of PHP function from JavaScript. As we can see, it is effortless to call it. The `callPHP` function accepts a function name and an object to serialize. When the script is rendered, the context contains a defined `CallPHP` function. We click on the button invoking the `Call` method on the context, which invokes the desired function. Then, the parameter is deserialized. There is an interesting thing about using `echo`, `print`, etc., when the script is not rendered.

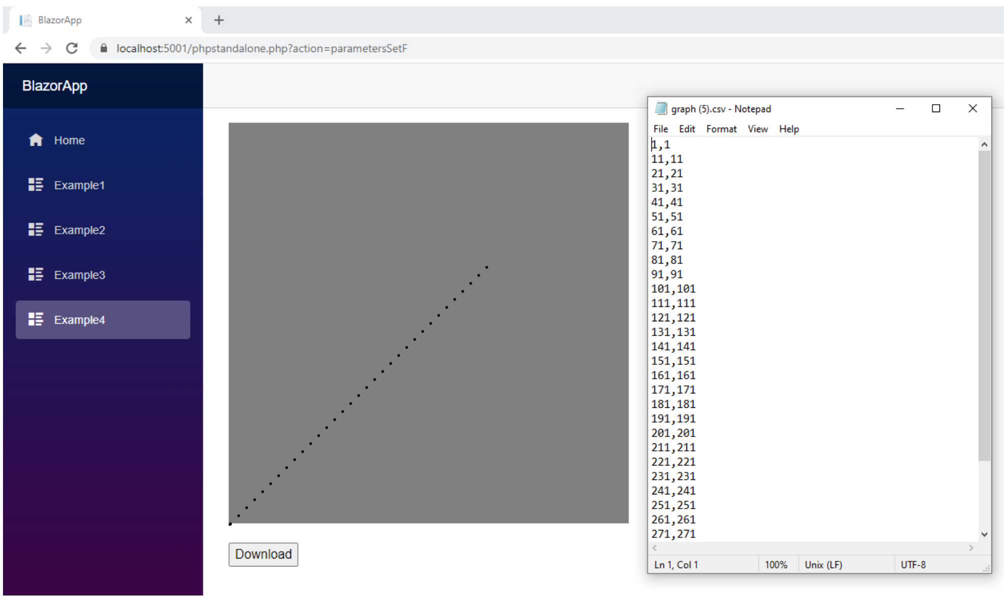

Figure 5.7: Application for visualising a graph.

```
...
<p>Click and look at console output</p>
<button onclick="window.php.callPHP('CallPHP',
    { name : 'Bon', surname: 'Jovi'});">PHP</button>
<?php

function CallPHP($data)
{
    $json = json_decode($data);

    echo "Hello " . $json->name . " ";
    echo $json->surname .  " from PHP\n";
}
```

Figure 5.8: *index.php* is a part of the `PHPScripts` project.

The context provides the second writer, which uses `Console` as the output. So the printing methods write messages into the web browser console.

Another part of the website uses forms to demonstrate `GET` and `POST` methods. We can see it in the *php* folder, where are three examples of forms using both methods and file loading. These examples can be navigated based on their names due to the unspecified URL of the Razor page, which uses the provider. After navigation to this page, the provider gets the script name from the URL.

A usual way of interaction with a user is an HTML form, but we want to show an advantage of the persistent context in the graph visualization. The forms are handled on a server side, but we can use them on a client side now due to `Peachpie.Blazor`. There is a simple application enabling us to visualize a graph, as we can see in the folder `fileManagement`. The application allows a user to upload a CSV file containing a graph or generate a new one based on the given parameters. We use a PHP library for parsing the file, which demonstrates a possibility to utilize the already created library on the client side. The application contains the main script `fileManagment/index.php`, which recognizes what to do based on superglobals and saved variables. It is possible because of context persistency.

5.4 AllTogether

This example aims at the fourth use case, where we want to connect PHP and C# to form one website. It uses the website made in the first example and includes it in the already existing Blazor website. It connects the game created earlier and the graph visualizer to show context sharing between `PhpScriptProvider` and `PhpComponent`. We can see the default page of the PHP website in Figure 5.9. The game can be navigated by an anchor, `Start`. When we play it, we can restart it with an anchor placed above the game. Afterward, we can see a graph, which contains a score generated by the game.

