Computer science → Backend → Ktor → Ktor Plugins

# Basic And Form Authentication

Theory Practice 0% completed, 0 problems solved

## Theory

24 minutes reading

Verify to skip Start practicing

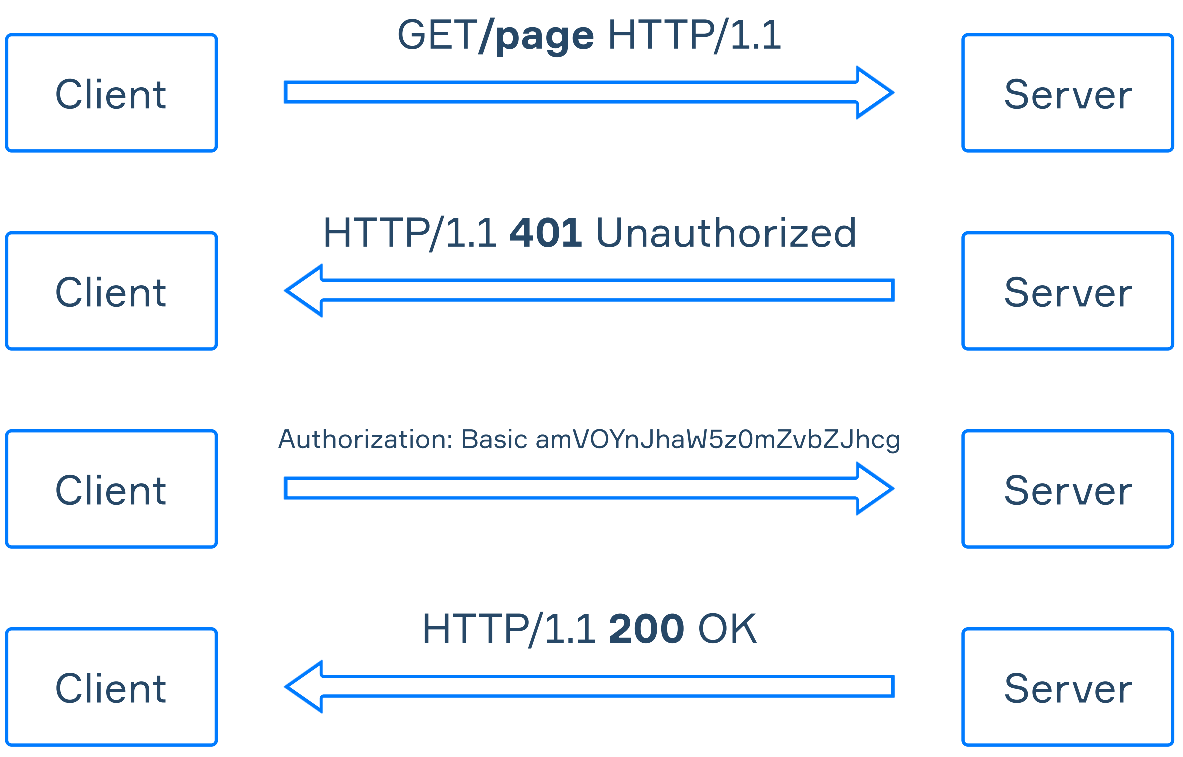
**You're starting a new topic!**
Each topic introduces one new concept with a short article. Read it, then apply what you've learned by solving a few problems. Practice makes perfect!

You are already familiar with the basics of routing in Ktor. What if you want to restrict access to some pages for unauthorized users? Today we'll get acquainted with two convenient authorization tools available in Ktor.

## §1. Basic authentication

**Basic authentication** is a mechanism built into the HTML protocol that makes it easy to limit access to the page by username and password.

