Briz Documentation

Release 0.36.0

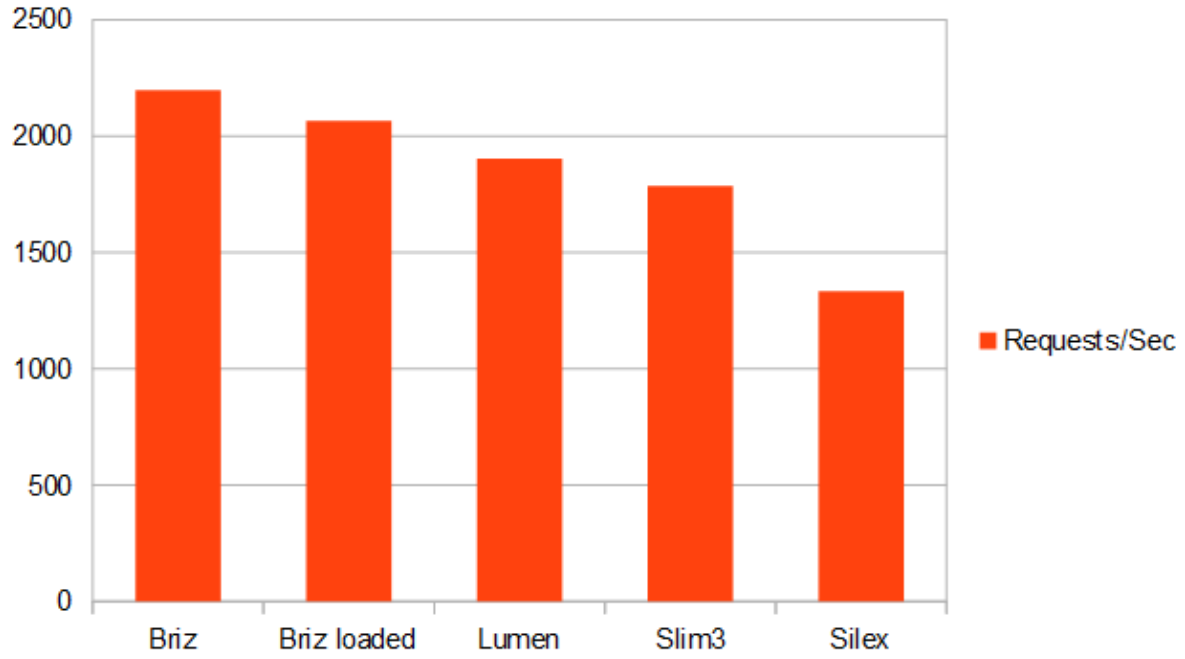
Briz

September 27, 2017

Contents

Briz is an easy to use PHP framework designed to write powerful web applications ranging from simple api to large web applications.

Rapid to Develop and Quick to run



It is faster than many micro frameworks out there. Even if you use most features like route inheritance, identity, controllers and views (as in `Briz loaded` in the chart) it will still be faster than many popular micro frameworks. so dont worry about speed.

Fork Briz on GitHub

CHAPTER 2

Features

Briz has a good set of features to help you.

Easy to Add Dependancies

Briz uses dependancy injection. It is not that strict. so adding dependancies is very easy.

Introducing new Routing System

Briz comes with a new Routing system. it helps in easily extending and seperating your web application. Route inheritance helps you to specify a route extends another route. this works just as in a programming language.

Identity

Identity is about identifying a route from one another. this feature can also be used in other parts using a trait.

PSR-7

Briz supports PSR-7. You can work interoperably with any Psr-7 message implementations.

view *Briz Hello world* for basic hello world examples or view *Quick Start* for more details.

Basic Usage

```
require './vendor/autoload.php';
$app = new Briz\App();
$app->route("web", function($router){
        $router->get('/',function($b){
                $b->response->write('hello world');
        }
        $router->get('/{name}',function($b){
                $data = 'hello'.$b->args['name'];
                $b->response->write($data);
```

```
        });
});
$app->run();
```

Contents:

Briz Hello world

Since Briz is a super flexible framework. you have many ways to write a hello world program.

install

You need composer to install Briz.

```
$ composer create-project briz/briz helloworld
```

this command creates briz inside the directory helloworld.

Basic Hello world

This version uses `echo` instead of PSR-7 `response` object. edit `index.php` in `www` directory.

```
<?php
//autoload the required files
require './vendor/autoload.php';

// create a new application instance
    $app = new Briz\App();

//create a router with the name web
$app->route("web", function($router){

    //a route for GET requests to '/'

            $router->get('/',function($b){
                    echo "Hello World";
            });
 });

//run the application
$app->run();
```

PSR-7 Hello World

this version uses built in PSR-7 `response` object. edit `index.php` in `www` directory.

```
<?php
   //autoload the required files
   require './vendor/autoload.php';

   // create a new application instance
       $app = new Briz\App();

   //create a router with the name web
```

```
   $app->route("web", function($router){

       //a route for GET requests to '/'

               $router->get('/',function($b){
                       $b->response->write("hello world");
               });
    });

   $app->run();
```

Hello World with Renderer

this version uses renderer so that we can use view. edit `index.php` in `www` directory. some framework features requires usage of renderer to work.

```
<?php
//autoload the required files
require './vendor/autoload.php';

// create a new application instance
    $app = new Briz\App();

//create a router with the name gen
$app->route("gen", function($router){

    //a route for GET requests to '/'

            $router->get('/',function($b){
                    $b->renderer('hello');
            });
 });

$app->run();
```

now create a directory `gen` (router name) inside `MyApp/views` and create a file `hello.view.php` in it with any content like this one,

```
hello world
```

HelloWorld with controller

```
<?php
//autoload the required files
require './vendor/autoload.php';

// create a new application instance
    $app = new Briz\App();

//create a router with the name gen
$app->route("gen", function($router){

    //a route for GET requests to '/'
            $router->get('/','HelloController@sayHello');
 });
```

```
$app->run();
```

now create a file (controller) `HelloController.php` in `MyApp/Controllers` directory.

```
<?php
namespace MyApp\Controllers;

use Briz\Concrete\BController;
class HelloController extends BController
{
    public function sayHello()
    {
        $this->response->write("hello world");
    }
}
```

more

usage of Briz framework for creating a web app and simple mobile api at the same time is explained in *Quick Start*

Quick Start

Let's create a simple two page website and a mobile application API for the website. this is very simple when using Briz.