The .NET solution contains three projects `BlazorApp.Client`, `BlazorApp-.Server`, and `PHPScripts`. `PHPScripts` consists of *Game*, *Graph*, *Web*, and *wwwroot* folders. The implementation of these parts is explained in previous examples. An interesting collaboration is between the game and the graph, where a global variable containing the graph is saved in `AsteroidsComponent`, and it is used later in *main.php* providing the graph visualization. It works due to `PhpScriptProvider`, where we navigate either *main.php* or `AsteroidsCompo-nent`. Because the context can be persistent, global variables are shared between

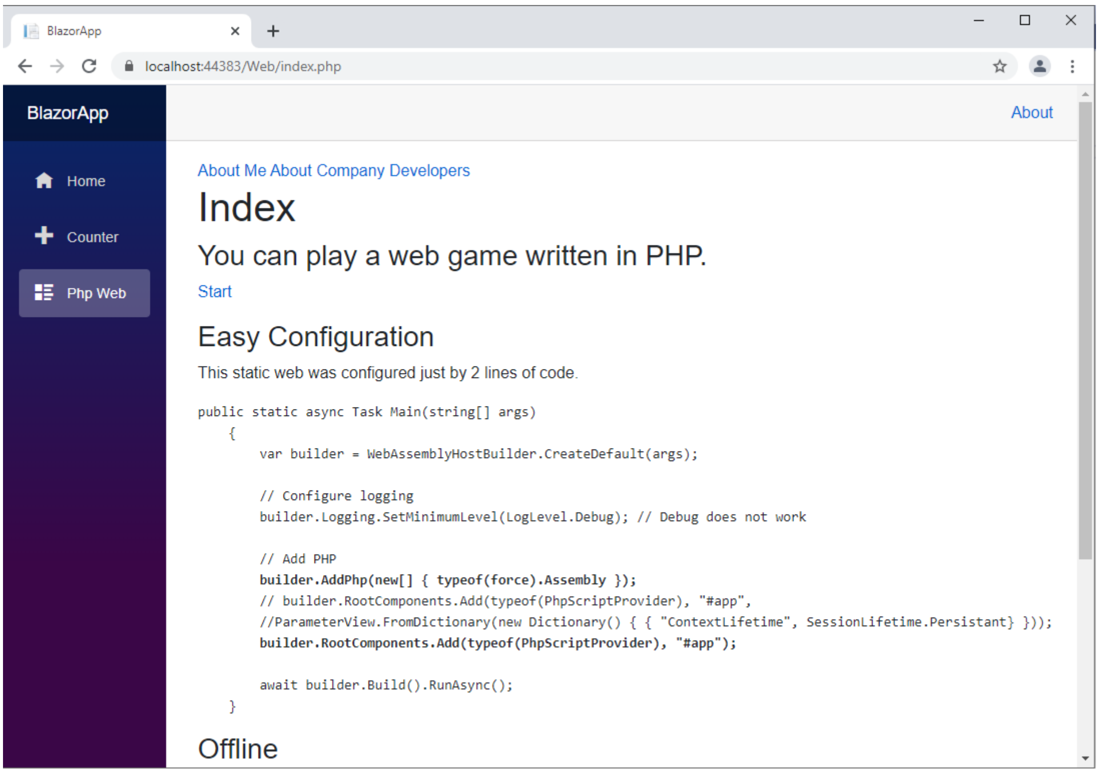

Figure 5.9: The example of interconnecting a PHP website with a Blazor website.

these parts, and they can communicate through them. We choose between navigated parts by submitting a form containing one button, which navigates the game, and an anchor containing a reference to *main.php*. Because the default `Router` can navigate `AsteroidsComponent`, we don't add the `PHPScripts` assembly to the router additional assemblies in order to let the provider navigate the component. The navigation, maintained by the provider, shares the provider content with the game component, which is a reason for keeping the provider alive and hiding the game component before the default router.

`BlazorApp.Client` contains the *Web.razor* page, where we insert an instance of `PhpScriptProvider` managing the PHP website navigation. There is the `Game.razor` page, which is shown in Figure 5.10. We use the advantage of defining more page paths because the default Blazor `Router` is a root component, it reacts to navigation, but we want to navigate the game and graph visualization by the provider. This can be done by defining two routable paths, which navigate the same Razor page, but the different content by our provider.

```
@page "/Graph/index.php"
@page "/Asteroids"
@using Peachpie.Blazor

<PhpScriptProvider ContextLifetime="@SessionLifetime.Persistant"
Type="@PhpScriptProviderType.ScriptProvider">
    <Navigating>
        <p>Navigating</p>
    </Navigating>
    <NotFound>
        <p>Not found</p>
    </NotFound>
</PhpScriptProvider>
```

Figure 5.10: *Game.razor* shares the context between the game and the graph visualizer.

6. Benchmarks

This chapter shows the difference between using `PhpTreeBuilder` properly and using only the `AddMarkupContent` method. Then it tests the speed of three PHP library functions in the Blazor and desktop environments. These benchmarks are available as .NET solutions in the attachment. The following sections introduce the background of the benchmarks, execute them and then evaluate the results.

The tests were executed on a HP Spectre x360 Convertible laptop with Intel(R) Core(TM) i7-8550U CPU, 8GB RAM, and NVIDIA GeForce MX150. They used Google Chrome (version: 90.0.4430.93) browser and Apache server.

6.1 Rendering

This benchmark observes the refreshing speed of a page using two ways for rendering a page content. The `Asteroids` game, mentioned earlier, is modified to explore FPS during rendering. The modified application generates asteroids, represented by `<div>` element with CSS styles, and lets them fall until they reach the bottom, as we can see in Figure 6.1. Then, they are removed. The current FPS, count of elements generated by the application, and time from starting the measurement are logged every second. Then, the results are evaluated.