Suppose we have a page ( `localhost:8080/page` ) that is protected by basic authentication. The mechanism of interaction between the client and the server, in this case, will look as follows:

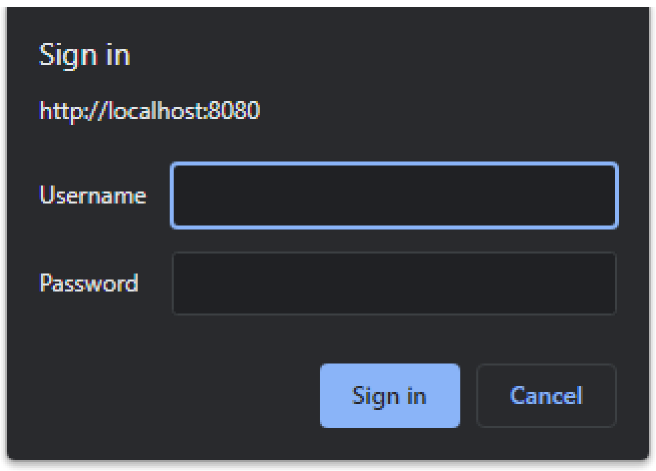


First, the client makes a usual request to `localhost:8080/page`. The server sees the page is protected and returns to the client the information that it needs to be authorized ( `401 Unauthorized` ). The response will contain a header:

```
WWW-Authenticate: Basic realm="Access to the '/page' path",
charset="UTF-8"
```

The `WWW-Authenticate` header tells the browser that the page is protected by authentication, specifying its type, encoding, and `realm` (text message from the server that the browser can show the user).

Upon receiving such a response, the browser will display a dialog box for the user to enter the username and password for authorization:



After the user enters his username and password, the browser will encode them using Base64 and make a second request to get the page. This time, the request will have an authentication data header:

```
Authorization: Basic amV0YnJhaW5zOmZvb2Jhcg
```

The server checks the login and password that are in the Authorization header and, if successful, returns the requested page to the user.

## §2. Basic authentication in Ktor

Ktor has a handy plugin that allows you to connect Basic authorization to your Routing handlers easily.

First, you need to add a dependency in `build.gradle.kts`:

```
implementation("io.ktor:ktor-server-auth:$ktor_version")
```

Now we have to install the plugin. To do this, we need to call the `install(Authentication)` function during server initialization:

```
import io.ktor.server.application.*
import io.ktor.server.auth.*

fun main() {
    embeddedServer(Netty) {
        install(Authentication) {
            basic("myAuth") {
                // Configure basic authentication
            }

        }

        //...

    }.start()
}
```

To set the Basic authentication provider, we call the `basic` function inside the `install` block. The `basic` function takes the authorization name `"myAuth"` as a parameter. We will need this name to refer to the authorization in the Routing handler.

Now let's configure Basic authorization:

```
install(Authentication) {
    basic("myAuth") {
        realm = "Access to the '/page' path"
        validate { credentials ->
            if (credentials.name == "Admin" && credentials.password
== "2425") {
```

```
                UserIdPrincipal(credentials.name)
            } else {
                null
            }
        }
    }
}
```

Inside the `basic` block, we have two properties: `realm` and `validate`.

The `realm` property sets the realm to be passed in `WWW-Authenticate` header.

The `validate` function compares the username and password with the correct ones and returns a `UserIdPrincipal` (with this object we can get in the Routing handler the name of the user who has logged in to our page.) in a case of successful authentication or `null` if authentication fails. Within the `validate` function, there can be any validation of the entered data, including database calls. For simplicity, we just compare them to `"Admin"` and `"2425"`.

Now let's connect our authorization to the `/page` handler:

```
routing {
    get("/page") {
        call.respondText("Protected content!")
    }
}
```