Installation

You can install Briz using composer which is the recommended way.

```
$ composer create-project briz/briz quickstart
```

replace the quickstart with the path to the directory you want to install. this will install all depandancies and make it ready to run.

if you like the other way you can use git-repository . clone it or use download zip option. then run the following command after navigating to the target directory (after extracting th zip if you did it in that way),

```
$ composer install
```

Running

You can easily run Briz navigating to `www` direcory and run the php Development Server. assuming the code is at the directory quickstart, if you prefer other servers such as apache don't frogetpoint it to `www`

```
$ cd quickstart/www
$ php -S localhost:8000
```

now navigate to `http://localhost:8000` using your favourite browser. you can see the Briz web page running.

Code

now use your favourite editor to open `www/index.php`. the default code in this file will be enough, since we are using the same code for this quickstart. let me explain,

web

```
<?php
require '../vendor/autoload.php';
$app = new Briz\App();

$app->route("web", function($router){
    $router->get('/',function($briz){
        $data = 'hello';
        $briz->renderer('hello',['title'=>'Welcome to Briz Framework',
            'content'=>'Thanks for using Briz Framework. You can Learn More at Our Documentaion Page
            'link'=>'http://briz.readthedocs.org/en/latest/'
        ]);
    });
});

$app->run();
```

this code will create a new Briz app and stores a reference to it in the `$app` variable. `$app->route()` function accepts four arguments but only two is required. first argument is route name it is gven as "web". the second argument is a callbak function. to which accepts we pass `$router`. now we can use `$router` to generate actual routes.

`$router->get()` function is used to create a Route with HTTP `GET` method. the first argument `/` indicates that it will match with the url `http://localhost:8000`. you can find more on *Route Patterns* section. the second parameter is a function which accepts a bridge $briz. `$briz->renderer()` is a function to render using a renderer which determine which view class should be used for loading. the first argument is `hello`. which is the route file it will be stored as `hello.view.php` under `MyApp/views/web`. keep thid file as it is.

mobile

Now you can create a mobile API for the above website with very few lines of code. The router `web` in above code contains only one route. even if it contains hundreds of routes we can create mobile part without much difficulty as long as you stick with `renderer` for rendering. now add the following code before `$app->run();` . what we do here is making the route `mobile` as a child of route `web`. so it will inherit the properties of it's parent. and set an *Identity* `header` to it for constrining access. and set the renderer to render results as Json.

```
$app->route('mobile',function($router){

        // Identify using header identity.
        // if a header With X-Request-Cli with value android encountered then it will use this route
        $router->identify('header','X-Request-Cli','mobile');

        //now the responses will be rendererd as json
    $router->setRenderer('JsonView');
 },'web');
```

Thats it. our website will render pages as json when acccessed with mobile. `$router->identify()` is used to provide an identity for the route. so, this will run only if certain conditions are satisfied. here we use a header identity which will match if there is a header with `X-Request-Cli` with the value `mobile`. then we set a render to JsonView. which means renderer will output everything as json. the last parameter `web` is the parent name. now we have to run it. for that lets make an html page `mobile.html` to mock mobile by sending the required header.

```
<html>
<body>
<script>
var request = new XMLHttpRequest();
```

```
var path = "http://localhost:8000";
request.onreadystatechange = function(){
if(request.readyState == 4 && request.status == 200)
{
    document.getElementById('yt').innerHTML = request.responseText;
}
};
request.open("GET",path,true);
request.setRequestHeader("X-Request-Cli","mobile");
request.send();

</script>
<p id ="yt">loading...</p>
</body>
```

This html code is just for sending the required request. i am not explaining it. it sends a request to localhost:8000 including header `X-Request-Cli` with value `mobile`. open this page in browser to see the result yourself. this is a json response our mobile device have to decode it and render it using mobile ui.

Note: if `mobile.html` is not loading then it is due to Cross Origin policy in php development server. in that case add cors support or move `mobile.html` to `www` directory and access it using `localhost:8000/mobile.html`

---

now your website is ready for both mobile and web. if accessed directly in browser it will provide text rich result. but when accessed using `mobile.html` it will send response as JSON.

If you want you can create an *Identity* for a checking domain or sub domain and use it instead of header identity. multi platform is not only the use of identity. you can create an identity for checking user roles and then create different routers for different users. this will create a perfect seperation between users. and there are many other uses. visit *Identity* to learn how to create an identity

Basics

Routing

in modern web frameworks all requests will be handled by a router. in our case there are two files in www folder 1 is an .htaccess file and other is an index.php file all requests to this web server will come to the access point index.php since it is the only web accessable part. so without .htaccess file our typcal request will be `localhost:8000/index.php/hello` but our .htaccess file passess all requests to index.php if there is no file extension such as .php .js etc. so we will have to use only `localhost:8000/hello` instead of `localhost:8000/index.php/hello`.

well, that is not the routing we are talking about. routing is done by router component. it generates appropriate responses based on the request for this it uses the url after `localhost:8000` here it is `/hello`. so if create a route for `/hello` it defines what should we do when our request reaches `localhost:800/hello`.

you can find more about routing at *Routing* page.

controller

controllers controlls the route. it is the main component of the system. when a request is recived by index.php it is passed to the router. router finds an appropriate controller to do actions related to the path in the url. and then this controller decides what to do. in Briz you can pass set a controller in two ways. as a callback function or as a Controller class. the

```
//Using Controller as a callback
$router->get('/{name}',function($briz){
                $briz->render('hello',['name' => 'briz->']);
});

//Using Controller as a class
$router->get('/profile/{name}','ProfileController@show');
```

‘’

when using a controller as a callback you are using can use passed $briz to handle rendering. then when using controller as a class it should be stored in the controller namespace. by default it is at `MyApp\controllers` so you should store it there. the `ProfileController@show` shows that we should use the public method show in the controller class named ProfileController

we have another options of using to directly define route in `$app->route()` that will be explained in *Routing* section. more on controllers can be found at *Controllers* page.

Views

Views are all about how the data is sent to the user. in a typical MVC workflow the application logic is seperated from view. we pass the output generated from controller to the view. The renderer is responsible for rendering the view. view_engines or renderers decide how to process the input. the default view_engines consists of components like ParsedView and JsonView for rendering html and JSON respectively.

