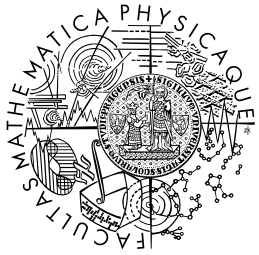

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Husák

Client-side execution of PHP applications compiled to .NET

Department of Software Engineering

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

Study programme: Computer Science (B1801)

Study branch: ISDI (1801R049)

Prague 2021

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

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

In ..... date ....

Author's signature

Dedication.

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

Author: Tomáš Husák

Department: Department of Software Engineering

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

Abstract:

Write an abstract.

Keywords: PHP .NET Blazor Peachpie

Contents

Introduction

We can divide web applications into two types by roles of server and client. However, they use different technologies for their purpose. We will start with common parts. An internet protocol, HTTP, usually carries out the communication between a server and a client. A client uses a web browser for requesting a server. A markup language HTML is essential for describing a page's structure. A browser is responsible for interpreting and rendering the page's content. It is a web application's environment for further interaction. The need to adjust content by different styles initiates standardizing CSS language, which enriches pages with a wide graphical content.

The first type is server-based web applications. A server prepares the page, makes additional computations related to the request, and sends it back to the client. Having a business logic on a server-side is the main objective. The most popular language for server-side scripting becomes PHP.

Add a statistic for the popularity of PHP

The second type is client-based web applications, where major business logic is moved to a client's browser. However, the combination of CSS and HTML is sometimes sufficient. This type of application needs dedicated technologies, which allow manipulation with a page structure, reacts to on-page events, and controls the browser's behavior. Many languages were enabling the manipulation, but they were not usually supported by most browsers like Google Chrome, Safari, Opera, and Mozzila. The scripting language Javascript became a browser standard from these supports.

However, Javascript is a powerful language. There are language-specific features, which are harder replaceable by the language. Despite the urge, 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. There appeared a portable binary-code format for executing programs, WebAssembly (abbreviated WASM) [1] in 2015. WebAssembly aims to secure high-performance applications on web pages. Interop with Javascript makes the format as powerful as the language. The advantage of WebAssembly is a being compilation target for many programming languages. Since December 2019, when the W3 Consortium has begun recommending WebAssembly, it is easy to migrate other languages to the browsers supporting this recommendation.

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

Add a statistic for popularity of .NET

The .NET and PHP popularity led to the creation of the Peachpie compiler. Peachpie [4] tries to make use of the .NET platform and offers PHP compilation to .NET. It is a modern compiler enabling interop between PHP and C#.

The project opens the PHP door to Blazor. An integration between Peachie and Blazor can yield to following benefits. A community of PHP developers

is significant. Thus, many PHP libraries apply to working with client's data, cooperation with databases, and other server tasks. The possibility to migrate the language PHP together with its conventions to a browser will impact developing dynamic web applications due to the PHP community and the libraries. It can join PHP and C# developers to collaborate with their programming languages using a minimum knowledge of the integration. Another interesting functionality of this idea is a full C#, PHP, and JavaScript interop which offers more options for developers and future extensions.

This thesis uses the compilation of PHP scripts to .NET in order to execute PHP in a browser powered by Blazor. The approach tries to achieve two goals. The first goal is to enable web development on a client-side with PHP. There are not libraries supporting this integration. The second goal is to design the support to offer a convenient way to combine a PHP code with a Blazor.

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

1. Existing technologies

In the beginning, we will observe conventions in web applications, and we will map ways of user interaction with the applications to utilize it later in the proposed solution. We will explore a server-side web application using PHP language and a client-side web application using Javascript language in other to obtain these observations. Afterward, it will present a WebAssembly, which will be followed by the existing project relating to run PHP in a browser. It will give short information about the .NET platform and C# language. In the end, it will introduce Blazor and Peachpie, which will be integrated.

1.1 PHP and server-based web application

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. An HTTP protocol uses a dedicated format of messages for communication. The protocol does not require any authentication at all. Statelessness is a typical characteristic of the protocol. A server has to retain information about clients and add additional information to the messages in order to distinguish between the clients.