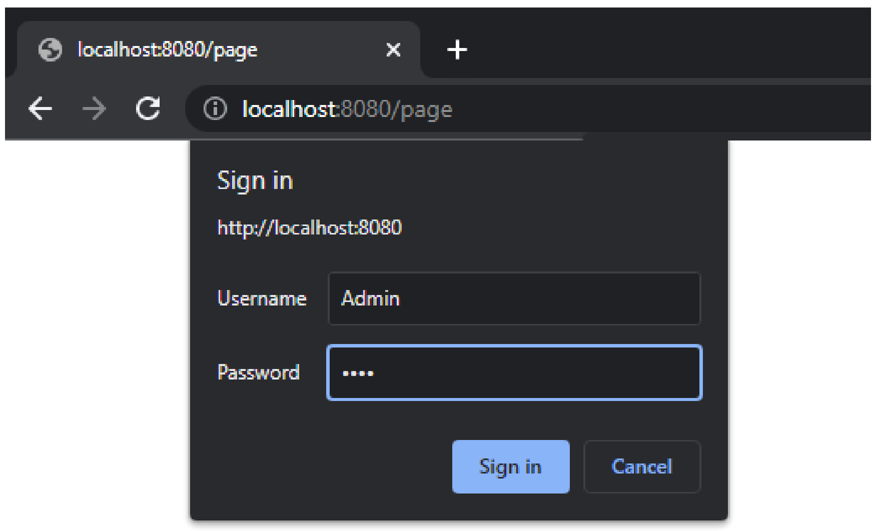
To protect the page, we need to wrap the appropriate handler in the `authenticate` function and pass the authentication name to it:

```
routing {
    authenticate("myAuth") {
        get("/page") {
            call.respondText("Protected content!")
        }
    }
}
```

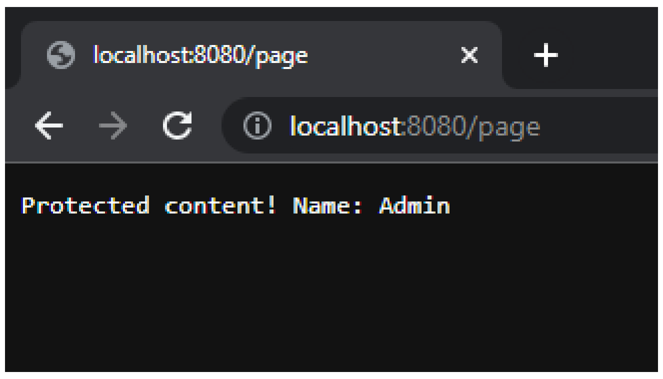
Also inside the handler, we can access the name under which the user is logged in using the `call.principal` function:

```
routing {
    authenticate("myAuth") {
        get("/page") {
            call.respondText("Protected content! Name: ${call.principal<UserIdPrincipal>()?.name}")
        }
    }
}
```

The `/page` can now only be accessed after entering the correct username and password:



If you entered the correct credentials, you will see the page:



```
Protected content! Name: Admin
```

After you log in successfully, the browser saves your username and password. So you don't need to enter them every time you refresh the page.

To protect multiple pages with `"myAuth"` authentication, you just need to place their handlers inside `authenticate("myAuth")`