Routing

The basics of Routing in Briz is explaind at *Routing* section in Basics you can have a look at it.

Basic Routing

The `App::route()` will create a new router with the name specified. we can use this router to create routes.

lets discuss about basic routing,

```
$app = new Briz\App();
$app->route("web", function($router){
        //code
        $router->get('/','IndexController@show');
   });
```

that will create a router with the name `web`. inside it you can specify routes. here `$router->get('/','IndexController@show');` is such a route

the second argument to the above method can be an anonymous function or a controller. if a controller is used the above code will change to

```
$app = new Briz\App();
$app->route("web", "SomeController");
```

where `SomeController` is the controller name . if so routes will be generated from the controller. see *Controller Routing* below. but i think the anonymous function version is better because it provides more flexibility. that was my personal opinion. you can use anything you like.

GET Route

The code shown above `$router->get()` is a get route.

```
$router->get('/details','IndexController@show');
```

This defines a GET route to the Controller `IndexController` and the method `show` in it. if this route matches to our url then it will execute the method `show` in `IndexController`. if no method name is specified it will use `index` method by default.

if you are using a callback function instead of a controller you can use.

```
$router->get('/details',function($b){
                $b->renderer('index',['title'=>'home']);
}
```

POST Route

```
$router->post('/','IndexController@create');
```

you can use POST routes when you have want to match a route with http POST method. see *getParsedBody* in *Request* for how to retrive post body.

PUT, DELETE, OPTIONS and PATCH

You can also create routes for PUT DELETE, OPTIONS and PATCH. but some of these routes may not be supported by most of the browsers. so you will need an adapter to convert browser requests to these HTTP methods from a dummy method such as post. see *Faking request method*.

```
//example of a put route

$router->put('/','IndexController@edit');

//example of a delete route

$router->delete('/','IndexController@delete');

//example of options route

$router->options('/','IndexController@show');

//example of patch route

$router->patch('/','IndexController');
```

ANY

If you want a route to match all the HTTP methods specified above. then you can use `any` method.

```
$router->any('/help','HelpController');
```

this will match any methods specified above. it will route to `index` method of `HelpController`

Matching more than one method using set

You can also match multiple Http methods with `set`

```
$router->set(['GET','POST'],'/help','HelpController');
```

the first parameter is an array specifying http methods. in the code above GET and POST methods are specified.

Controller Routing

It is possible to pass router directly to a controller. I am assuming you read *Controllers* before reading this resource.

when a router is directly passed to a Controller. the controller will have to generate routes. for this we use Docblock. Here `@Route` is used to specify a route

```
<?php
namespace MyApp\Controllers;

use Briz\Concrete\BController;

class AdminController extends BController
{
   /**
    * @Route [get]
    */
    public function index()
    {
        $this->response->write('Administration Panel');
    }

   /**
    *
    * @Use app
    * @Route [get,post]
    */
    public function details($app,$n)
    {
        $this->renderer('details',[]);
    }
}
```

here the comma seperated values in `@Route` inside `[]` in the doc block is used as HTTP methods to resolve routes. these routes are resolved as `/admin` since the first method name is `index` and `/admin/details/{name}` . the first part is the name of the controller before `Controller` in lower case. here it is `admin`. the second part is the name of the method. and last part is parameters passed to it. if there are `n` number of `@Use` anotation then first `n` parameters will be discarded. here it discards the `$app` in `details` this way

Route Inheritance

The first parameter passed to `$app->route()` is route name. the second parameter is controller and the third parameter is parent. the parent should be name of another router.

a *child* route will have all the properties of its *parent*. including routes and identities. as stated in *Quick Start* the main advatage of this is to extend current route.

if a child route matches then its parent will be ignored.

Route Patterns

```
//static route matches only /hello
$route->get('/hello','handler');

//dynamic route matches with hello/*
$route->get('/hello/{name}','handler');

// Matches /user/42, but not /user/xyz
$r->get('/user/{id:\d+}', 'handler');

// Matches /user/foobar, but not /user/foo/bar
$r->get('/user/{name}', 'handler');

// Matches /user/foo/bar as well
$r->get('/user/{name:.+}', 'handler');

// This route
$r->get('/user/{id:\d+}[/{name}]', 'handler');
// Is equivalent to these two routes
$r->get('/user/{id:\d+}', 'handler');
$r->get('/user/{id:\d+}/{name}', 'handler');

// This route is NOT valid, because optional parts can only occur at the end
$r->get('/user[/{id:\d+}]/{name}', 'handler');
```

Controllers

controllers are a great way to keep code clean. From here onwards only controller class(not closure) will be treated as controllers and closure controllers will be specifically called closure controllers(see *Basics*). controllers helps to increase reusability of code. and it helps to group code better based on functionality

Basics

You must specify a controller directly from router or pass router to controller.

to specify a controller from router

```
$app = new App();
$app->route('name',function($r){
        $r->get('/','IndexController');
        $r->get('/contact','IndexController@contact');
},'parent',$args);
```

`IndexController@contact` means the `contact` method at `IndexController`

here a router instance is passed to `$r` and using `$r->get()` we set route `/` to `IndexController` since no method is specified it will use the default `index` method. next we set `/contact` to `contact` method of `IndexController`. `@` is the spererator to identify controller from action (action is another name commonly used for methods inside a controller)

Or we pass the controller to router.

```
$app = new App();
$app->route('admin','AdminController');
```

here we passed the router to `AdminController` . more about controller routing is available at *Controller Routing* section.

Creating a controller

a controller must be inside controller namespace which is by default `MyApp\Controllers` if you look at `MyApp/Controllers` directory you can see a file `ErrorsController` Dont delete that file. it is your responsibility to care that file. you can edit it in the way you want but dont delete it.

back to creating controller. the controller should extend `Briz\Concrete\BController`. if you dont do so nothing is wrong. it will still work. but you will not get access to some helper methods such as `renderer()` and `show404()`

basic structure of a controller is as follows

```
namespace MyApp\Controllers;

use Briz\Concrete\BController;

class IndexController extends BController
{
   /**
    * Index Page
    */
    public function index()
    {
       $this->renderer('index',['param'=>'value']);
    }

   /**
    * Contact page.
    *
    * @Use app
    */
    public function contact($app)
    {
        $data = $app;
        $this->renderer('index',['name'=>$app]);
    }
}
```

this is our `IndexController` with the routes we defined above in *Basics* . index method will match with the route `/` and contact method will match with the route `/contact` if used with the routes defined above. if you want to know what is the `@Use` in the docblock above the function `contact`, read the section below.

Dependancy Injection and Passing Values

Dependancies are the components used by the Briz. In Briz adding a dependancy is very easy. dependancies are added using the config files in *config* directory. there is a container for storing all the dependancies. every value stored in this container can be accessed

consider there are two routes `/profile` and `/profile/{name}` for get method (see *Route Patterns*). which will point to the methods `index` and `showProfile` in the controller `ProfileController`

```
namespace MyApp\Controllers;

use Briz\Concrete\BController;
```

```php
/**
 * Profile controller
 *
 * @Use auth
 */
class ProfileController extends BController
{
   /**
    * Index Page
    *
    * @Use membership
    */
    public function index($mem)
    {
       $min = $mem->type();
       $name = $this->auth->getname();
       $this->renderer('profile',['name'=>$name,'mem'=>$min]);
    }

   /**
    * show profile.
    *
    * @Use app_name
    */
    public function showProfile($app,$name)
    {
        $data = $app;
        $this->renderer('index',['name'=>$app,'user'=>$name]);
    }
}
```

we can pass dependancies using `@Use` in docblock. it specifies which components should be loaded from container. by default you will have request and response dependancies injected. you can pass more using `@Use`. here we use two imaginary components `auth` and `membership`. if we want a dependancy available everywhere in the class the @Use can be used above the class as in the example above. in that case it will be in the form $component where *component* is the key for the value stored in the container use it as `$this->component` inside a method. when the dependancy is only needed inside a method we can pass it using `@Use` above the method. in that case it will be passed to the arguments in the function. if there are two injections then it will be passed to first two function parameters in order. this is done by the internal `ControllerResolver` which resolves the controller for router.

the `showProfile` method has one parameter from `@Use` and one from `/profile/{name}` the parameter from route will be stored to $name in that method. which means the @Use parameters will be resolved before route parameters

there is no limit on what should be stored in container. here `app_name` is a string storing the application name. you can edit it inside `config/application.php`.

Note: if you dont want to use or don't like using `@Use` annotation to get values. you can simply use something like `$mem = $this->container->get('membership')` inside the method. the `container` holds reference to all dependancies. you can access it using `$this->container->get('key')` method. more about this at *Container Reference*.

Available Methods

The Following Methods are Available from BController class. if you create a new `Response` object then there must a `return` statement with that `Response` or set `$this->response` to that object.

show404

displaying 404 page. you can edit its default look inside `ErrorsController`

show404()
Show a 404 response message

Returns `Psr\Http\Message\ResponseInterface` with 404 Page

usage

```
public function show($name)
{
    if($this->notFound($name)
    {
        return $this->show404();
    }
}
```

redirect

redirect(*$url*, *$code=302*)
redirect to a given url

Parameters

- **$url** (*string*) – Url to redirect
- **$code** (*int*) – code to redirect.

Returns `Psr\Http\Message\ResponseInterface` with redirect

usage

```
public function show($name,$redirectUrl)
{
    if($this->isAuthorized)
    {
        return $this->redirect($redirectUrl);
    }
}
```

renderer

use ther selected renderer to process the input to generate response

renderer()
minimum number of arguments is two. but can have more than that based on number of responses.

Parameters

- **$name** (*string*) – Name of the rendering

- **$params** (*array|object*) – Array or object containing data in `'key' => 'value'` format

Returns `Psr\Http\Message\ResponseInterface` with processed output

Basic usage

```
public function show($name,$redirectUrl)
{
    $params = $arrayOrObject;
    $this->renderer('hello',$params);
}
```

View

views are the rendering part of Briz. it sets outout for sending response. renderer constructs response from views. where the viewfiles are stored depends upon the viewEngine and views class used. the default view engine uses the configration from `config/view.php` to store the file. the default location is MyApp/views. the default view engine requires you to store it in a directory with the same name as the router.

By default only the class `ParsedView` will use the view file configuration for view directory. it requies you to store files with a `.view.php` file extension. You can use other view engines like twig if you want.

ParsedView

`ParsedView` is a class which uses view directory. it checks if the view file exists and parses array or object to variables to the file. it then stores it's output to response.

if we call renderer like

```
$b->renderer('hello',['name'=>'haseeb','greet'=>'hello']);
```

if the name of the router is *web*. it will check inside `views/web` for the file `hello.view.php` and converts the array to variables. so, we can use it in view file like `$greet` and `$name`. if our view file contains,

```
<?php
echo "$greet $name";
?>
```

it will output

hello haseeb

JsonView

the `JsonView` will generate output in JSON format and sets a Json content-type header. this will not use any files.

DirectView

This view is not of much use. it will just write to response evrything thrown at it. it expects an array to be single dimentional.

View Configuration

the view configurations should be stored inside `config\view.php`

If you open this file you can find what are the values stored init.

Identity

Identity is a global way for checking several conditions are satisfied. an identity will be given access to the Container. using this container an Identity can check for some conditions are satisfied or it can add values to the container.

for example, the default Identity included with Briz `header identity` checks if a header with a specific value exists in the request. for doing this it accesses `request` from container and checks request headers.

An identity should extend `Briz\Base\Identity`. and it must have a method `identify()` which will contain the logic. the container will be available as `$this->container` inside an identity. have a look at *Container Reference* for more details about container

Creating an identity

for example lets create a simple identity for checking port number

create a directory `Identities` under MyApp and create a file `PortIdentity.php`

```
<?php
namespace MyApp\Identities;

use Briz\Base\Identity;

/**
 * Identity for checking port values.
 */
class PortIdentity extends Identity
{
    /**
     * Check for port.
     *
     * @param int $port
     * @return bool
     */
    public function identify($port, $v='')
    {
        $request = $this->container->get('request');
        if($request->getUri()->getPort() == $port){
            return true;
        }
        return false;
    }
}
```

This code checks if a specific port $port is matched with the port number in the request.$v is initialized with blank value because `IdentityInterface` for identities requires two parameters for method `identify()`.

at first, we need to register the identity. for this edit `config\identities.php` and add a key `port` and set its value to our class name with namespace. Then this file will look like,

```
<?php
return [
    'header' => 'Briz\Beam\HeaderIdentity',
    'port' => 'MyApp\Identities\PortIdentity'
];
```

next in your *Quick Start* file (or the default `www/index.php` when briz is installed). edit the line for checking identity.

```
//$router->identify('header','X-Request-Cli','mobile');

$router->identify('port',8080,'');
```

just like above we had to pass a blank third parameter inside `$router->identify()`. now use PHP Development Server to go to `localhost:8000` by

```
$ php -S localhost:8000
```

and then after exiting, goto `localhost:8080` by typing

```
$ php -S localhost:8080
```

check both and see the difference yourself. Now you have created your first Identity.