The .NET solution comprises three projects: `BlazorApp.Client`, `BlazorApp.Server`, `PHPScripts`. The setting of BlazorApp is similar to the previous examples. `PHPScripts` contain the modified game together with additional scripts providing the interactive setting for the benchmark in a browser. When we start the server and navigate the website, we can set various properties of measurement like an upper bound of FPS, a frequency of asteroids, or dimensions of background. We explain only the *Rendering* property because the rest of the properties are self-explanatory. There are two options for the property. We call the first one `String` rendering because it uses `AddMarkupContent` method for updating the page content. This method accumulates the generated HTML entities into a single string and adds it into `PhpTreeBuilder`. We call the second one `Diff` rendering because it utilizes a specialized builder interface for adding HTML entities, which includes `AddAttribute`, for example.

We suppose that the first method is less effective than the second because the diff algorithm makes lesser update optimizations in the first case. It is caused by not providing the sequence numbers for each of the HTML entities, but it is regarded as the one part. This method simulates the rendering of a whole script

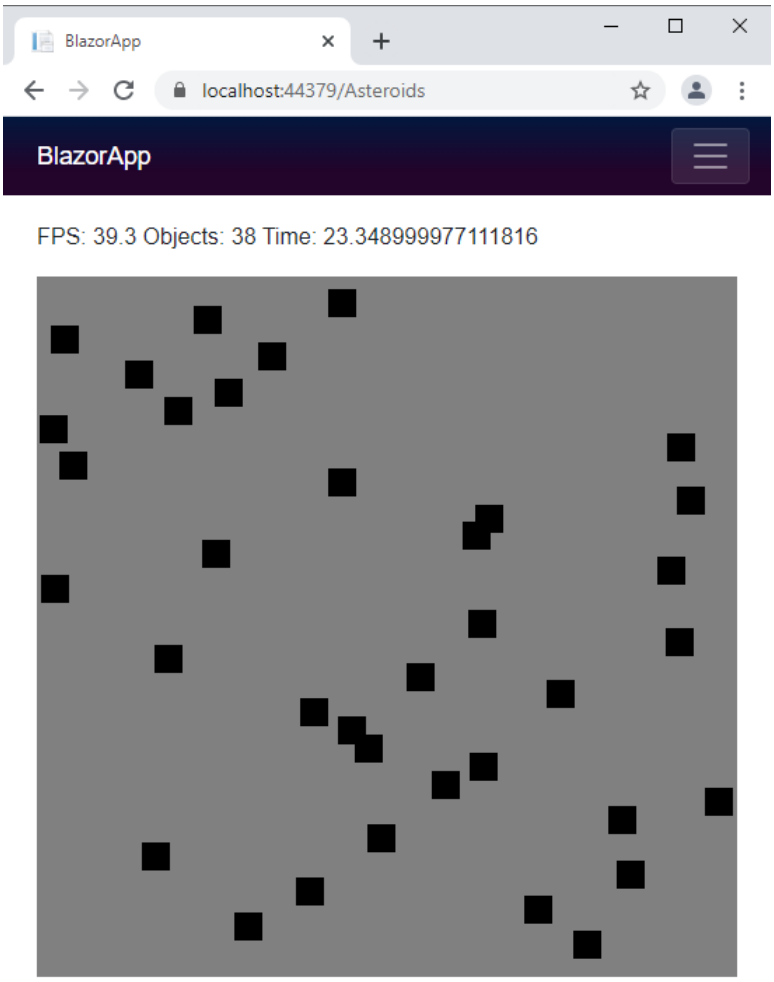

Figure 6.1: The benchmark observing refreshing speed of a page.

by `PhpScriptProvider`, which obtains the script output as a string and passes it to the `AddMarkupContent` method. This part is regenerated every time instead of update only the changed parts. We measured FPS 10 times for each configuration and then made a mean of them. Data of these measurements can be found in the attachment.

We can see results from the first measurement in Figure 6.2, which uses one tag element representing an asteroid. The timer is set to refresh the window in one-thousandth of a second after building the rendering tree. The resulting speed will be significantly lower, but we want to measure the maximum rendering speed of Blazor, which can be done by almost immediately waiting for the next rendering. We generate five asteroids per second. The dimensions of the background are 800 x 800. The FPS and number of objects are measured for 60 seconds. We can see a graph displaying the current FPS and the number of objects. The lines represent measured values for each type of rendering. In spite of mean, the lines are not smooth. It is caused by a difference of a sampling rate, where the measurements