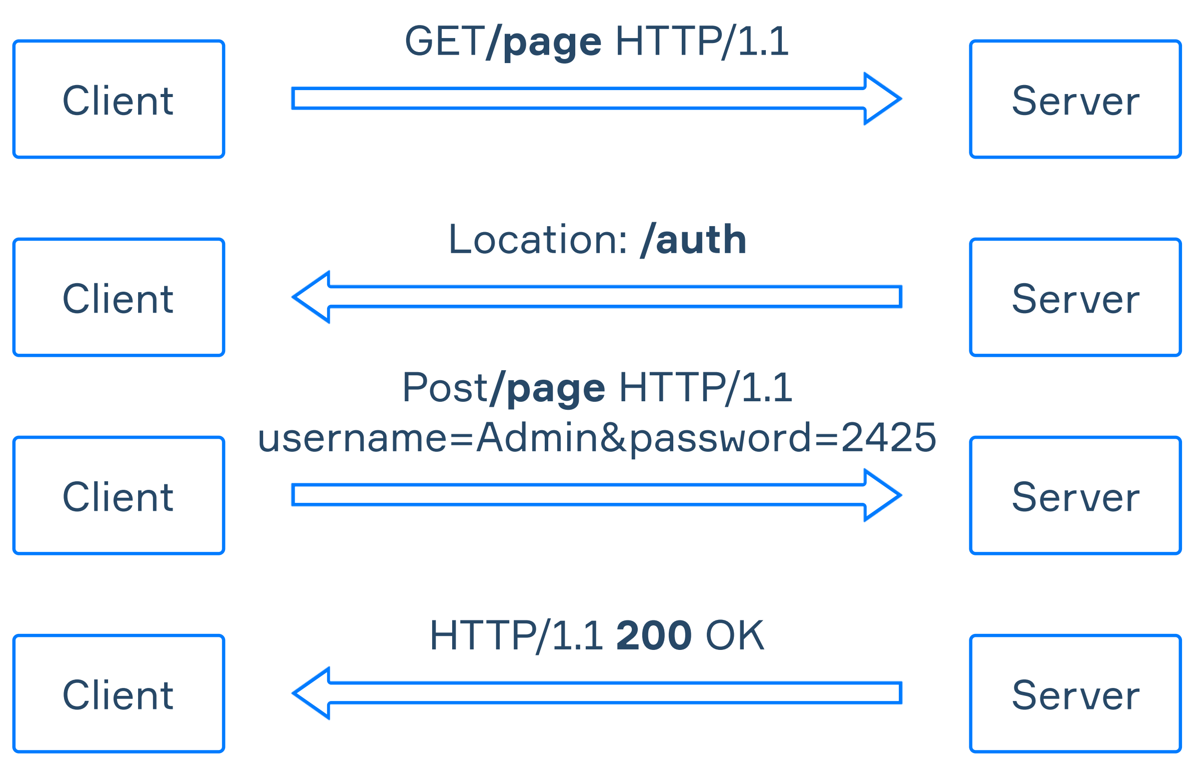
```
routing {
    authenticate("myAuth") {
        get("/page") {
            call.respondText("Protected content!")
        }
        get("/page2") {
            call.respondText("Another protected content!")
        }
        get("/page3") {

            call.respondText("Another protected content2!")

        }

    }

}
```

Table of contents:

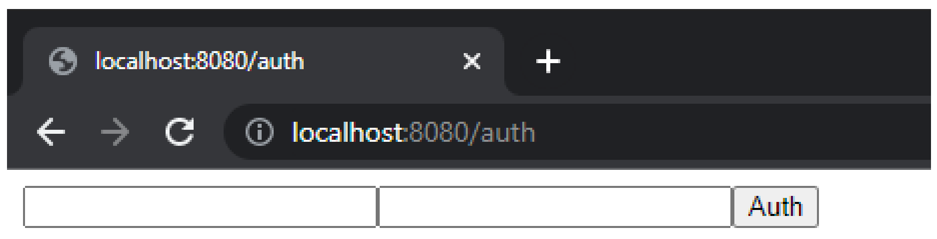
## §3. Form authentication

Now let's look at a more common way to protect your pages, **Form authentication**.

Suppose we have a page ( `localhost:8080/page` ) that is protected by Form authentication. Page `localhost:8080/auth` contains an HTML authorization form with login and password fields. The mechanism of interaction between the client and the server, in this case, will look:



First, the client makes a request to `localhost:8080/page`. The server sees the page is protected by form authentication and redirects the client to the page with the authorization form.



After filling out this form, the user sends a POST request with login and password to `/page`. The server checks the login and password and, if successful, returns the requested page to the user. If authorization fails, the user will be redirected back to the authorization page.

As you can see, Form authentication is very similar to Basic authentication. The advantage of form authorization is that we can change the design of the form on the `/auth` page. You can't change the design of the dialog box in Basic authorization.

## §4. Form authentication in Ktor

Just as with Basic authentication, Ktor has a handy Form authorization plugin.

Make sure the following dependency is in `build.gradle.kts`:

```
implementation("io.ktor:ktor-server-auth:$ktor_version")
```

Now let's install the Form authentication plugin:

```
import io.ktor.server.application.*
import io.ktor.server.auth.*
import io.ktor.server.response.*

fun main() {
    embeddedServer(Netty) {
        install(Authentication) {
            form("myFormAuth") {
                //Configure form authentication

            }

        }

        //...
    }.start()
}
```

To set the form authentication provider, we call the `form` function inside the `install` block. We pass our authorization name `"myFormAuth"` as an argument to the `form` function. We will need this name to refer to the authorization in the Routing handler.