Using Identities outsite Route

you can also use identities outside the Route. for that, you have to use `IdentityTrait`. As traits provide automated copy paste in php. we can use `IdentityTrait` to add some methods and variables for accessing `Identity` to your Controller.

adding the trait.

```
class IdentityController
{
    use \Briz\Beam\IdentityTrait;

    public function idcheck()
    {
        //code goes here.
    }
}
```

the `use \Briz\Beam\IdentityTrait;` will add the identity management trait for your controller.

now, you have the following methods and two `protected` arrays `$identifies`, the list of identites registered with the class and `$idt_keyIndex`, the key index

addIdentity

add a new identity.

this function can accept any number of arguments. you need to pass minimum two arguments.

addIdentity (*string $name*, *mixed $value*)

Parameters

- **$name** (*string*) – The name of the identity in container

- **$value** (*mixed*) – value to be identified

Throws BadMethodCallException , InvalidArgumentException

removeIdentity()

Remove an added Identity.

removeIdentity(*string $name*, *mixed $key*)

Parameters

- **$name** (*string*) – The name of the identity in container
- **$value** (*mixed*) – second argumet passed when AddIdentity was called.

removeAllIdentity()

if a name is specified it will only remove all identities under that name

removeAllIdentity(*string|null $name = null*)

Parameters

- **$name** (*string|null*) – optional name.

identifyAll()

check if all added identities matches.

identifyAll()

Returns boolean

identifyByName()

identifies everything under a name if no key is specified otherwise check for identity matched by key.

identifyByName(*string $name*, *mixed|null $key = null*)

Parameters

- **$name** (*string|null*) – name of the identity.
- **$key** (*mixed|null*) – second parameter when added Identity

Returns boolean

Providers

Providers are the components used by a Briz application. if a provider is added to the system container it will be available everywhere in the application. in Briz `Request` and `Response` components are providers.since they should be available everywhere in the system they are added to the system container using `config\providers.php` file. everything in this file will be available inside container. the `logger` in this file is an external component called monolog

With Briz you can easily add providers. Briz is designed for easily adding any components into the system. i call it as "just use it strategy". so you can use many external components easily.

Creating a Provider

Since there is no default `model` in Briz, I decided to add it first. it is the M in any MVC framework (Model View Controller). models are classes which should work with data. mainly they are used for communicating with database. since we dont have it by default, it is just VC. so to make it MVC we are adding a model.

there are two types of models which are widely used

1. Active Record.
2. Data mapper.

php-active-record and Eloquent are examples of active record type models. spot-2 and doctrine are examples of data mapper approch.

Both are good and have thier own advantages and disadvantages. you have to find it yourself. here i am going to add php active record as a provider in example.

Example

Here we are going to add php active record as a provider. so that we will have an active record model implementation ready to use. first find details about php active record from Php active record website . there is a quick start guide in that website. i am using the same connection code in that page bellow. but with few changes to pass arguments.

create a directory `Providers` inside `MyApp`. where we will store all our providrs. and create an file `PhpAR.php` in it.

```
<?php
namespace MyApp\Providers;

use Activerecord;

class PhpAR
{
    /**
     * @param string $root root directory
     * @param string $app user directory name
     */
    public static function register($root,$app)
    {
        $cfg = ActiveRecord\Config::instance();
        $cfg->set_model_directory($root.'/'.$app.'/Models');
        $cfg->set_connections(
            array(
                'development' => 'mysql://dev:dev@localhost/bridge',
                'test' => 'mysql://username:password@localhost/test_database_name',
                'production' => 'mysql://username:password@localhost/production_database_name'
            )
        );
        $cfg->set_default_connection('development');
    }
}
```

here we created a class `PhpAR` and a static method `register` in it. next we have to find what values from container are needed by this provider. here we will need to pass the location of the model directory. we have to store this value in `MyApp/Models`. currently `MyApp` directory is at `/home/projects/briz` so i will have to use `/home/projects/briz/MyApp/Models` as path to models. to get this path we have to pass `root_dir` and `app` from container. so we defined two parameters `$root` and `$app` for this function. the rest of the code is copy

paste from the quick start at phpactiverecord website. you have to edit user name and password. visit phpactiverecord website for more details

for this we need to have php active record library available in our application. for this we can use composer.

```
$ composer require php-activerecord/php-activerecord
```

Now registering the provider, if you want to register this provider globally. you can use `config/providers.php` file. it will return an array containing providers to application initialization code which will add this to container. there is a format for adding content to this array.

`NameSpace\Class@StaticMethod@argument@argument2`

So, for our provider we will have to add `"MyApp\Providers\PhpAR@register@root_dir@app"` to this array. here `MyApp\Providers\PhpAR` is the class with namespace. and `register` is the static method and `root_dir` and `app` are parameters from container passed to this provider.

if you want it only available to a single controller. you can just load it inside that controller by passing arguments from container.

Container Reference

this section describes about how to use container and what data are available for you by default.

Using Container

to add data to container use:

```
$this->container['key'] = 'value'
```

to get data from container you have two ways:

```
// access directly
$value = $this->container['key'];

//this method has an advantage of interoperability. so it is recommended.
$value = $this->container->get('key');
```

to check if a key exists in a container:

```
if($this->container->has('key')){
    //do something
}
```

Default data

the default data available in container are the following,

Keys set by `Briz\App.php`

key	Details
`framework`	an instance of `Briz\App`
`root_dir`	the root directory
`inherit`	an object containing details about parent and child relationship
`container`	a reference to container itself.