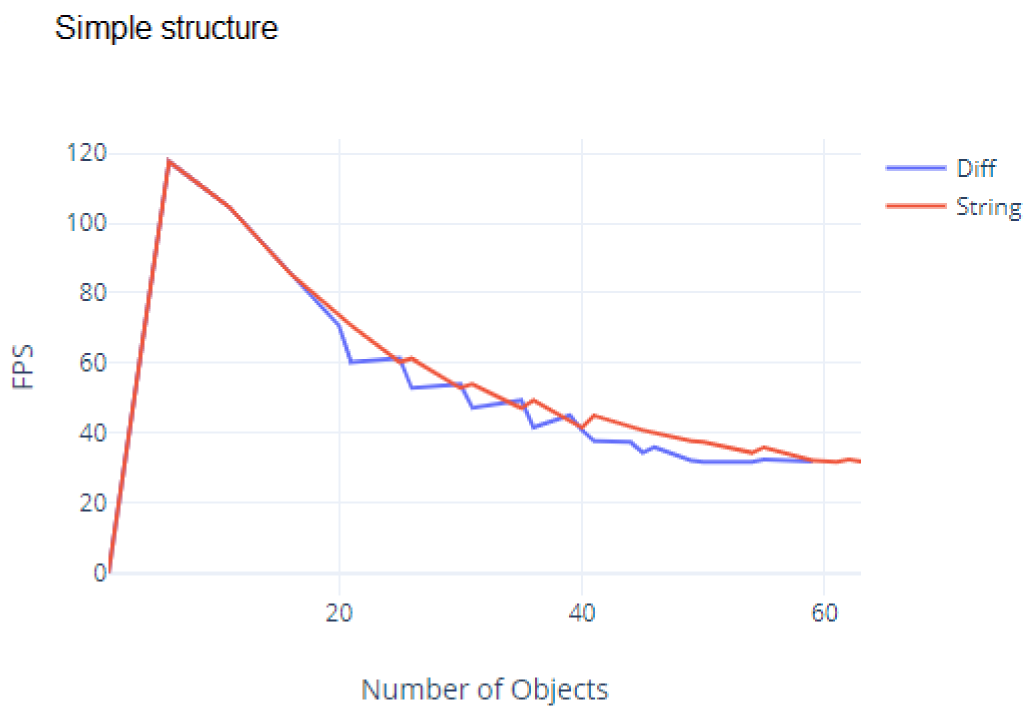

Figure 6.2: The graph represents measured FPS for the current number of objects, where we use only a single tag for an asteroid.

do not log FPS in the common numbers of objects. However we think that the graphs sufficiently prove our hypothesis. The second measurement aims at the deeper structure of updating elements in Figure 6.3. The setting is the same as the previous one.

We can see that the number of objects was changing during the application runtime. It is because of the beginning state of the application, where there are no asteroids. When the first asteroids fell to the bottom, the number was stable. We can see that when the updating objects had not inner elements, the rendering ways were the same at the end of the graph. When the updated objects had inner elements, the FPS was lower in the first way of rendering, as we can see in Figure 6.3. This is caused by optimized updates made by the diff algorithm, which generates only DOM updates for the element style and leaves the inner tree unchanged. As we can see, the FPS was about 30 in a second way when the number of asteroids was stable, which is fine because the untrained eye is able to distinguish up to 23 images per second. The first way had about 24 FPS, which looked glitchy in the browser. This benchmark demonstrates when it is a good choice to use the builder properly and the need to use `PhpComponent` for render-demanding applications.

Another problem caused by the first way of rendering is handling click events

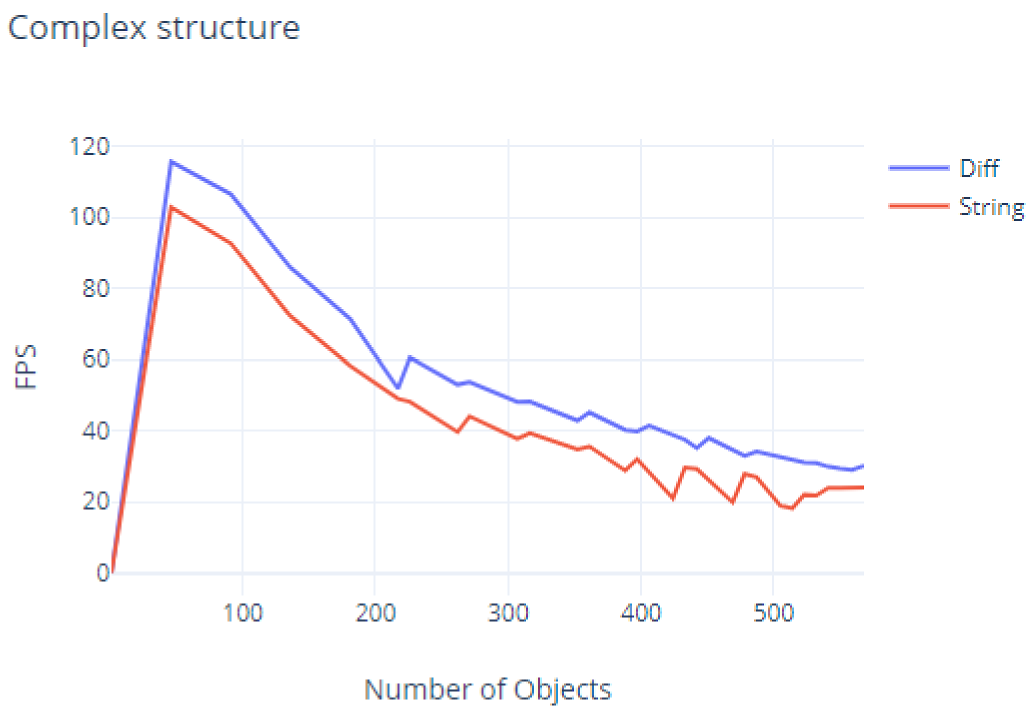

Figure 6.3: The graph represents measured FPS for the current number of objects, where we use the complex structure of HTML tags for an asteroid.