Since the server contains all business logic, a browser has to send necessary data for required actions by an HTTP message. The data are usually encoded as a part of an URL address or in the HTTP message body. HTML presents a tag Form that makes sending the data easier for a client. Listing 1.1 contains an example of a Form tag. The form can contain other tags, which are displayed as various types of fields. A client fills these fields, and the browser sends the data as a new HTTP request to the server. We can specify how the data will be encoded. Get method is one of the basic ways. It encodes the data as a pair of keys and its values to the query part of the URL. There is a example of an URL `http://www.example.com/index.php?par1=hello&par2=2&arr=hello&arr=2`. A query part begins with a question mark. Parameters of the query part are described in the table 1.1. Another method is called post, which encodes it in the request body,

Key	**Value**
par1	*hello*
par2	2
arr	[*hello*, 2]

Table 1.1: Parameters of the query part.

which does not appear in the URL.

PHP [5] was designed for user page templating on the server-side. It has been adjusted gradually to enable writing application logic. PHP is an interpreted language maintained by The PHP Group. We will describe the language by using listing 1.1 as an example.

Listing 1.1: A PHP code.

```
1  <?php
2    inlude("header.php");
3  ?>
4
5  <h1>Superglobal POST</h1>
6  <?php
7    foreach($_POST as $key => $value) { ?>
8    <p><?php echo $key; ?> => <?php echo $value; ?></p>
9  <?php } ?>
10
11 <h1>File content:</h1>
12 <p>
13 <?php
14   if($_FILES["file"])
15   {
16     echo file_get_contents($_FILES["file"]["tmp_name"]);
17   }
18 ?>
19 </p>
20
21 <form action="/index.php" method="post">
22  <label for="name">Name:</label>
23  <input type="text" id="name" name="name"><br>
24  <label for="file">File:</label>
25  <input type="file" id="file" name="file"><br>
26  <input type="submit" value="Submit">
27 </form>
28
29 <?php
30   inlude("footer.php");
31 ?>
```

An HTML interleaving has appeared to be a helpful method for data binding. The feature allows inserting a PHP code between HTML. These fragments do not have to form individual independent blocks of code closed in curly brackets. The interleaving is related to code execution when an interpreter executes a script from top to bottom. Everything outside a PHP section is copied into the body of the request.

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

PHP has superglobals [6], which are built-in variables accessible from all scopes of the script. Following superglobals are relevant to the thesis. The $_GET variable stores parsed query part of the URL. The $_POST variable stores variables which are sent by post method. The $_FILES variable contains uploaded files.

Maybe add information about SESSION.

We can divide code in several ways. Global functions are the most notable characteristic of PHP despite wide-spread object-oriented programming. They

are defined in the global scope and accessible from anywhere. The next option is an object inspired by object-oriented programming. There is also namespace support. We can include a PHP code from other scripts. The execution continues after the inclusion, which is a common execution of the defined script.

The nature of the request-response semantic usually results in a one-way pass of the application. After dealing with a request, the whole application state is terminated. 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 listing 1.1. In the beginning, we delegate header rendering to header.php script. Then we render the body and include footer.php, which cares about the proper ending of the HTML page.

Uploaded files sent by the client reside in two places. The file's information is stored in $_FILES. 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.

1.2 Javascript and client-based web application

Since a client-based web application aims to do a major business logic in a browser, it needs to control the rendered page and access a web interface providing additional services. The interface is usually available from Javascript. We will start with a description of loading Javascript in a browser. We will introduce a page representation in a browser alongside page events. In the end, we will present Javascript as a scripting language for the creation of a responsive web page.

The process of generating a web page follows several steps. A browser parses the HTML line by line. If a script occurs, the browser starts to execute the code. The order of processing is important for manipulation with an HTML structure. This limitation can be solved by web events mentioned later, but it is a convention to add scripts to the end of the body part after all HTML tags are parsed.

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

Events are the most common method of how to react to changing a web page state. Every event can have some handlers(listeners). Whenever an event occurs, it calls all its listeners. There are many event types, but we will mention the ones that are important for us. HTML tags are the most common entities, which can have some events. For example, a button has an event onclick which triggers when a client clicks on the button as we ca see in listing 1.5. Other events can represent a state of a page like onload which fires when the whole HTML document is parsed.