The configuration of this plugin is:

```
install(Authentication) {
    form("myFormAuth") {
        userParamName = "username"
        passwordParamName = "password"
        validate { credentials ->
            if (credentials.name == "Admin" && credentials.password == "2425") {
                UserIdPrincipal(credentials.name)
            } else {
                null
            }
        }
        challenge {
             call.respondRedirect("/auth")
        }
    }
}
```

Inside the `form` block, we have four properties: `userParamName`, `passwordParamName`, `validate` and `challenge`.

`userParamName` and `passwordParamName` contain the names of the parameters that will come in the POST request of the authorization form.

The `validate` function compares the username and password with the correct ones and returns a `UserIdPrincipal`. With this object, we can get in the Routing handler the name of the user who has logged in to our page. In a case of successful authentication or `null` if authentication fails. In our example, this function is the same as it was in basic authentication.

Finally, the `challenge` function sets where to redirect the user if they are not authorized or have entered an invalid login or password.

Now let's connect our authorization to the `/page` handler:

```
authenticate("myFormAuth") {
    post("/page") {
        call.respondText("Protected content! Name: ${call.principal<UserIdPrincipal>()?.name}")
    }
}
get("/page") {
    call.respondRedirect("/auth")
}
```

For users who directly access `/page` using the get method, we also redirect them to the authorization form.

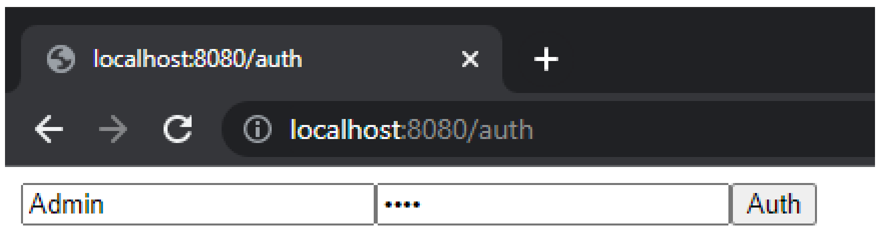
The last thing we need to do is the authorization form itself. We send the appropriate HTML code for the form to the user.

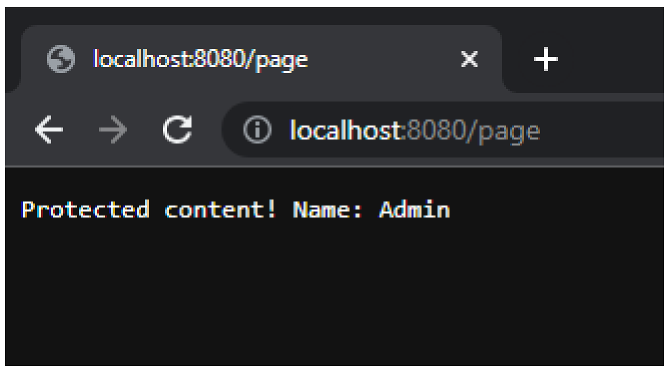
```
get("/auth") {
        val formHtml = """
            <form action="/page" method="post">
                <input type="text" name="username">
                <input type="password" name="password">
                <input type="submit" value="Auth">
            </form>
        """.trimIndent()
        call.respondText(formHtml, ContentType.Text.Html)

}
```

We set the `Content-type: text/html` header so that the browser treats our response as an HTML code, not just text.

Note the action and method attributes of the form. They specify that when the user clicks the button, a post request will be sent to the `/page`. We also set the names of the input fields to match the ones we specified when we initialized the form authentication.

Now you can only access to the `/page` by correctly filling out the authentication form.





```
Protected content! Name: Admin
```

## §5. Conclusion

In this topic, we learned about implementing a simple authorization in Ktor.

We learned about the mechanisms of Basic and Form authorization and their implementation in Ktor

Keep in mind that:

- With Basic authentication, the user enters the data in the dialog box embedded in the browser.
- With Form authentication, the user is redirected to an HTML page with an authorization form.

Now let's put what we've learned into practice.

4 users liked this piece of theory. 0 didn't like it. **What about you?**

Start practicing Verify to skip Provided by: JetBrains Academy

Comments (0) Useful links (0) Show discussion