when the window is being quickly refreshed. Because the click events consist of two inner events (mouse up, mouse down), the event is not fired due to removing the element before the second event appears.

6.2 Performance of libraries

This benchmark wants to highlight the performance of PHP libraries in the Blazor environment. We prepared a simple script, which measures the execution time of the following three functions: `imagecreatetruecolor`, `str_getcsv`, `md5`. The first function creates a 1920x1080 blank image. The second function converts a short string representing CSV data to an array. The last function computes a hash of the CSV string. We measure elapsed time during the function execution multiple times and then make a mean. This approach should make the time spend with the JIT compilation less important. We compare the results of executing the script in a desktop environment, where we use Peachpie, the Blazor environment using Peachpie as well, and the native PHP interpreter.

The benchmark is created as a .NET solution consisting of four projects: `BlazorApp.Client`, `BlazorApp.Server`, `Desktop`, `PHPScripts`. The `PHPScripts` project contains *index.php*, which does the benchmark. The `Desktop`

Environment	imagecreatetruecolor	str_getcsv	md5
Blazor - Peachpie	23×10^6	60	Error
Desktop - Peachpie	2×10^4	1.9	1.8
Desktop - Native	5×10^2	7.2	0.4

Table 6.1: Elapsed time (microseconds) of the function executions.

project runs the script in a console application. `BlazorApp` projects run the script in a browser.

We can see the result of the measurement in Table 6.1. The first observation is that the call of the `md5` function caused an error: *This function is unsupported in this platform*. The second observation is an extremely slow execution of the `imagecreatetruecolor` function in the Blazor environment. We suppose that it is caused by the `ImageSharp` library, which has not to be optimized in the browser environment. The difference between the execution times with the PHP interpreter and the console application is caused by using the different libraries. Peachpie uses libraries, which are written in C#. The PHP interpreter uses different libraries written in C. Based on the results we can not decide whether the Peachpie or the PHP interpreter is faster.

This benchmark reveals the need to test a library before it is being used in the Blazor environment. We suppose that these issues with speed and the error will be solved because Peachpie uses a standard library for encryption and `ImageSharp` for the graphics, which has a big community. On the other hand, if the execution time will be slower than the execution with the PHP interpreter, it would consume a client resource, which is a server benefit.

7. Conclusions

We introduced the PHP language and its conventions as a server-side scripting language. We continued with the JavaScript language and a new supported standard, WebAssembly. We observed the basics and limitations of Peachpie and Blazor.

Then, we considered possible ways of combining the PHP script compilation with the Blazor environment to provide executing these scripts on a client side. We proposed four use cases aiming at different end-users to cover the goals of this thesis. The first use case aims at a pure PHP web application written in PHP, which is moved to a client side by using the library components navigating the PHP scripts. The second use case considers PHP scripts a part of a Blazor website, which can be inserted into Razor pages. The third use case implements a web game, which utilizes the Blazor logic to improve rendering performance. The last use case joins the previous use cases and interconnects them by sharing PHP context between them to preserve an application state. The proposed solution was designed and implemented to solve these use cases.

It results in the `Peachpie.Blazor` library, which offers API for inserting PHP scripts to Razor pages of the Blazor website, navigation to PHP scripts, helper classes representing HTML entities to make the rendering with Blazor easier, a form handling, and sharing PHP context between the navigation. Different approaches of inserting PHP scripts into Blazor utilize different levels of abstraction to cover all possible users with different knowledge of Blazor and Peachpie. We are able to work with Blazor structures directly or render the content of PHP scripts transparently due to our `Peachpie.Blazor` library. These features help PHP programers to use Blazor and move the script executions to a client side and help PHP and C# programmers to co-work on a joint website.

We demonstrated the use cases by implementing four examples, which present different levels of difficulty utilizing the integration. One more project is part of the library. It consists of two templates and is created for Visual Studio environment. These templates help a programmer to get started with development.

In the end, we used two benchmarks to reveal issues caused by the used technologies and approaches of using structures for rendering. We learned that the proper usage of `RenderTreeBuilder` has the purpose, and C# libraries have to be tested before we will use them in Blazor because they have not not be optimized or even supported in the Blazor environment.