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

ECMAScript is a Javascript standard recommending across browsers. It is abravieted as ES. We can see later an abbreviation ES2015 which relates to the ECMAscript version. Javascript is a high-level language usually executed by a browser's dedicated Javascript engine but can also be run on a desktop. Node.js is an example of a Javascript runtime running outside a browser. Listing 1.5 will be used to show the language in the simple scenario. The page contains a button that invokes an alert with a second delay when a client clicks on it.

Listing 1.2: A Javascript code.

```
<!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>
```

At first glance, we can see the type system is dynamic as well as PHP. A window is an essential global variable, which is an object representing the browser window of the running script. The window object consists of all defined global variables. It also contains a document property, which is an API for manipulating the DOM tree. We can see the usage of the document property in the example. Javascript object is often used as a wrapper of Web APIs.

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

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

Javascript is single-threaded, which can be confusing with its constructs for promises. The promise is a structure representing an unfinished process. These processes can be chained. However, the structure can give an illusion of multi-threading. It uses the scheduler for planning the next task executed by the main thread. The single thread is critical for blocking operations which causes thread freezing.

Web workers [11] are a browser feature enabling to run the script in the background. There are wrapped as Worker objects. The worker limitation is communication with UI thread only by handling message events. Messages have to be serialized and deserialized.

Javascript module is the last thing, which we will need to present. It gathers a collection of code. Global entities of this code can be exported to another script. These exports make an API of the module. The module's advantage is defining the API and hiding the internal code, which is not relevant for the user.

1.3 WebAssembly

WebAssembly [9] is a new code format that can be run in today's browsers. It has a compact byte format, and its performance is near to a native code. WebAssembly 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++. The advantage of this format is a similarity with Javascript modules ES2015 after compilation into a machine code. This enables browsers to execute it by a JavaScript runtime. So its security is as good as a code written in Javascript. Because of the same runtime, WebAssembly can call Javascript and vice versa.

Threads [10] support is currently discussed nowadays and appears to be promising. After all, new versions of Google Chrome experiments with proper multi-threading support despite the chance of vulnerability. A replacement of multi-threading can be Web workers mentioned in Javascript section.

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

1.4 Project PHP in browser

The project [2] aims to use compiled PHP interpreter into WebAssembly, which allows evaluating a PHP code. The page has to import a specialized module php-wasm. A PHP code is evaluated by writing a specialized script block or manually by JavaScript and API. PHP can afterward interact with JavaScript using a specialized API. At first glance, that might be a good enough solution, but they are several parts that can be problematic due to PHP semantics. The solution doesn't solve superglobals. This is reasonable because this is the server's job, but you are not able to get information about a query part or handling forms without writing a JavaScript code. The next problem is navigating how a script can navigate to another script without an additional support code which has to be JavaScript.

1.5 C# and .NET 5

We have to introduce the Common Language Infrastructure (abbreviated CLI) [18] before diving into .NET. CLI is a specification describing executable code and runtime for running it on different architectures. CLI contains descriptions of a type system, rules, and the virtual machine (runtime), which executes specified Common Intermediate Language (abbreviated CIL) by translating it to a machine code. The virtual machine is often named CLR (Common Language Runtime).

CIL's advantage is a compilation target of languages like C#, F#, and C++ CLI, which gives us great interoperability. .NET Framework, .NET 5, and Mono are implementations of CLI.

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

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

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

1.6 Blazor

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

From now on, I will use Blazor App to refer Blazor WebAssembly App. The application can be hosted by a standalone project representing a standard ASP.NET web server. It will become useful for further server settings, and the proposed solution utilizes it. The division enables a choice of a place for the implementation of business logic. If there is a bad connection, we can move the majority of business logic to the client and use the server for connection to a database; otherwise, we can use the client only for rendering the page. It consists of the following components. When we choose the template, there are two main projects to describe.