Keys set by `config/application.php`

Key	Details
`app`	Application namespace
`app_name`	Application name
`controller_namespace`	Controller Namespace
`log_dir`	Directory for logging errors
`display_errors`	if sets to true it will display errors ONLY IF application error_reporting is set
`output_chunk_size`	output chunk size when sending response.
`fake_method`	The request faking string.

Keys set by `config/collections.php`

key	Details
`routes`	a collection containing all routes.
`route_collector`	holds refernce to individual routes for each router.
`id`	holds reference to all idenities registered.

Keys set by `config/view.php`

key	Details
`view_engine`	view engine namespace.
`default_view_class`	default renderer class for view.
`view_dir`	path to the directory where view details are stored.

Keys set by `config/providers.php`

key	Details
`server`	Server parameters in $_SERVER stored as a collection.
`router`	The route collection.
`request`	PSR-7 Request implementation by Briz.
`response`	PSR-7 Response implementation by Briz.
`logger`	helper for logging.
`FakeMethod`	It will change the method of an request based on a head (see fakingrequestmethod)

Collections

A `collection` is a wrapper around an array. basic and most common operations on an array are implemented as methods in a collection. in Briz collections uses `Briz\Base\Collection` class.

Adding an empty collection to Container

If you want an empty collection available throughout the application from start. then you can add it to the container using `config/collections.php` file. you can add a collection in this file by adding `"name" => "init_collection"` to the array.

adding non empty collection to container

To add a non empty container which should be available throughout the application you have to use it as a provider (see *Providers* section)

constructing

you can create a new collection by

```
use Briz\Base\Collection;

$collection = new Collection();
```

Creating from an array

You can create a collection from an array by passing it to the constructor.

```
use Briz\Base\Collection;

$array = [ 'name' => 'value','foo' => 'bar'];

$collection = new Collection($array);
```

Adding/replacing

set

we can add content to a collection by using `set` method. if a key already exists it will get replaced by new value.

Usage:

```
$collection->set('key','value');
```

replace

The `replace($array)` method will replace the collection with the elements of new array.

Usage:

```
$array = ['a'=>'b','c'=>'d'];

$collection->replace($array);
```

merge

To merge values an array with existing collection you can use `merge($array)` method.

Usage:

```
$array = ['a'=>'b','c'=>'d'];

$collection->merge($array);

//to merge two collections
$newCollection = new Collection(['key'=>'value','foo'=>'bb','bar'=>'h']);
$newCollection->merge($collection->all());
```

Get values

get

`get($key, $default)` method is used to get a value from a collection. `$default` is the value which we will use if there is no value available. if we don't set the `$default` the collection will return null if key doesn't exist.

Usage:

```
//If key not found it will return an empty array otherwise return the value
$value = $collection->get('key',[]);

//it will return null  if key is not in collection. otherwise will return the value.
$value = $collection->get('key');
```

all

The `all()` method will return all key value pairs stored in the collection as an array.

Usage:

```
$value = $collection->all();
```

Remove Items

remove

To remove a single item you can use `remove($key)`.

Usage:

```
$collection->remove('key');
```

clear

`clear()` will remove all items from the collection. then the collection will be an empty collection.

Usage:

```
$collection->clear();
```

Checking

has

The `has($key)` method is used to check if a key exists in the collection.

Usage:

```
if($collection->has('key'))
{
    //do something
}
```

isempty

check if a collection is empty with `isempty()`

Usage:

```
if($collection->isempty())
{
    //do something
}
```

count

get the number of items in a array

Usage:

```
$count = $collection->count();
```

Operations

Each

The `each($callback)` method will return the collection after executing a callback function of each elements in the array.

Usage:

```
//callback to add prefix.
$callback = function($key,$value){
        $key = 'rf_'.$key;
        $value = 'rf_'.$value;
}

$collection->each($callback);
```

Request

In Briz Request is a `PSR-7` implementation. it reprecents an incoming request received by the server. since it uses `Psr\Http\Message\ServerRequestInterface` you can access and inspect any request parameters.

the request is available as `request` inside the container (see *Container Reference*). this is loaded to controller by default. so it is available as `$this->request` inside controller. or you can get it from container as `$container->get('request');`

from here onwards assume `$this->request = $container->get('request');`

Request Method

The HTTP request methods supported by Briz are,

- CONNECT
- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT
- TRACE

getMethod

The `getMethod()` function is used to retrieve HTTP method of the request. it returns a string value reprecenting HTTP request method.

usage:

```
$method = $this->request->getMethod();
```

withMethod

With `withMethod($method)` you can create a copy or clone of the current request with another method. so if you want to change the request method you have to use

```
$this->request = $this->request->withMethod($method);
```

where `$method` is a supported HTTP method.

Faking request method

Most browsers dosn't support any methods other than GET and POST. so we can create a fake method. the default method faking string is `X-HTTP-Method-Override`. you can edit it inside `config/application.php` file. if this string is added in `request header` or `request body` it will override with the new one.

Usage:

```
<input type="hidden" name="X-HTTP-Method-Override" value="put">
```

Uri

There is an interface in PSR-7 for reprecenting Uri `Psr\Http\Message\UriInterface`. So in request uri reprecents an object of a class implementing that interface. more about our implementation is explained in *Uri*

getUri

The `getUri()` method is used to retrieve the Uri object

Usage:

```
$uri = $this->request->getUri();

//now you can use $uri object like,
$port = $uri->getPort();
```

withUri

The `withUri($uri)` method will create a request object with a new `uri`

Usage:

```
//this will change the request Uri
$this->request = $this->request->withUri($uri);
```

getRequestTarget

The `getRequestTarget()` method will retrieve our request target. if our uri is `www.example.com/page/item/1?a=b` the `'/page/item/1?a=b` part is called request target. this method will retrieve this part as string.

Usage:

```
$target = $this->request->getRequestTarget();
```

withRequestTarget

The `withRequestTarget()` method will create a clone of the request object with the specified string request target .

Usage:

```
$request = $this->request->withRequestTarget();
```

Server Parameters

The server parameters part of the request are mostly derived from $_SERVER super global. but they can also contain other values.

getServerParams

The `getServerParams()` method is used to retrieve server parameters as an array.

Usage:

```
$params = $this->request->getServerParams();

//then we can use it like $params['xyz'] etc.
```

Cookies

getCookieParams

this method will retrieve request cookies as an array.

Usage:

```
$cookies = $this->request->getCookieParams();
```

withCookieParams

`withCookieParams(array $cookies)` method will clone the request with the given array of cookies

Usage:

```
//this will change the request cookies
$this->request = $this->request->withCookieParams($array);
```

Query String

consider the url `http://example.com/briz/i?a=b&c=d` the `a=b&c=d` part is called query string. query string parameters are `a=b` and `c=d`

getQueryParams

The `getQueryParams()` method will retrieve query string parameters as an array.

Usage:

```
$queryParams = $this->request->getQueryParams();
```

withQueryParams

`withQueryParams(array $query)` method will clone the request with given array of query params.

Usage:

```
//this will change the request query parameters
$this->request = $this->request->withQueryParams($queryasArray);
```

Uploaded Files

There is an interface in PSR-7 to reprecent UploadedFiles (`Psr\Http\Message\UploadedFileInterface`). so we will use our implementation of this interface for reprecting uploaded files. Each uploaded File is reprecented by `Briz\Http\UploadedFile` class.

getUploadedFiles()

This method will retrieve an array of `Psr\Http\Message\UploadedFileInterface` objects reprecenting each uploaded file.

Usage:

```
$files = $this->request->getUploadedFiles();
```

withUploadedFiles

The `withUploadedFiles(array $uploadedFiles)` method will create a clone of the current request with an array of uploaded files.

```
//this will change the uploaded files with current request
$this->request = $this->request->withUploadedFiles($uploadedFiles);
```

Request Body

Http request body usually contains `POST` data. when you use HTTP POST method the request will send your data o server in the request body. You can get this body directly or can get a parsed version (the data in `$_POST` is actually parsed version)

getBody

The `getBody()` method is used to get unparsed HTTP body as it is available in pure HTTP message as a stream object.

```
// unparsed body
$body = $this->request->getBody();
```

getParsedBody

The `getParsedBody()` method will retrieve a parsed body with respect to the content type header as string. the default content types supported are,

Content Type	Return value
`application/x-www-form-urlencoded`	array
`multipart/form-data`	array
`application/json`	array
`application/xml`	object
`text/xml`	object

Usage:

```
$body = $this->request->getParsedBody();
```

registerParser

The `registerParser($type, $callable)` method will add a new parser for parsing current request body. it will override existing parser if an existing content type is used as `$type`. `$callable` is a callback function to parse http body. the callback function must return an array or an object. the callback function will get a string containing unparsed body as input.

Usage:

```
//parser to parse json as object instead of array.
$parser = function($body){
        return json_decode($body);
   };

$this->registerParser("application/json", $parser);

$body = $this->getParsedBody();
```

withBody

the `withBody(StreamInterface $body)` method will return an instance of request object with given stream object.

Usage:

```
$this->request = $this->request->withBody($newStream);
```

Headers

getHeaders

The method `getHeaders()` Retrieves all Request header values as an array.

Usage:

```
$headers = $this->request->getHeaders();
```

hasHeader

`hasHeader($name)` method checks if the current request has a header `$name`. it will return a boolean value.

Usage:

```
if($this->request->hasHeader($name))
{
    //do something.
}
```

getHeader

The `getHeader($name)` method returns header values for the header `$name`. since an Http header can have multiple values it returns an array.

Usage:

```
$header = $this->request->getHeader($name);
```

Note: since some clients send multiple headers as comma seperated values. you may get only single value even if multiple header are there. a better alternative is to `explode() getHeaderLine()`

---

getHeaderLine

The `getHeaderLine($name)` method will return a string containing comma seperated values of the header specified by `$name`

Usage:

```
$header = $this->request->getHeaderLine($name);
```

withHeader

The `withHeader($name, $value)` method will return an instance with the provided value replacing the specified header. please note that this will replace the header.

Usage:

```
$request = $this->request->withHeader($name, $value);
```

withAddedHeader

unlike `withHeader` the `withAddedHeader($name, $value)` will return an instance of request object with appending a header to the current header.

Usage:

```
$request = $this->request->withAddedHeader($name, $value);
```

withoutHeader

it will return an instance of the request after removing a header

Usage:

```
$request = $this->request->withoutHeader($name);
```

HTTP version

The http protocol version supported by briz are `1.0`, `1.1` and `2.0`

getProtocolVersion

The `getProtocolVersion()` method will return HTTP protocol version of the request.

Usage:

```
$version = $this->request->getProtocolVersion();
```

withProtocolVersion

The `withProtocolVersion($version)` method returns an instance of current request with the given protocol version

Usage:

```
$request = $this->request->withProtocolVersion($version);
```

Response

In Briz a `response` object implementing `Psr\Http\Message\ResponseInterface` is used to reprecent outgoing response. Briz will send this as response to client after processing. in briz default response object is `Briz\Http\Response`

the response is available as `response` inside the container. so you can use `$this->container->get('response');` to get the current response object. it is loaded inside the controller as `$this->response`.

This Page assume `$this->response = $this->container->get('response')`

HTTP Status

an HTTP response will contain a three digit status code which explains the status of the current response to the client. the most common HTTP status anybody know is `404 Not Found` since it is an error message. there are other status codes too. in `Briz\Http\Response` status codes and reason phrases are defined as below.

```
protected static $phrases = [
    // INFORMATIONAL CODES
    100 => 'Continue',
    101 => 'Switching Protocols',
    102 => 'Processing',
    // SUCCESS CODES
    200 => 'OK',
    201 => 'Created',
    202 => 'Accepted',
    203 => 'Non-Authoritative Information',
    204 => 'No Content',
    205 => 'Reset Content',
    206 => 'Partial Content',
    207 => 'Multi-status',
    208 => 'Already Reported',
    // REDIRECTION CODES
    300 => 'Multiple Choices',
    301 => 'Moved Permanently',
    302 => 'Found',
    303 => 'See Other',
```

```
    304 => 'Not Modified',
    305 => 'Use Proxy',
    306 => 'Switch Proxy', // Deprecated
    307 => 'Temporary Redirect',
    // CLIENT ERROR
    400 => 'Bad Request',
    401 => 'Unauthorized',
    402 => 'Payment Required',
    403 => 'Forbidden',
    404 => 'Not Found',
    405 => 'Method Not Allowed',
    406 => 'Not Acceptable',
    407 => 'Proxy Authentication Required',
    408 => 'Request Time-out',
    409 => 'Conflict',
    410 => 'Gone',
    411 => 'Length Required',
    412 => 'Precondition Failed',
    413 => 'Request Entity Too Large',
    414 => 'Request-URI Too Large',
    415 => 'Unsupported Media Type',
    416 => 'Requested range not satisfiable',
    417 => 'Expectation Failed',
    418 => 'I\'m a teapot',
    422 => 'Unprocessable Entity',
    423 => 'Locked',
    424 => 'Failed Dependency',
    425 => 'Unordered Collection',
    426 => 'Upgrade Required',
    428 => 'Precondition Required',
    429 => 'Too Many Requests',
    431 => 'Request Header Fields Too Large',
    451 => 'Unavailable For Legal Reasons',
    // SERVER ERROR
    500 => 'Internal Server Error',
    501 => 'Not Implemented',
    502 => 'Bad Gateway',
    503 => 'Service Unavailable',
    504 => 'Gateway Time-out',
    505 => 'HTTP Version not supported',
    506 => 'Variant Also Negotiates',
    507 => 'Insufficient Storage',
    508 => 'Loop Detected',
    511 => 'Network Authentication Required',
];
```

For normal response the status will be `200 OK`. The codes above are the only codes supported by Briz. But you have an option to change the reason phrase.

getStatusCode

The `getStatusCode()` will return the current status code set with the response object as integer.

Usage:

```
$status = $this->response->getStatusCode();
```

getReasonPhrase

The `getReasonPhrase()` method will return the current reason Phrase set with the response object

Usage:

```
$reason = $this->response->getReasonPhrase();
```

withStatus

`withStatus($code, $reasonPhrase = '')` method creates an instance or clone of current response object with new status code. if optional `$reasonPhrase` is given it will change the reasonPhrase from default one.

Usage:

```
$reason = $this->response->withStatus($code);
```

Response Body

Http Response body contains what you will get in browser window. it is the Html part of the response.

write

the `write($data)` or `setContent($data)` will write data to output stream of briz response object which will be latter added to the output stream of php after processing.

Usage:

```
$this->response->write($data);
//OR
$this->response->setContent($data);
```

getBody

The `getBody()` method will return current response body as a stream object (`Psr\Http\Message\StreamInterface`).

Usage:

```
$this->response->getBody();
```

withBody

the `withBody(StreamInterface $body)` method will return an instance of response object with given stream object.

Usage:

```
$this->response = $this->response->withBody($newStream);
```

Headers

setHeader

The `setHeader($header, $value, $replace = true)` will add a new http header. `$header` is the header name and `$value` is the header value in string. if the optional `$replace` is set to `false` it will append the header without replacing it.

Usage:

```
//replace
$this->response->setHeader($header,$value);
//append
$this->response->setHeader($header2,$value,false);
```

Other header methods

other header methods are exactly same as the one in Request *Headers*

HTTP Version

the supported operations on http protocol version are exactly same as the one in request *HTTP version*

Uri

In PSR-7 URI is reprecented by an implementation of `Psr\Http\Message\UriInterface`. we implement this as `Briz\Http\Uri`. according to PSR7 a uri is of the format `scheme://authority/path?query#fragment` or in more detail `scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]`. but what we need in most cases is regular url like `scheme://host:port/path?query#fragment`

You can get the Uri object of current request from `request` inside the container. from here onwards assume that:

```
$this->request = $this->container->get('request');
$uri = $this->request->getUri();
```

Scheme

Uri schemes supported by Briz are `HTTP` and `HTTPS` or it can be blank as a placeholder. Other uri schemes will result in an `InvalidArgumentException`.

getScheme

The method `getScheme()` will return the scheme

Usage:

```
$scheme = $uri->getScheme();
```

withScheme

The `withScheme($scheme)` method will return an instance of the Uri object with the given scheme.

Usage:

```
$uri = $uri->withScheme($scheme);
```

Host

A host can be an IP address or an address like `example.com` or long address like `web.www.ex.example.com`

getHost

The `getHost()` method will return a string host name.

Usage:

```
$host = $uri->getHost();
```

withHost

The `withHost($host)` method will return an instance of the Uri object with the new Host.

Usage:

```
$host = $uri->withHost($host);
```

Port

a port number must be between 1 and 65535. the default port number of http is 80 and that of https is 443. If you are using a port number other than default one then you must specify it. `http://example.com` is actually `http://example.com:80`. since 80 is the default port of http it is nor given. the briz quick start tutorial will start php development server at port 8000. so you have to specify it as `http://localhost:8000` since 8000 is not the default port.

getPort

The `getPort()` method will return the port number as integer.

Usage:

```
$port = $uri->getPort();
```

withPort

The `withPort($port)` method will return an instance of the uri object with the given port number.

Usage:

```
$uri = $uri->withPort($port);
```

Path

getPath

gets the current uri path.

Usage:

```
$path = $uri->getPath();
```

withPath

return an instance of current uri with given path.

Usage:

```
$uri = $uri->withPath($path);
```

Query String

The Query String is separated from the preceding part by a question mark.

getQuery

The `getQuery()` method will return the query string

Usage:

```
$query = $uri->getQuery();
```

withQuery

The method `withQuery($query)` will return an instance of the Uri object with given query string.

Usage:

```
$uri = $uri->withQuery($query);
```

Fragment

the part of the Uri after # is called Uri Fragment. it is usually used as identifiers for sections in html page.

getFragment

The `getFragment()` method will return the Uri fragment as string.

Usage:

```
$fragment = $uri->getFragment();
```

withFragment

The `withFragment($fragment)` method will return a clone of the Uri object with given fragment.

Usage:

```
$uri = $uri->withFragment($fragment);
```

User Information

Uri as defined by `rfc3986` and `PSR-7` can contain user information. user information is usename and password. empty passwords are also valid.

getUserInfo

this method will retrieve `username:password` format string. if there is no password it will return `username`. if there is no usename or password it will return a blank string.

Usage:

```
$user = $uri->getUserInfo();
```

getAuthority

The `getAuthority()` will return the URI authority, in `[user-info@]host[:port]` format.

Usage:

```
$authority = $uri->getAuthority();
```

withUserInfo

`withUserInfo($user, $password)` returns an instance of the Uri with the given user info. the password is optional.

Usage:

```
$uri = $uri->withUserInfo($user, $password);
```

Symbols