7.1 Future work

The thesis explored the possibilities of integration. A new goal of the work is to spread it to people and get feedback from them, which determines the next improvements of this approach. We contacted the Peachpie team, which started to solve the interoperability issues with C#. Thus, we will be able o adjust the `Peachpie.Blazor` in compliance with given solved issues. The library will be available as a NuGet package on `www.nuget.org`. It will have a public repository on GitHub for potentially interested people.

Bibliography

[1] WebAssembly. https://en.wikipedia.org/wiki/WebAssembly. [Online; accessed 2021-03-23].

[2] The project PHP in a browser. https://github.com/oraoto/pib?fbclid=IwAR3KZKXWCC3tlgQf886PF3GT_Hc8pmfCMI1-43gdQEdE5wYgpv07ObRwXqI. [Online; accessed 2021-03-17].

[3] Blazor homepage. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor. [Online; accessed 2021-03-17].

[4] Peachpie homepage. https://docs.peachpie.io. [Online; accessed 2021-03-24].

[5] PHP. https://en.wikipedia.org/wiki/PHP. [Online; accessed 2021-03-24].

[6] PHP manual. https://www.php.net/manual/en/. [Online; accessed 2021-03-24].

[7] Javascript. https://en.wikipedia.org/wiki/JavaScript. [Online; accessed 2021-03-24].

[8] WebAPI introduction. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction. [Online; accessed 2021-03-24].

[9] Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. [Online; accessed 2021-03-17].

[10] WebAssembly concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. [Online; accessed 2021-03-17].

[11] Threads in WebAssembly. https://developers.google.com/web/updates/2018/10/wasm-threads. [Online; accessed 2021-03-17].

[12] Common Language Infrastructure. https://en.wikipedia.org/wiki/Common_Language_Infrastructure. [Online; accessed 2021-03-30].

[13] .NET Core. https://en.wikipedia.org/wiki/.NET_Core. [Online; accessed 2021-03-26].

[14] Mono compilation into WebAssembly. https://www.mono-project.com/news/2017/08/09/hello-webassembly/. [Online; accessed 2021-03-18].

[15] Hosting models. https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0. [Online; accessed 2021-03-18].

[16] Component lifecycle. https://docs.microsoft.com/en-us/aspnet/core/blazor/components/lifecycle?view=aspnetcore-5.0. [Online; accessed 2021-03-18].

[17] Blazor, Razor, WebAssembly, and Mono. https://daveaglick.com/posts/blazor-razor-webassembly-and-mono. [Online; accessed 2021-03-18].

[18] Routing in Blazor. https://chrissainty.com/an-in-depth-look-at-routing-in-blazor/. [Online; accessed 2021-03-18].

[19] Diff algorithm. https://gist.github.com/SteveSandersonMS/ec232992c2446ab9a0059dd0fbc5d0c3. [Online; accessed 2021-03-30].

[20] Render tree. https://www.infoq.com/articles/blazor-rendertree-explained/. [Online; accessed 2021-03-30].

[21] Razor Class Libary with static assets. https://docs.microsoft.com/en-us/aspnet/core/razor-pages/ui-class?view=aspnetcore-3.0&tabs=visual-studio#create-an-rcl-with-static-assets. [Online; accessed 2021-05-1].

[22] User persona. https://xd.adobe.com/ideas/process/user-research/putting-personas-to-work-in-ux-design/?fbclid=IwAR0IcZTn2G80ilY4C2juxMJZAWG9B3ipQmYKqoUPVb9uA840J9sEs65ACYw. [Online; accessed 2021-03-30].

[23] Custom router. https://github.com/chrissainty/BuildingACustomRouterForBlazor. [Online; accessed 2021-03-30].

[24] Javascript from .NET. https://docs.microsoft.com/en-us/aspnet/core/blazor/call-javascript-from-dotnet?view=aspnetcore-5.0. [Online; accessed 2021-03-30].

[25] .NET from Javascript. `https://docs.microsoft.com/en-us/aspnet/core/blazor/call-dotnet-from-javascript?view=aspnetcore-5.0`. [Online; accessed 2021-03-30].

[26] Query strings. `https://chrissainty.com/working-with-query-strings-in-blazor/`. [Online; accessed 2021-03-30].

[27] Peachpie templates. `https://github.com/peachpiecompiler/peachpie-templates`. [Online; accessed 2021-05-17].

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

SDK Software Development Kit

FPS Frames Per Second

RCL Razor Class Library

VS Visual Studio

CLI Common Language Infrastructure

CIL Common Intermediate Language

CLR Common Language Runtime

A. Attachments

Folder	Description
`bin/`	Executable files of examples
`src/Peachpie.Blazor/`	Source code of library implementing the proposed solution
`src/examples/`	Source code of mentioned examples
`src/benchmarks/`	Source code of mentioned benchmarks
`src/templates/`	Source code of mentioned templates
`data_benchmarks/`	Measured data using the benchmarks
`docs/`	Autogenerated technical documentation of the library
`nugets/`	The library and templates packed into nugets.

The instructions for getting started and a short description of the library API can be seen in the last appendices.

A.1 Build and run

The library, examples, and benchmarks are .NET solutions, which can be opened by VS 2019. The `bin` folder consists of compiled examples with supports for various OS and architectures like Windows (x86, x64), Unix, OSX, or Linux (x86, x64, arm). These binaries were compiled on Windows 10, but the packages do not utilize any dependent platform specification. The ASP.NET Core Runtime 5.0, which is available from `https://dotnet.microsoft.com/download/dotnet/5.0`, is required to run binaries. After the installation, the particular example can be run by moving to the `bin/NameOfExample/` directory and run the command line below:

```
dotnet BlazorApp.Server.dll
```

Then, the website can be accesed by navigation to `https://localhost:5001`, or `http://localhost:5000`.

The .NET 5.0 SDK, which is available from the same website as the runtime, is required to build .NET solutions. After the installation, the `Peachpie.Blazor` library has to be compiled and packed. This is done by moving to the `src/Peachpie.Blazor/Peachpie.Blazor` directory and run the commands below:

```
dotnet build --configuration Release
dotnet pack --configuration Release
```

Then, an example can be compiled by moving to its directory containing the `BlazorApp.Server` project and run the command below using the NuGet generated by the previous step:

```
dotnet build --configuration Release --source PathToTheLibraryNuGet
```

After the compilation, the server can be launched by the command below:

```
dotnet run --no-build --configuration Release
```

`dotnet publish` can be used for packing all application parts. But static resources of Peachpie projects have to be copied in the publish folder and the *appsettings.json* has to be change in order to the server finds the static resources.

The template compilation can be done by running a PowerShell command `./build/build.ps1` in the template folder. It results in a NuGet package, which can be added to VS template collection. The next appendix gives instructions for adding the templates to VS.

B. Instructions for getting started

The library contains Blazor components and helper classes enabling navigation and execution of PHP scripts, defined in a Peachpie project, inside a browser. It is available as a NuGet package, which can be found in the attachment. There are two ways how to get started. We created templates, which can be adjusted to fit your goal. The second way creates a new Blazor project referencing PHP scripts. We recommend beginning with the first (easier) way.

Prerequisities

- .NET 5.0 SDK - `https://dotnet.microsoft.com/download/dotnet/5.0`
- Visual Studio 2019 (Optional) - `https://visualstudio.microsoft.com/vs/`
- *Peachpie.Blazor.1.0.0-alpha.nupkg* - A part of the attachment
- *Peachpie.Blazor.Templates.1.0.2.nupkg* - A part of the attachment (Required by the first approach of creating a website)

From templates

Step 1 - Install the templates by `dotnet new -i /path/to/template/package`

Step 2 - Add a new source, where *Peachpie.Blazor.1.0.0-alpha.nupkg* can be find by the package manager, to the *NuGet.Config* file. It is usually located in *%appdata%/NuGet/NuGet.Config/* on Windows or *~/.nuget/NuGet/NuGet.Config* on Mac.

Step 3 - Choose a template depending on your intention:

Peachpie Blazor Web - A simple PHP website running in browser

Peachpie Blazor Hybrid - A simple Blazor website combining PHP and Razor Pages

Step 4 - Create the project by `dotnet new blazor-hybrid` or `dotnet new blazor-web`.

Step 5 - Modify the `BlazorApp.Client` or `PHPScripts` (Optional).

Step 6 - Navigate to *BlazorApp/Server/* and run `dotnet run`.

Step 7 - Access `https://localhost:5001`.

From Blazor project

Step 1 - Create Blazor WebAssembly project with ASP.NET Hosting and targeting .NET 5.0.

Step 2 - Install Peachpie Visual Studio extension (`https://www.peachpie.io/getstarted`).

Step 3 - Create a Peachpie Class library project.

Step 4 - Add a new source, where *Peachpie.Blazor.1.0.0-alpha.nupkg* can be find by the package manager.

Step 5 - Add references to the library in `BlazorApp.Client`, `BlazorApp.Server` and Peachpie projects.

Step 6 - Add `<script src="_content/Peachpie.Blazor/php.js"></script>` to the head of *index.html* located in *BlazorApp.Client/wwwroot/* folder.

Step 7 - Add assemblies containing PHP script to `WebAssemblyHostBuilder` (located in *BlazorApp.Client/Pragram.cs*) using the `AddPhp` extension method.

Step 8 - Modify projects (optional).

Step 9 - Launch the server by `dotnet run` in the *BlazorApp/Server/* folder.

Step 10 - Access `https://localhost:5001`.

C. API Short Description

This section picks useful classes and methods up, which are used to provide PHP scripts defined in the Peachpie project. See also generated library documentation for more detailed information about function parameters and return types. The library provides two approaches how to use PHP in Blazor. The first one creates a Blazor component by inheriting the `PhpComponent` class. The second approach inserts the `PhpScriptProvider` component into a Razor page or sets it up as a root component. This component navigates the PHP scripts, evaluates them, and generates page content based on their output. There are also helper classes representing HTML entities, which are useful in the first approach, where the builder has a complex API for generating page content. There are functions handling file management and interoperability between PHP and JavaScript.