The first one is a server, which serves the Blazor App to a client. There is nothing special about the project expects a middleware, which provides the Blazor files. A middleware is a segment of an HTTP request pipeline, which cares about some functionality related to processing the request.

We will describe the second project (Blazor App) in figure 1.1 to explain basic entities and their interaction with each other.

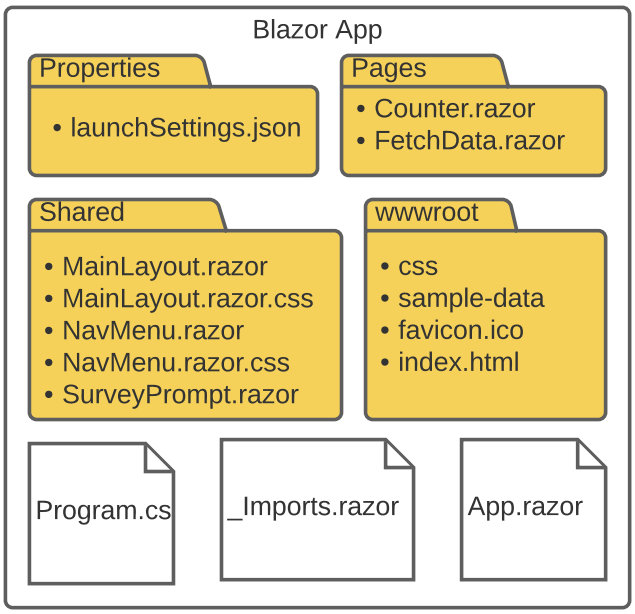

Figure 1.1: A basic Web Assemby App project.

We will start with a new format Razor to get familiar with it. Razor is a markup language interleaving HTML with C#. Razor uses special sign at with keywords to identify C# code in HTML. Razor's compilation results in a pure C# code representing the web page fragment. We can see an example of Razor in listing 1.3.

Although the format is self-explaining, we point to the keywords. The first line begins with a page keyword determining a part of the page's URL. The next keyword is inject, representing a HttpClient service injection. An if keyword determines a standard condition. A code keyword contains a regular C# code, which can be used in the whole razor file.

A Razor file is transformed into a C# dedicated class. The class inherits from ComponentBase or implements IComponent, which provides necessary methods for rendering the page. Components can be arbitrarily put together in order to form the desired page. We can see the generated Component from listing 1.3 in listing 1.4.

We can assign the Razor keywords to parts of the code in the listing. Page keyword stands for Route attribute. Inject keyword stands for parameter attribute. The parameter is assigned by a dispatcher, mentioned later, during the initialization. Code keyword is a part of class content. Another markup is transformed into calling a specialized method in the BuildRenderTree function, which describes the page content for rendering. There will be more information about rendering later in this section.

A Component has several stages, which can be used for initialization or action. Virtual methods of ComponentBase represent these stages. We can see

Listing 1.3: Example of Razor page.

```
@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;
  }
}
```

Listing 1.4: Razor page generated to the C# class.

```
[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;
  }
}
```

the OnInitializedAsync method, which is invoked after setting the component's parameters.

We should mention asynchronous processing because it helps render a page with long-loading content. Blazor allows using Tasks and async methods. Blocking operations in Blazor are projected into UI because it is single-threaded due to Javascript and Web Assembly.

We return to the project description. Folders Pages and Shared contains parts of Blazor pages written in Razor. _Imposts.razor contains namespaces, which are automatically included in others .razor files. The next folder is wwwroot, containing static data of the application. We can see index.html, which cares about loading parts of the Blazor application to the browser.

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

Listing 1.5: A Javascript code.

```
1 <body>
2   <div id="app">Loading...</div>
3   <div id="blazor-error-ui">
4     An unhandled error has occurred.
5     ...
6   </div>
7   <script src="_framework/blazor.webassembly.js"></script>
8 </body>
```

The index.html contains a script initializing Blazor. The first step is to load all resources, which are defined in a separate file. Blazor cuts all unnecessary .dll to reduce the size. For this reason, all .dll has to be used in the Blazor App code in order to be contained in the file. These resources contain Mono runtime compiled into WASM, additional supporting scripts, and all .dll files containing the whole application (Blazor App with referenced libraries). The supporting scripts initiate the runtime a run it. The runtime includes the .dll into the application and calls the Main method in Program.cs defined in Blazor App project. We can see the process in figure 1.2.

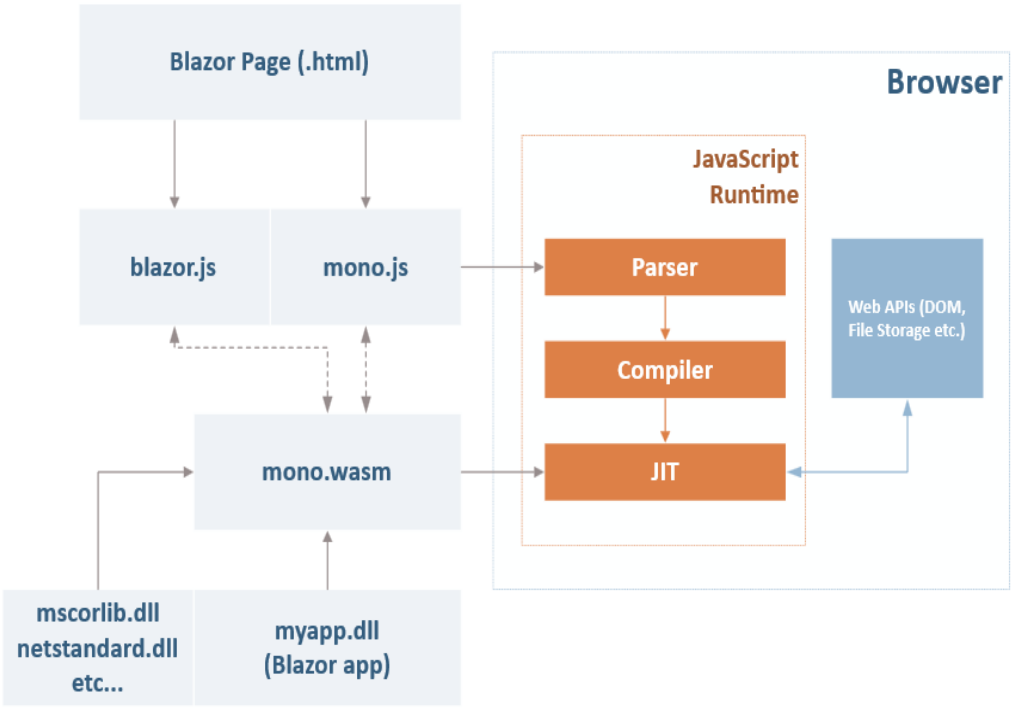

Figure 1.2: Running a Blazor WebAssembly App on client-side.

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

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

In the end, we will describe page navigation and rendering and handling events. The navigation [14] can be triggered by an anchor, form, or filling up the URL bar. The URL bar is handled separated by a browser. JavaScript can influence the remainings elements. Blazor App handles only an anchor. 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. The method represents a navigation handler in Blazor App. A user can add listeners to the handler, but the Router implements default behavior for navigating. The Router finds out all components, which implements an IComponent interface, by a reflection, and tries to render the page according to path matching RouteAttribute of a component whenever the navigation is triggered. The navigation can be redirected to the server if there is no match.

The rendering process begins with the Renderer initialized in the application's builder. Render cares about representing a virtual DOM of the page, creating a page's updates, and updating it with the runtime interop's support. Renderer provides RenderTreeBuider for describing page contents. The builder provides an API for adding various types of content to Batch, which is a specialized structure for describing previous and present virtual DOM changes. The changes are recognized by a diff algorithm, which is used to reduce changing a DOM directly

in a browser to its demanding performance. The usage of RenderTreeBuilder is complicated due to the algorithm. The purpose of Razor is to make an implementation C# method BuildRenderTree easier. When the Renderer prepares the Batch, it calls specialized Javascript API for changing the page through Mono runtime.