C.1 PhpComponent

This class can be inherited in a script to provide Blazor component API in PHP. Every output initiated by echo or similar functions is copied into a browser console. The virtual method of the Blazor component can be freely overridden. The inherited class has to implement the `BuildRenderTree` method. It generates the page content using `PhpTreeBuilder`, which is a wrapper of the original `RenderTreeBuilder` and provides adjusted API for PHP use. We can see an example in Figure C.1

```
#[\Microsoft\AspNetCore\Components\RouteAttribute("/Asteroids")]
class AsteroidsComponent extends \Peachpie\Blazor\PhpComponent
{
    ...
  public function BuildRenderTree($builder) : void { ... }

  public function OnInitialized() : void { ... }
}
```

Figure C.1: `PhpComponent` inheritence

C.1.1 Helper classes representing HTML

The library provides a collection of helper classes representing HTML entities because `RenderTreeBuilder`, used in Blazor, is complicated for writing an HTML page. We can see an example of usage in Figure C.2.

```
$button= new \Peachpie\Blazor\Tag("button");
$button->attributes["style"]["position"] = "absolute";
$button->attributes["style"]["top"] = "700px";
$button->attributes["style"]["left"] = "100px";
$button->attributes["class"]->add("mybutton");

$button->attributes->addEvent("onclick", function($seq, $builder)
{ $builder->AddEventMouseCallback($seq, "onclick", function($e)
{ $this->HandleFire();})
;});

$button->content[] = new \Peachpie\Blazor\Text("Fire");
```

Figure C.2: Using helpers

- **BlazorWritable** - Interface defines `writeWithTreeBuilder` generating an HTML output.
- **Tag** - Class represents an HTML tag comprising its name, attributes and inner content. It implements `BlazorWritable`.
- **AttributeCollection** - Helper class manages tag attributes in array style. It is used by the `Tag` class.
- **ClassBuilder** - Helper class adds and removes class names from the *class* attribute. It is used by the `AttributeCollection` class.
- **CssBuilder** - Helper class builds css style format used in the *style* attribute. It is used by the `AttributeCollection` class.
- **Text** - Class represents an HTML text element. It implements `BlazorWritable`.

C.2 PhpScriptProvider

The component navigates PHP scripts in Peachpie project. It has three modes:

- **Router** - Handles navigation and script routing (Used when it is set as a root component).
- **ScriptProvider** - Provides script routing based on URL (Can be used with the default Blazor router).
- **Script** - Always navigates one script defined by its parameter.

The component can preserve the Peachpie context to next evaluation. This behaviour is determined by the `ContextLifetime` parameter having two modes.

- **OnNavigationChanged** - Every navigation causes a new context, which is a standard behaviour of PHP.
- **Preserved** - The context is preserved until the component is not disposed.

We can see an example of usage in Figure C.3. It is a Razor page containing the provider.

```
@page "/php/{*sth}"
@using Peachpie.Blazor

<PhpScriptProvider
ContextLifetime="@SessionLifetime.OnNavigationChanged"
Type="@PhpScriptProviderType.ScriptProvider">
    <Navigating>
        <p>Navigating to script</p>
    </Navigating>
    <NotFound>
        <p>Script not found</p>
    </NotFound>
</PhpScriptProvider>

@code
{
    [Parameter]
    public string sth { get; set; }
}
```

Figure C.3: Using `PhpComponent`

C.2.1 BlazorContext

The context inherits the original PHP context and adds functionality for managing Blazor. The functionality includes handling `$_GET`, `$_POST`, and `$_FILES` superglobals. It includes evaluating HTML forms on a client side, where forms are not sent to a server but it is parsed and handled by PHP scripts inside a browser. It also provides file management as reading uploaded files by forms or creating a new one, which can be downloaded back to a client. For this purpose, there are the following functions and structure:

- **BrowserFile** - Represents a file, which can be downloaded to a client.
- **DownloadFile** - Downloads a file stored in memory with a specified ID.
- **CreateFile** - Creates a new file with given content, name, and type.
- **GetBrowserFileContent** - Reads a content of a specified file.
- **CreateUrlObject** - Creates an object, which can be navigated by returned URL.

C.3 Helper classes

- **Timer** - Wrapper of the .NET `Timer` class. It enables to handle the `Elapsed` event.
- **addSimpleEventListenner** - Function for adding a simple event listener (does not have any parameters) to C# objects in PHP.
- **addEventListenner** - Function for adding an event listener (has parameters) to C# objects in PHP.

C.4 Interoperability

- **window.php.callPhp** - Calls a PHP function by given name and parameters passed as JSON. The context has to be presented in JavaScript, meaning that the page has to contain one of the components mentioned earlier, and the context has to already know the function (requires at least one script execution).
- **CallJSVoid** - Calls a JavaScript void function.

- **CallJS** - Calls a JavaScript function.
- **CallJSVoidAsync** - Calls a JavaScript void function asynchronously.
- **CallJSAsync** - Calls a JavaScript function asynchronously.