The diff algorithm is used to minimize the browser's DOM update after all components used RenderTreeBuilder to render their content. This algorithm used sequence numbers for parts of HTML to identify modified sections. Sequence numbers respond to an order of RenderTreeBuilder's instructions in the source code. A benefit of this information is detecting loops and conditional statements to generating smaller updates of DOM.

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

Blazor provides API for invoking Javascript functions and vice-versa.

1.7 Peachpie

Peachpie [16] is a modern compiler based on Roslyn and Phalanger project. It allows compiling PHP into a .NET assembly, which can be executed alongside standard .NET libraries. Peachpie introduces several structures representing states, scripts, and variables of PHP written in Csharp. The first of them is a context representing one request to PHP code. The context consists of superglobals, global variables, declared functions, declared and included scripts. The possibility of saving the context and using it later is a significant advantage used in the solution. The context can also be considered as a configuration of the incoming script's execution. All information about a request can be arranged to mock every situation on the server-side. The compiler offers a dedicated type of assembly for PHP libraries. Using this assembly can add additional functions, which can provide an extra nonstandard functionality as an interaction with a browser. Another advantage of the compiler is the great interoperability between PHP and .NET. An option to work with Csharp objects, attributes and calling methods will become crucial for achieving advanced interaction between Blazor and PHP.

Saving script information in the assembly

Inheriting the CSharp classes

However, there are limitations following from differences in the languages and the stage of development. Availability of PHP extensions depends on binding these functions to CSharp code which gives equivalent results. The time and memory complexity of this code can be tricky in Blazor. The previously mentioned interoperability has limits as well. Csharp constructs like structs and asynchronous methods are undefined in PHP.

2. Problem analysis

This chapter describes the main problems together with using technologies to help to solve them. In the end, we will introduce the proposed solution to the problem.

Current problems of migrating PHP to a browser divides into different types of tasks that have to be done. In the beginning, we have to load scripts alongside the Blazor app into a browser. Afterward, navigation to the scripts is essential for making the PHP website client-side. The ability to find the desired script should be adopted to Blazor's environment in order to combine PHP scripts with Razor pages. Mocking the server role on the client-side will be another important feature to make this migration familiar with standard PHP usage. It consists of managing superglobals like GET or POST, file management. Because we can save the application state, we bring a new feature of choosing the preservation of the script's context to the next evaluation. The interaction with a user is a challenge for PHP due to its server role. And the last problem will be with evaluating the PHP code. Rendering the whole page is a demanding computation. It can be critical for sections, which tend to change their content often.

2.0.1 Proposed solution

The main idea of migrating PHP to a browser is to integrate Peachpie with Blazor. In the beginning, we have to think of how we can put the scripts into a browser. Peachpie can solve it. It compiles our scripts into .NET assembly, which consists of all information about the scripts. We can reference it from a Blazor app as a standard CSharp code when we have the assembly. We have to ensure that compiler will think the application uses the assembly due to cutting unnecessarily assemblies mentioned in the Blazor section. The process results in loading the assembly alongside the Blazor application into a browser. However, this could be considered as the major part of the problem. How to reference and evaluate the scripts is not clear, and there will be many decisions that will not be silver bullets.

We can use a Blazor component class in order to represent a particular script in the Blazor application. This component should be able to find and evaluate a specified script from the assembly. This approach can benefit from the component's reusability. Afterward, we will be able to compose the component with others to make the desired layout. Before that, we have to make finding and evaluation the script clear.

From this time, we have to distinguish between the purposes of the PHP code. We will create two components for each of them. We make the reason clear later in the section and describe detail in the following chapter.

The first purpose is to free the script from Blazor. It is done by finding the script by the name obtained from a component's parameter or the URL. The script's evaluation is done via caching its output and adding it as markup text to the RenderTreeBuilder. It is is a good approach for transparent use of Blazor, but it is ineffective for often rendering.

User interaction -¿ forms

Superglobals get, post, files

The second purpose is to offer the complete interface of Blazor. An inheritance can achieve this. The PHP class can inherit the CSharp component class due to Peachpie. The consequence of this is accessible Blazor functions for rendering the content.

However, these purposes are different. They can be combined due to a Component interface.

Full control over the rendering

Conclusion

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment