PRESSIDIUM®

WordPress and Object-Oriented Programming

Pressidium is a technology company that builds powerful, secure and scalable infrastructure for businesses. Our fully managed Enterprise WordPress platform is trusted by Digital Agencies, Media Outlets, Tech startups and Fortune 500 companies. It is considered to be vastly superior to competing solutions on the market today.

From quality of service, workmanship and engineering, to sheer speed, security and site performance, our platform is built to deliver.

WordPress and Object-Oriented Programming

Published by PRESSIDIUM®

TABLE OF CONTENTS

PART ONE

Introduction

PART ONE

Introduction

If you are a WordPress developer it is likely that you write your code in a procedural way. You set some simple ordered steps and follow them to achieve the desired output and solve a problem. A simple example is a WordPress loop like the one shown below:

```php
<?php
if ( have_posts() ) {
      while ( have_posts() ) {
            the_post();
            //
            // Post Content here
            //
      } // end while
} // end if
?>
```

In a loop, you first check whether you have posts that correspond to the running query and if so, you start looping through the posts with the while loop. The same goes for every custom function you will insert in the `functions.php` file.

> **In this document we're going to transition away from procedural programming and look at a style of programming called Object-Oriented Programming (OOP).**

We will examine how this is different from what you've used so far as a programming style and take a look at some of its advantages. But most importantly, **we will explain how object-oriented concepts work in a WordPress environment and the relation between them.**

What we would also like to establish before we get further into this is that **it is important that you learn about the role of object-oriented programming and when it's suitable to be used.** Learning how to build an object-oriented plugin or theme may be suitable for your project but it's important to understand when this is the case. Hopefully, by the end of your reading, you will be able to better understand when and why a certain project may be suitable to be written in object-oriented code.

The limitations in procedural programming

WordPress itself has already pushed you into thinking procedurally. And, though WordPress uses PHP objects all the time, it doesn't follow that WordPress itself is Object-Oriented. **This is a common misinterpretation of what OOP is about and the reason many people believe WordPress is Object-Oriented although it's not.**

It's important to clarify that we're not saying you should refrain from using procedural programming but it is true that **there are limitations in this coding style.** For example, when customizing a theme or plugin, it is likely that you will achieve the desired HTML output by writing code in a procedural way. But what about the scalability and future maintenance?

When building a custom plugin it is very important that you take stock and assess how scalable it will be. For example, you will probably want to be able to grow your plugin by adding further features as the plugin matures. It's at this point where you will start having difficulties organizing your code when you code procedurally.

But even if you're not coding plugins, with procedural programming you always take the risk of reaching a point where you're not sure how everything fits together anymore. As a result, **every small change you make may have unintended consequences.**

> **Object-oriented programming is used to solve more complicated problems. It may share the same goal with procedural programming, but in many cases offers a superior way of working. It allows you to create bold solutions to a problem that are reusable, organized and secure.**

How Object-Oriented Programming Solves Problems

Object-oriented programming (OP) has two important concepts: **Classes and Objects.**

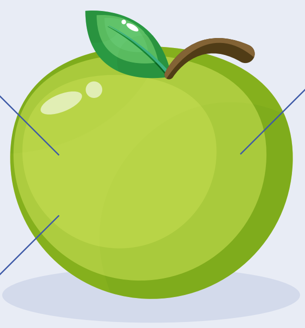
Classes are essentially a template used to make objects. Each object has its own methods and set of properties and the value of those properties may vary.

So before moving forward it is vital to make sure we understand what an Object is. Let's break down a simple sentence as an example:

'I eat green apples'.

Apple is the **Object**

Green is the color of the Object (an **Object property**)



The verb 'eat' is the **method (function)**

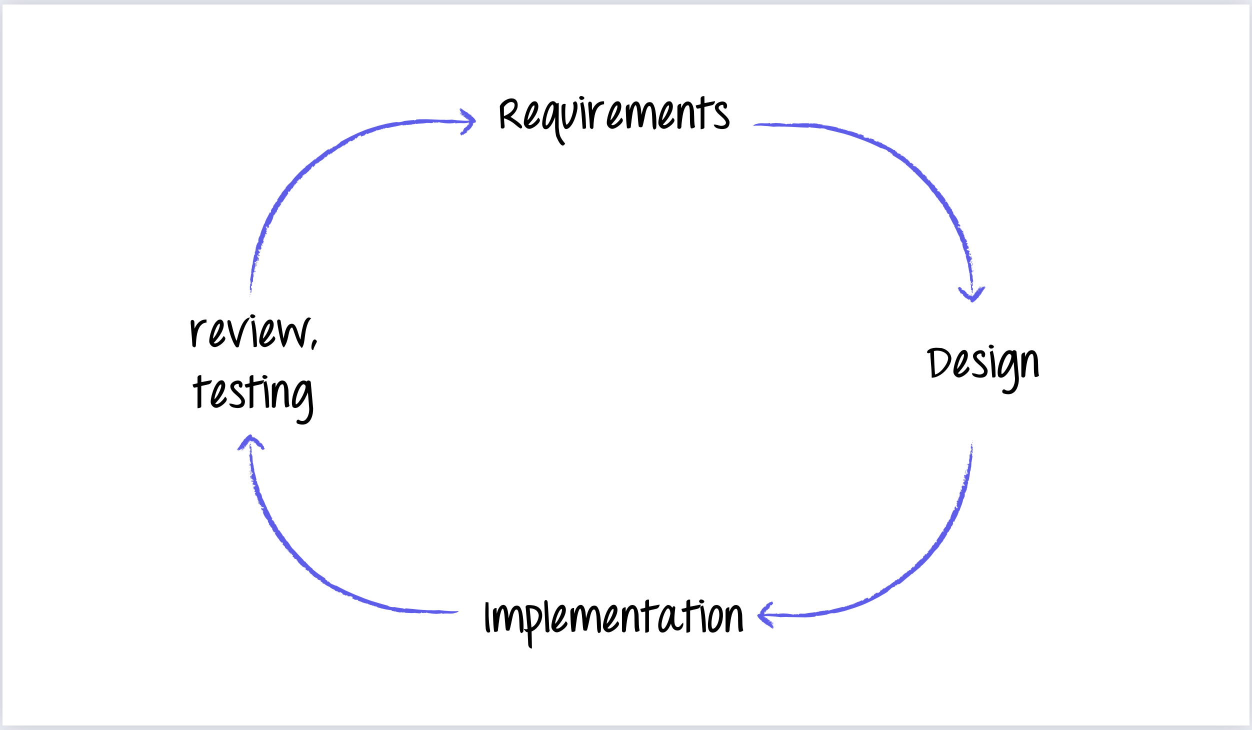
A Class defines the properties and behavior of all objects that use it. Solving problems with OOP is achieved by combining these programming 'bricks' the correct way.

```
Class: Apple -----> Object: Green apple
|
|   color (Property)            |  green
|   eat (method)                |  eat
|                               |
```

Before diving in, an important first step is to take some time to **design your proposed solution.** In essence, what you need to do is:

1. Define the problem and what the plugin should do to solve it
2. Describe the classes and their relationships as well as the interaction between the objects
3. Turn everything into code
4. Review and test the project

While you are at this step, keep in mind that you'll be lucky if you get it perfect at the first attempt! You will need to start over until you get the right concept that will solve the problem. Consider it as part of a creative process that will offer you a set of tools you can later use as a team with no conflicts.



Let's get into the features of Object-Oriented programming and clarify how it works, with the help of a simple metaphor.

The Features of Object-Oriented Programming

Let’s say we have a boiler that we use to boil some water.

When we use the machine, we do it in a certain way, the only way the manufacturer allows us. We cannot override the way the boiler runs when we press the ‘On’ button and, to be honest, neither would we want or need to.

That gives us **two very important advantages.**

First of all, the limited functionality (i.e. to boil some water) means the system is both more robust and more secure. The limited feature set and the provision of just an ‘On/Off’ button mean we, as the user, can’t get anywhere near the mechanism of the boiler. By default, this means there is less to go wrong and the system is more robust as a whole. The simplicity of the operation also makes it much more user friendly with pretty much anyone able to use it.

Keeping this in mind let’s see how this relates to Object-Oriented programming. In theory, OOP has three valuable characteristics:

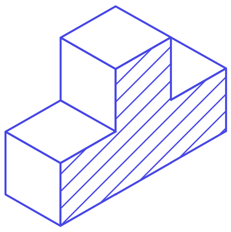
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

What **Encapsulation** does is to group the data and behaviour inside a single entity. Inside the Class, you can define how it will be used (boiler button) and control the visibility by using the Public, Protected or Private labelling on Classes. This is what also makes it a very secure system because it gives you the ability to control who has access. This also means it's easier to maintain and also use.

Inheritance is about helping you reuse code between your classes. One such example is when a Class extends another Class. You might have used this when extending the Walker Class to change the HTML output of a tree-like data like a menu or comments structure. Understanding the relationship between your classes will enable you to create reusable object-oriented code.

Finally **Polymorphism** is what makes all those relationships work together, what defines what your Classes have in common, how to reuse these common elements and whether they are related to anything else.

> **But without some concrete examples it's likely that all of the above is still a little confusing.**



PART TWO

A Real World Example

PART TWO

A Real World Example

Before we proceed with more specific coding examples using OOP, we will try to explain how a real world scenario can be approached with the different mindset required for OOP and how this is analyzed using objects and classes.

A Real Life Scenario: Sending an SMS

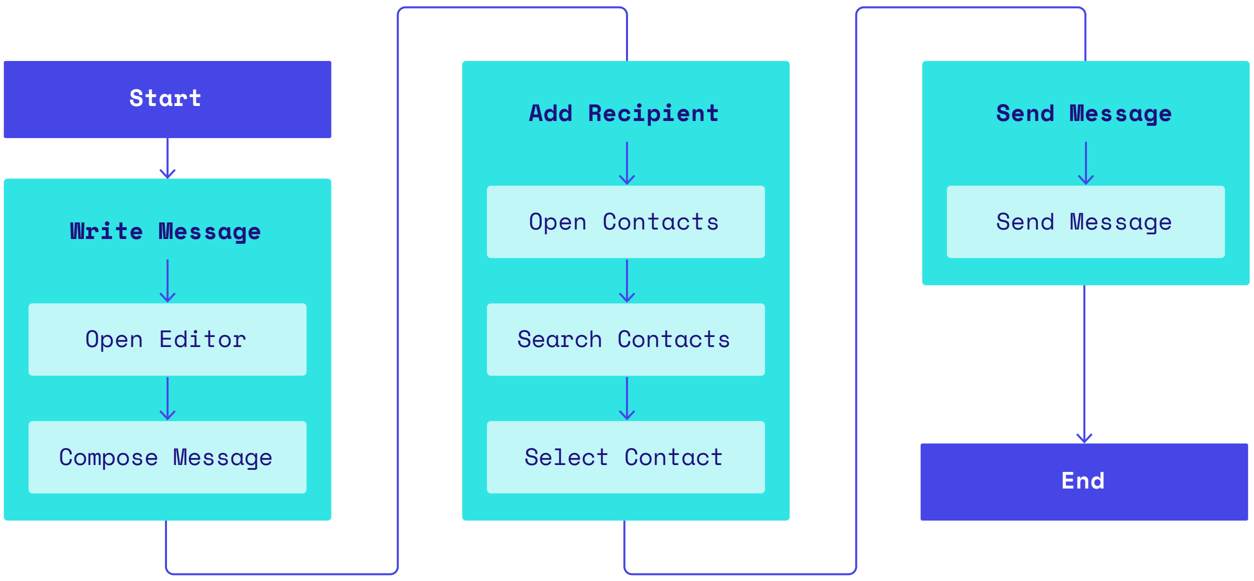
This is more like, a “past” life scenario actually as SMS is used less and less nowadays but as you’ll see, there is a reason we use this as an example!

Suppose you have a mobile device and you want to send a text message to one of your contacts. Keeping the example as simple as possible, the sequence of actions would be:

- preparing the message in the device editor
- selecting the recipient, and finally
- sending the message.

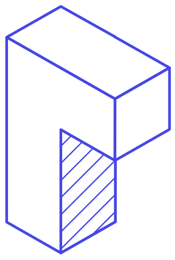
To:
Hello
To:
John
George
Michael
Helen
Daniel
To: John
Hello

So let's try and visualize the steps that you would follow in order to send your message:



We added some more detailed descriptions of the actions but more or less all that you do is **3 basic steps.** You prepare the message in the device editor, you select the recipient from your contacts, and then send the message. And you are done! Your message is now sent.

Now, if we were to represent in code an application that sends an SMS message we should analyze which route is better to follow; the procedural or OOP approach.



The Application with a Procedural Approach

If you're a WordPress plugin developer, you're most likely familiar with ***procedural programming.***

As we described previously, procedural programming is a type of imperative programming, where our programs consist of one or more procedures. So, as a developer you break down your plugin into a bunch of variables that hold your data, and functions that operate on the data.

In our example above with the SMS message, you would perform a series of actions that would lead to the desired result. As you may have already guessed, you would have, for example, a variable that holds the message's text content, a function with a `$contact` parameter that returns the phone number and finally, a function that sends the message. In code it would look like this:

```php
function get_phone_number( $contact ) {
    // Code that finds the contact's number in the list of contacts
    return $phone_number;
}
function send_sms( $contact, $message ) {
    $phone_number = get_phone_number( $contact );
    // Code that sends the message to this number
    print "Message Sent!";
}
```

And you would use it like this:

```
$text = "Hello John";
function send_message( "John Doe", $text );
```

So, you would complete a series of tasks that will lead you to the desired result.

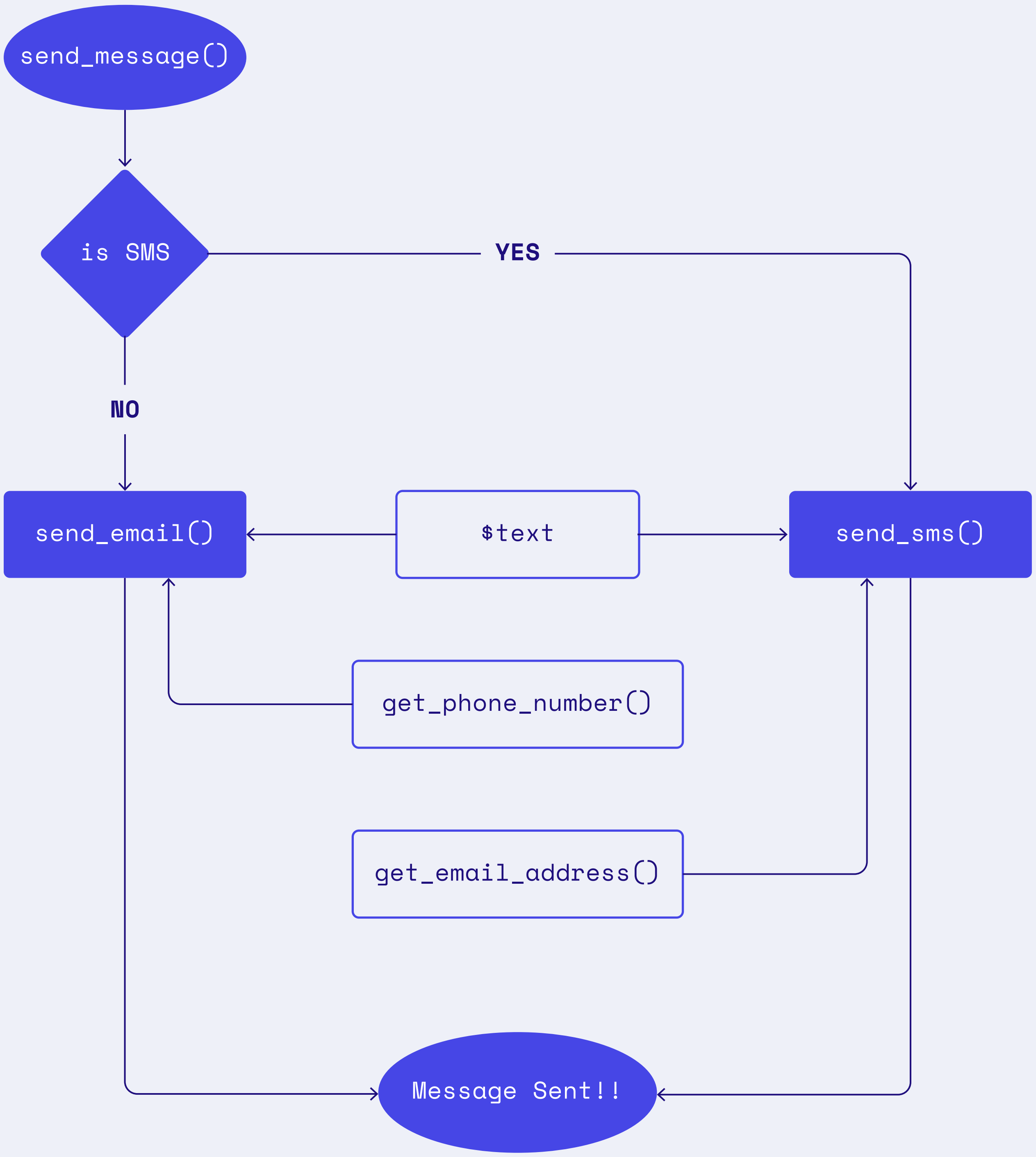
In this very simple example of course, that has limited and very specific requirements, there is no reason to consider using OOP at all. Procedural programming is more than enough to achieve your goal.

However, if you think of some scenarios as to how this application could expand in the future, you might realize that, in the long run, you could have issues in terms of scalability. We will try and explain why below.

Expanding the Application with Procedural Approach

Let’s say that you want to improve this application and provide the ability to send other kinds of messages as well, like an email for example. The function that delivers the message would be different in each case.

When sending an email, you need the contact's email address, not the phone number. Apart from this, we will need to add a parameter in the final `send_message()` function that will correspond to the type of technology we use; email or SMS.



The corresponding code could look something like this:

```
function get_phone_number( $contact ) {
    // Code that finds the contact's number
    return $phone_number;
}
function get_email_address( $contact ) {
    // Code that finds the contact's email address
    return $email_address;
}

function send_sms( $contact, $message ) {
    $phone_number = get_phone_number( $contact );
    // Code that sends the message to this number
    print "SMS Sent!";
}

function send_email( $contact, $message ) {
    $email_address = get_email_address( $contact );
    // Code that sends the email to this number
    print "Email Sent!";
}

function send_message( $contact, $message, $technology ) {
    if ( $technology == "SMS" ) {
        send_sms( $phone_number, $message );
    } else if ( $technology == "Email" ) {
        send_email( $email_address, $message );
    }
}
```

So, it is not like this could not be implemented with a procedural approach. But if you are an experienced developer you've probably already understood how this could become messy in the future.

The Drawbacks with a Procedural Approach

What if we had multiple types of messages? The `if` statements would become annoyingly large. And, most importantly, what if you had functions that use the `send_message()` function? In that case, you would need to add the `$technology` parameter in all those functions as well.

As your code grows, functions will be all over the place meaning you will be starting to copy/paste chunks of code (never desirable), and making a small change to a function might break several other functions. We've all been there. You would want to avoid this and be able to easily add features to your code without interfering in the structure too much.

> Object-oriented programming (or OOP) is a programming paradigm that attempts to solve this issue by allowing us to structure our plugin into small, reusable pieces of code, called **classes.** As we described in our introduction article, a class is basically a template that we use to create individual instances of the class, called **objects.**
>
> An object contains data and code. We still have variables that can store information, called **properties.** And procedures that operate on the data, called **methods.**

The Application with an OOP Approach

Now let's analyze the same scenario as above with an OOP approach.

First, we will define what objects we have here, what characteristics each has and what actions they perform. The characteristics are what later will be our properties and actions will be our functions or methods as they are called in OOP.

Let's think about what we have in the first scenario of sending an SMS in the simplest way possible. There is a device which has a layout that we use to send the SMS message. We have the message content, we choose a contact as a recipient and finally the message.

```php
<?php
interface MessagingCapable {
    public function send_message( $contact, $message );
}

class Phone implements MessagingCapable {
    public function send_message( $contact, $message ) {
        print "You sent " . $message;
    }
}

function say_hi( MessagingCapable $device, $contact, $message ) {
    $device->send_message( $contact, $message );
}
```

We declare the `Phone` class which implements the `MessagingCapable` interface. So we have to implement all the methods declared in it. The `say_hi()` function requires 3 parameters:

- A device that supports messaging
- A contact
- The message

In order to actually send a message we use this function like this:

```
$phone = new Phone();
say_hi( $phone, "John Doe", "Hello John" );
```

We are basically creating an object by instantiating the Phone class and passing the contact and message content. This would output:

```
You sent "Hello John"
```

> We demonstrated this simple scenario of sending a text message by using classes. In the next section, we will see how we can expand the application's capabilities following the OOP approach and while scaling up, we will examine where the OOP features play their role as well as the benefits of using this technique.

Expanding the Application with the OOP approach

Let’s add the ability to send emails as well, like we did before procedurally.

Regardless of the device, we ideally would want to use the `say_hi()` function the same way. Take a look at code below:

```php
<?php
interface MessagingCapable {
    public function send_message( $contact, $message );
}

class Phone implements MessagingCapable {
    public function send_message( $contact, $message ) {
        print "You sent "' . $message . '" SMS to ' . $contact;
    }
}

class Computer implements MessagingCapable {
    public function send_message( $contact, $message ) {
        print "You sent "' . $message . '" email to ' . $contact;
    }
}

function say_hi( MessagingCapable $device, $contact, $message ) {
    $device->send_message( $contact, $message );
}
```

When we use this piece of code, we would pick up the mobile device to send an SMS and the computer to send an email. We would either:

```
say_hi ( new Phone(), "John Doe", "Hello John" );
```

or:

```
say_hi ( new Computer(), "John Doe", "Hello John" );
```

that would output `You sent a "Hello John" SMS to John Doe` and `You sent a "Hello John" email to John Doe` correspondingly.

Here we already start to detect some OOP features. We introduced interfaces by using the `MessagingCapable` interface.

An interface declares a set of methods that must be implemented by the class without defining how these methods are implemented. All methods declared in an interface must be public.

PHP doesn’t support multiple inheritance, meaning a class cannot inherit the properties/methods of multiple parent classes.

Using a Phone to send a message will be different from using a Computer. Instances of different classes act differently when asked to perform the same action (i.e. `send_message()`).

> This is an example of **Polymorphism.** If we later create a new device, we won't need to modify our code to accommodate it, as long as they all share the same interface.

We would also like to point out here that we already see the difference in readability as well. The way we finally use this script by just coding:

```
say_hi( new Computer(), "John", "Hi" );
```

This is totally straightforward to any developer that works on the project. And of course, the more complex the plugin, it becomes more obvious how helpful this is, especially when working in a team.

To try and explain better how easy it is to expand your plugin in Object-Oriented Programming let's try adding some more functionality.

Adding More Functionality

If we want to add the ability to browse the internet, we would just add an extra interface for any device that could respond to this ability, like a computer for example.

```
interface InternetBrowsingCapable {
    public function visit_website( $url );
}
```

The implementation of this interface will be coded like this:

```
class Computer implements MessagingCapable, InternetBrowsingCapable {
    public function send_message( $contact, $message ) {
        print 'You sent a "' . $message . '" email to ' . $contact;
    }
    public function visit_website( $url ) {
        print 'You visited "' . $url . '"';
    }
}
```

Here you can see how we can implement multiple interfaces.

So in the current Computer class we just added the extra interface to be implemented, since a computer can send a message and browse the internet, and the visit_website($url) method.

NOTE: Of course, since visiting a url is totally irrelevant with the say_hi() function we will also introduce a new function, something like:

```
function visit_url( InternetBrowsingCapable $device, $url ) {
    $device->visit_website( $url );
}
```

And that's it! For any device that can visit a URL we can use this function like we did with computer. There are no worries that you will break the rest of functionality. This shows the scalability available when using OOP compared to procedural programming.

Let's add a smartphone device just to demonstrate some more features. Here is the whole code, with the smartphone class addition so that you can have a better picture of what's going on:

```
<?php
interface MessagingCapable {
    public function send_message( $contact, $message );
}

interface InternetBrowsingCapable {
    public function visit_website( $url );
}

class Phone implements MessagingCapable   {
    public function send_message( $contact, $message ) {
        print 'You sent a "' . $message . '" SMS to ' . $contact;
    }
}

class Computer implements MessagingCapable, InternetBrowsingCapable {
    public function send_message( $contact, $message ) {
        print 'You sent a "' . $message . '" email to ' . $contact;
    }
    public function visit_website( $url ) {
        print 'You visited "' . $url . '"';
    }
}
```

```
class Smartphone extends Phone implements InternetBrowsingCapable {
    public function visit_website( $url ) {
        print 'You visited "' . $url . '"';
    }
    public function send_message( $contact, $message ) {
        parent::send_message( $contact, $message );
        print ' from your smartphone';
    }
}

function say_hi( MessagingCapable $device, $contact, $message ) {
    $device->send_message( $contact, $message );
}

function visit_url( InternetBrowsingCapable $device, $url ) {
    $device->visit_website( $url );
}
```

The Smartphone class extends the Phone parent class and implements the `InternetBrowsingCapable` interface. That means it can send a message and visit a URL. Here, we detect the Inheritance feature. In other words, we have a hierarchy of classes, a parent class (Phone) and a subclass (Smartphone).

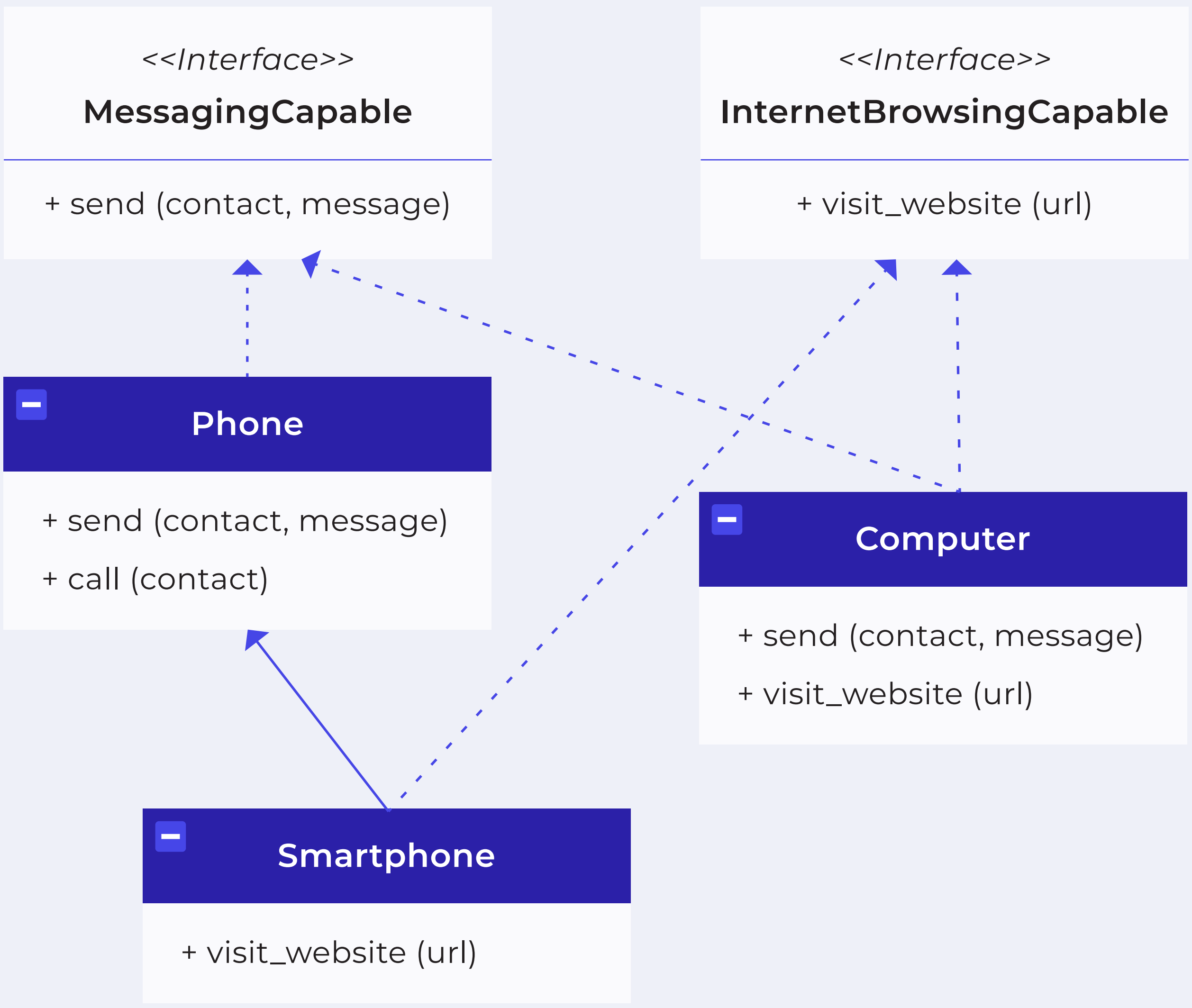
So a Smartphone object inherits all the properties and behaviors of the parent Phone class. That way, inside the child class we can add a method or override a method of the parent class, like we did with the `send_message()` in the Smartphone class. We did this to change the output. We could totally ignore this method and use the send_message() of the parent class as it is.

You can try the code yourself by pasting it in the code block to this great PHP online tool. Under the code, try any of these code lines and see the different results.

```
say_hi ( new Phone(), "John Doe", "Hello John" );
say_hi ( new Computer(), "John Doe", "Hello John" );
say_hi ( new Smartphone(), "John Doe", "Hello John" );
visit_url ( new Smartphone(), "https://www.pressidium.com" );
visit_url ( new Computer(), "https://www.pressidium.com" );
```

For an even better understanding of the whole concept take a look at the Class diagram of the above code.

GUIDE		
public (+) protected (#)	private (-) abstract (italic)	inheritance - - - - - - - - ▶ implementing an interface ————▶



As depicted above, when designing the relationships between classes, we do not include the common elements in the child class. Further-more, do not forget to pay attention in the guide on the left so you can identify the relationships and the visibility of their properties and methods.

If you would like to see the Encapsulation feature in action as well, try and include a Contact class in any of the above example scripts we pro-vided. The class would look like this:

```
class Contact {

    private $name;
    private $phone_number;
    private $email_address;

    public function __construct( $name, $phone_number, $email_address ) {
        $this->name          = $name;
        $this->phone_number  = $phone_number;
        $this->email_address = $email_address;
    }


    public function get_name() {
        return $this->name;
    }

    public function get_phone_number() {
        return $this->phone_number;
    }

    public function get_email_address() {
        return $this->email_address;
    }

}
```

The `__construct()` method, by design, is called automatically upon the creation of an object. Now when we instantiate the Contact class, its constructor gets called and sets the values of its private properties. Then we use our “getters” that are the `get_name()` , `get_phone_number()` and `get_email_address()` public methods to retrieve these values.

> **Encapsulation** is bundling the data with the methods that operate on the data while restricting direct access preventing exposure of hidden implementation details.

What we've covered and what's next

Hopefully at this point you are more familiar with Object-Oriented programming in a more practical way. OOP really helps make it easier for the application to expand in the future if necessary by being clear and reusable.

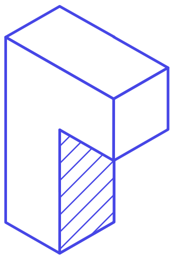
Security is also improved because of encapsulation. In procedural programming on the other hand, encapsulation is not emphasized. All data is often global which means access is available from anywhere.

As a result of the above, **code maintenance, productivity, scalability and troubleshooting also become much easier for you and your team.**

Moving forward, we will see this programming style in action by applying it to a WordPress plugin. Specifically, we will create a copy of the Limit Login Attempts plugin version 1.7.1 created by Johan Eenfeldt but converted with an Object-Oriented approach as much as possible.

During this process, we'll break down the plugin flow and set the requirements. Going forward, we'll try out our first thoughts on the plugin's design and, in the implementation step, we'll write the code. During the implementation process we'll make some back'n'forths and redesign, if necessary, in order to get the desired results.

We'll not get into details on all parts of the code though. Instead, we would like to focus on sharing the way plugins are built the Object-Oriented way. We are confident that, once you've finished reading, you can very well create an OOP plugin of your own.



PART THREE

A WordPress Example – Defining the Scope

PART THREE

A WordPress Example – Defining the Scope

Our first goal is to define its requirements and, since we're rewriting an existing plugin, we'll need to break its current version down and fully understand how it works.

Our example, like we said, will be based on the Limit Login Attempts plugin by Johan Eenfeldt.

We'll thoroughly examine version 1.7.1 of the "Limit Login Attempts" open source WordPress plugin, which was last updated 9 years before the time this e-book is written. Then, we'll rewrite it following an object-oriented approach.

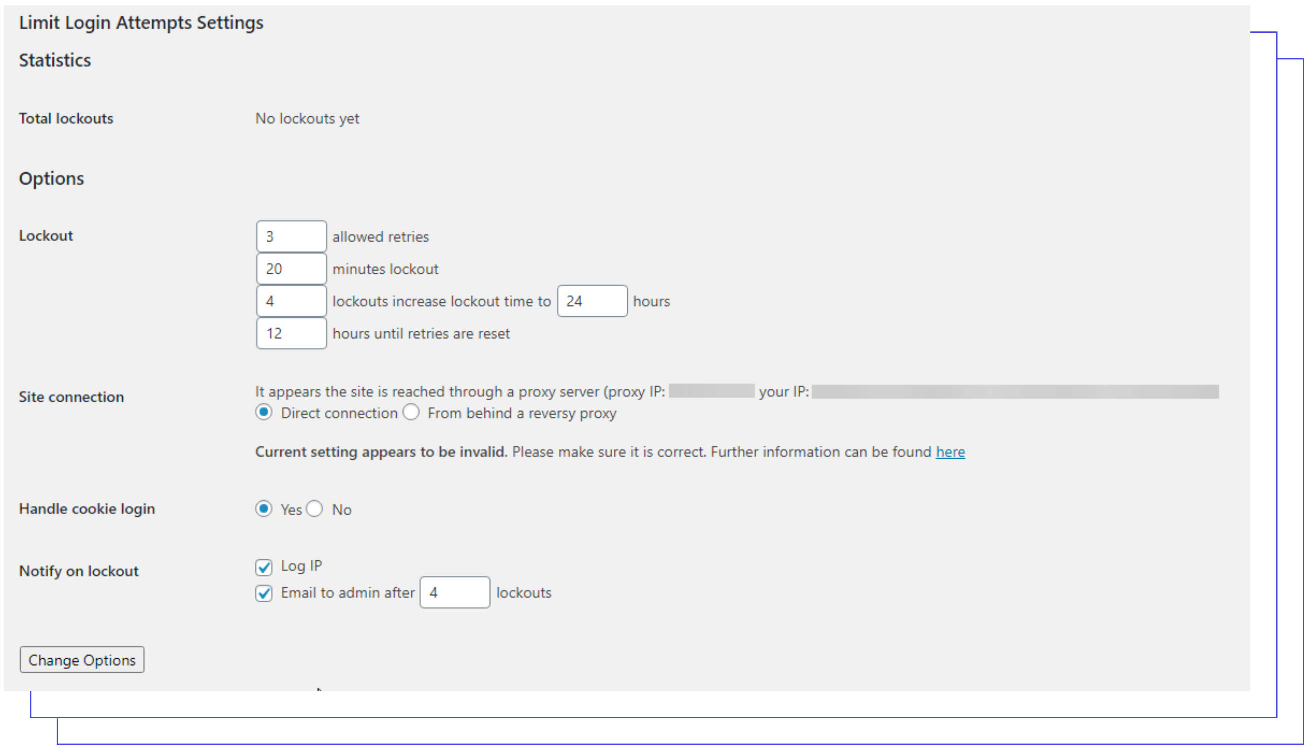
Now, without any further ado, let's dive into this! We'll take a closer look at the current version of the plugin.

Getting Started

If you want to follow along, you can download, install and activate version 1.7.1 of the plugin.

NOTE: If you're a Pressidium customer, you can immediately go to wp-admin → Settings → Limit Login Attempts, as we use an adapted built-in version in our installations.

This is what the admin page of the plugin looks like:



Limit Login Attempts Settings

Statistics

Total lockouts: No lockouts yet

Options

Lockout:
3 allowed retries
20 minutes lockout
4 lockouts increase lockout time to 24 hours
12 hours until retries are reset

Site connection: It appears the site is reached through a proxy server (proxy IP: [redacted] your IP: [redacted]
[✓ Direct connection] From behind a reversy proxy

Current setting appears to be invalid. Please make sure it is correct. Further information can be found here

Handle cookie login: [✓ Yes] No

Notify on lockout:
✓ Log IP
✓ Email to admin after 4 lockouts

The current version we use as a starting point already provides some great functionality.

- ✓ Limits the number of retries when logging in (for each IP address), also providing a customizable limit.
- ✓ Limits the number of attempts to log in using auth cookies in the same way.
- ✓ Informs the user about their remaining retries or lockout time on the login page.
- ✓ Notifies administrators when a user gets locked out.
- ✓ Supports both direct and reverse proxy connections.
- ✓ Offers a WordPress filter to whitelist IP addresses.

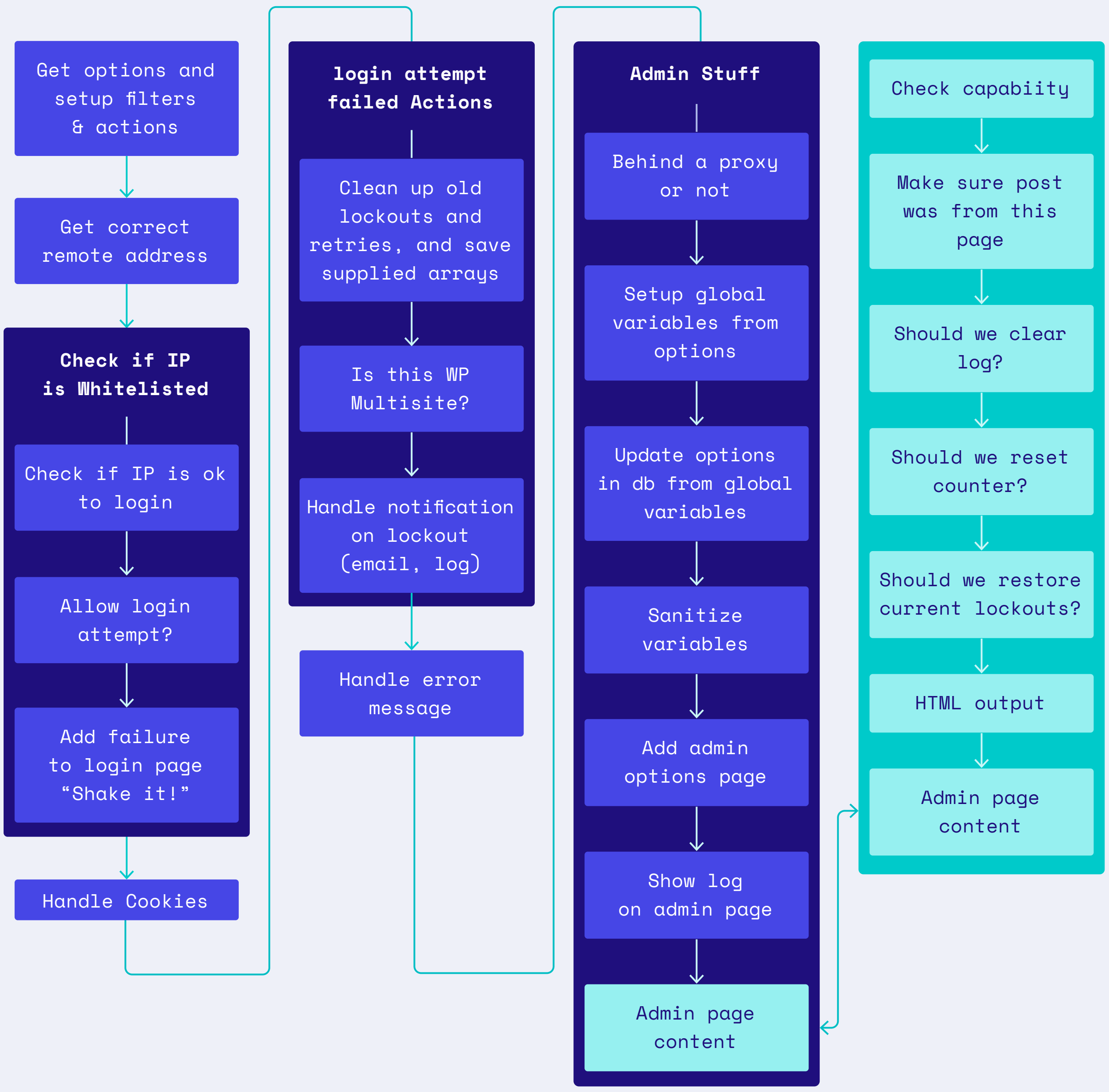
Now we already know what the plugin is supposed to do and what features it offers, let's take a closer look into its flow.

Breaking it down

You don't have to know much about the current version of Limit Login Attempts to read the rest of this content. However, we'll briefly explain what it does, so if you want to take a look at the GitHub repository of the object-oriented version, it will hopefully be easier to understand what's going on.

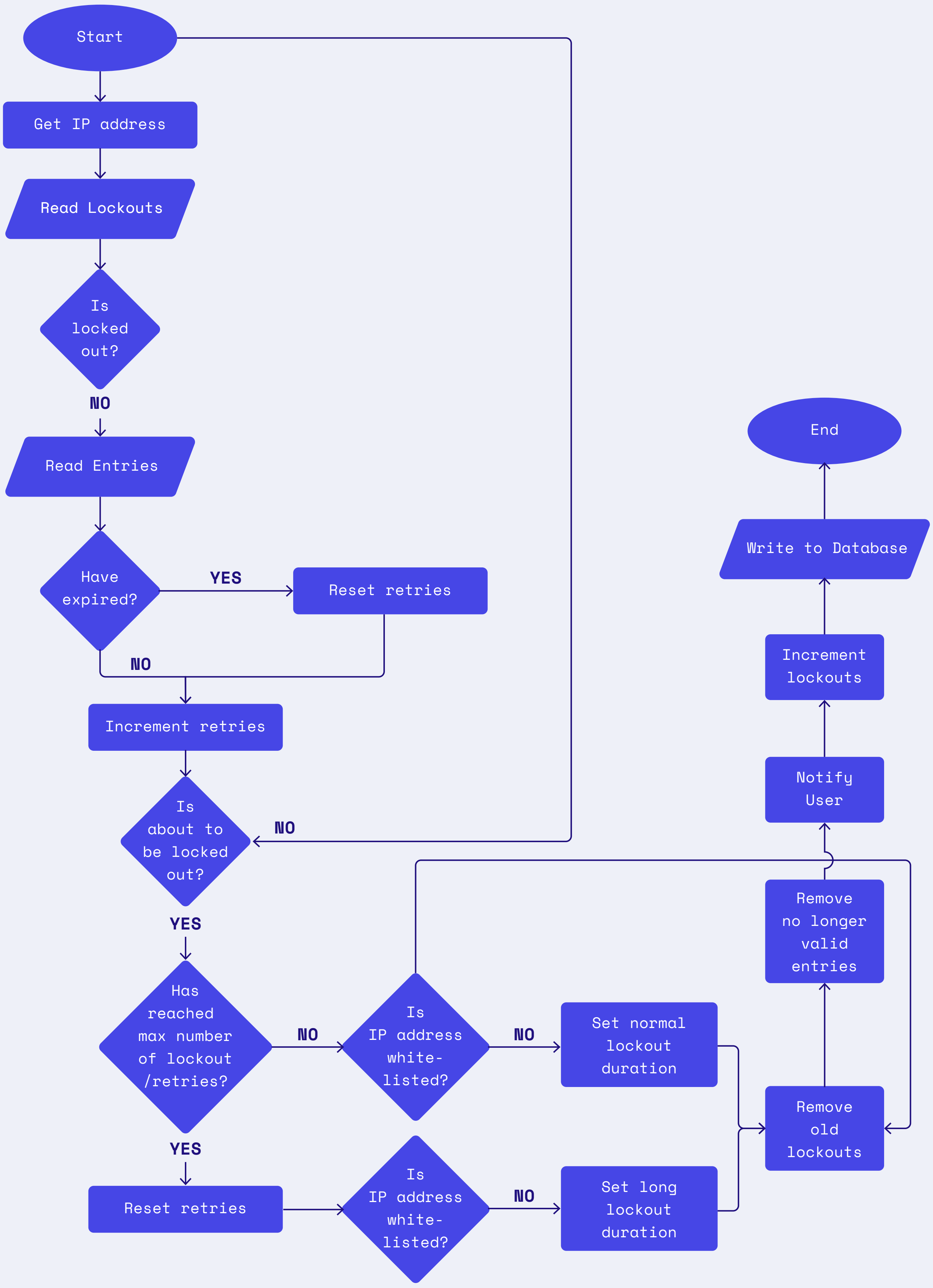
Flow

Studying the main plugin file, almost line by line, we thought a simplified diagram, grouping similar functionality together, might help you wrap your head around the plugin's code flow.


Get options and setup filters & actions
Get correct remote address
Check if IP is Whitelisted
Check if IP is ok to login
Allow login attempt?
Add failure to login page "Shake it!"
Handle Cookies
login attempt failed Actions
Clean up old lockouts and retries, and save supplied arrays
Is this WP Multisite?
Handle notification on lockout (email, log)
Handle error message
Admin Stuff
Behind a proxy or not
Setup global variables from options
Update options in db from global variables
Sanitize variables
Add admin options page
Show log on admin page
Admin page content
Check capabiity
Make sure post was from this page
Should we clear log?
Should we reset counter?
Should we restore current lockouts?
HTML output
Admin page content

Basically, the plugin handles login attempts, determines whether a user can be authenticated, keeps track of failed attempts, locks IP addresses out if necessary, and provides an administration menu.

Here's a flow chart showing how the plugin handles a login attempt:

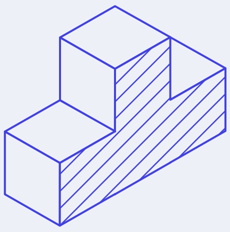

Start
Get IP address
Read Lockouts
Is locked out?
NO
Read Entries
Have expired?
YES
Reset retries
NO
Increment retries
Is about to be locked out?
NO
YES
Has reached max number of lockout /retries?
NO
Is IP address white-listed?
NO
Set normal lockout duration
YES
Reset retries
Is IP address white-listed?
NO
Set long lockout duration
Remove old lockouts
Remove no longer valid entries
Notify User
Increment lockouts
Write to Database
End

This is made possible by hooking into various parts of WordPress during the authentication of a user.

Hooks

Upon further inspection of the plugin's code, we listed all of the actions and filters it hooks into:

- `wp_login_failed`, `login_head`, `login_errors`, `wp_authenticate` and `admin_menu` actions
- `wp_authenticate_user`, `shake_error_codes` and `wp_authenticate` filters



Configuration Options

As we've already mentioned, the plugin also provides an administration menu to allow you to configure its options.

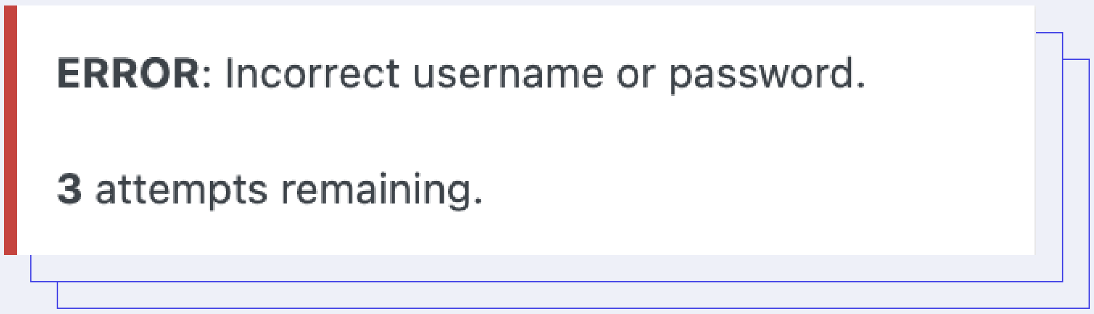
These options are:

- ✓ Client type: Whether the site can be reached directly or through a proxy server (defaults to direct)
- ✓ Allowed retries: Lockout user after that many retries (defaults to 4 retries)
- ✓ Lockout duration: Lockout user for that many minutes (defaults to 20 minutes)
- ✓ Allowed lockouts: Allow that many lockouts before increasing the lockout duration (defaults to 4 lockouts)
- ✓ Long duration: Lockout user for that many hours when the lockout duration is increased (defaults to 24 hours)
- ✓ Valid duration: Reset the failed login attempts after that many hours (defaults to 12 hours)
- ✓ Cookies: Whether to limit malformed/forged cookies (enabled by default)
- ✓ Lockout notify: Notify administrator on lockout (logging, sending an email, both, or none)
- ✓ Notify email after: If notify by email is selected, send an email after that many lockouts (defaults to 4 lockouts)

Error messages

The plugin also customizes the error messages that will be displayed when a login fails.

If a user isn't currently locked out, it will display the remaining number of allowed retries. Otherwise, if a user is already locked out, it will display the time remaining till their lockout gets lifted.



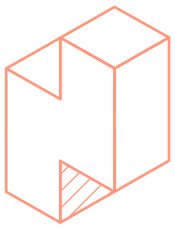
ERROR: Incorrect username or password.

3 attempts remaining.

It also omits any information indicating the existence of the provided username which is intentional in order to prevent username enumeration.

NOTE: Avoiding inconsistent error messages that might accidentally tip off an attacker whilst and ensuring that they only contain minimal details is important to help maintain site security. Otherwise, a potential attacker might be able to discover (or confirm) valid usernames by trying different values until the wrong password error message is displayed. You can read more about that at "CWE-204: Observable Response Discrepancy".

Next up, we'll discuss how we came up with the design of our new and improved plugin in order to provide the exact same features the current one does, while taking advantage of what OOP has to offer. That is, our code will (hopefully) be reusable, extensible and readable.



PART FOUR

Design

PART FOUR

Design

Now that we've got clearly defined the requirements, it is time to start thinking about the design of our new and improved plugin!

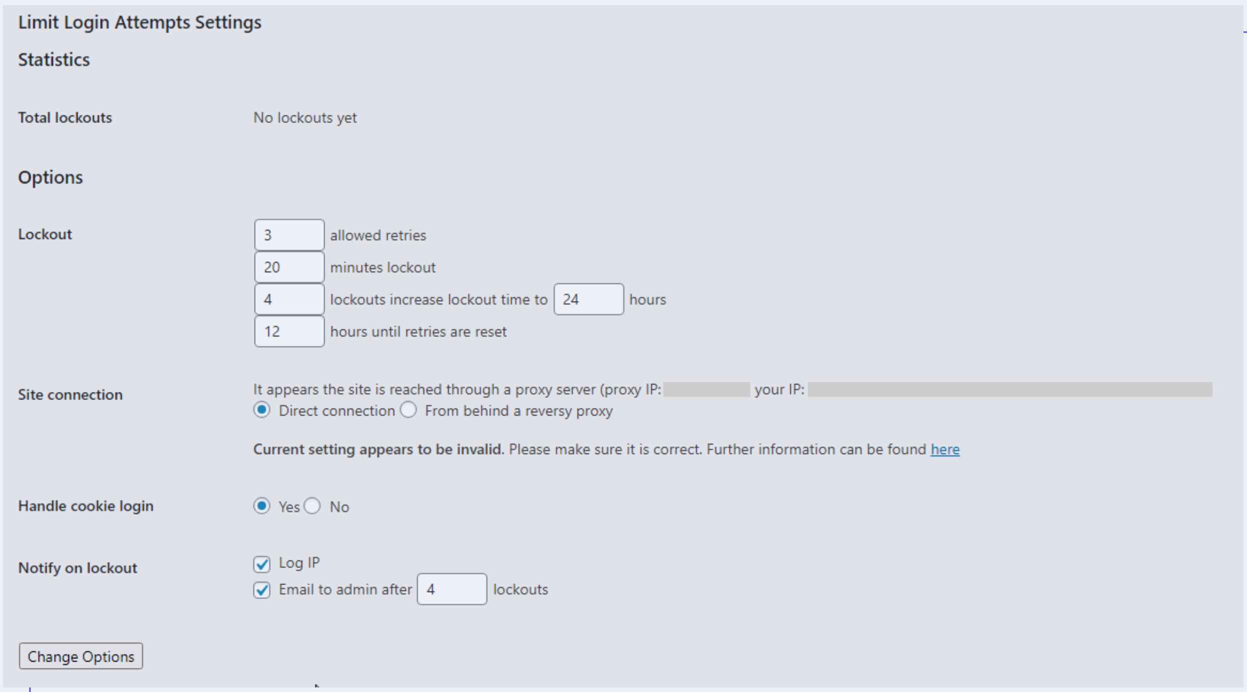
We'd like to remind you that this step may take you a long time when you try it out on your own projects. You probably won't get everything right the first time either. Odds are you'll come up with a design, start implementing it, and then realize that you need to go back and rethink your approach.

It is totally worth the effort though so take as much time as necessary to get everything just right. A well-structured project will make it easier to maintain and extend it, and even reuse its code in other projects so in the long run it's a good use of time.

Having said that, we will focus on some key parts of the plugin and discuss how we came up with our design.

Dissecting the Settings Page

Let’s take a closer look at the admin page of the plugin.



Limit Login Attempts Settings

Statistics

Total lockouts	No lockouts yet

Options

Lockout	3 allowed retries 20 minutes lockout 4 lockouts increase lockout time to 24 hours 12 hours until retries are reset
Site connection	It appears the site is reached through a proxy server (proxy IP: [redacted] your IP: [redacted] [✓ Direct connection] From behind a reversy proxy **Current setting appears to be invalid.** Please make sure it is correct. Further information can be found here
Handle cookie login	[✓ Yes] No
Notify on lockout	[✓ Log IP] [✓ Email to admin after] 4 lockouts

Change Options

You will notice that there is a title (“Limit Login Attempts Settings”), several sections containing some fields, and a “Change Options” button at the bottom of the page.

Each section consists of a title, like “Statistics”, and **a few fields.**

Each field has a title on the left, and the rest of its content on the right side. There are text fields, radio buttons and checkboxes and, some of them, like “Total Lockouts”, only display information and cannot be directly modified by the admin user.

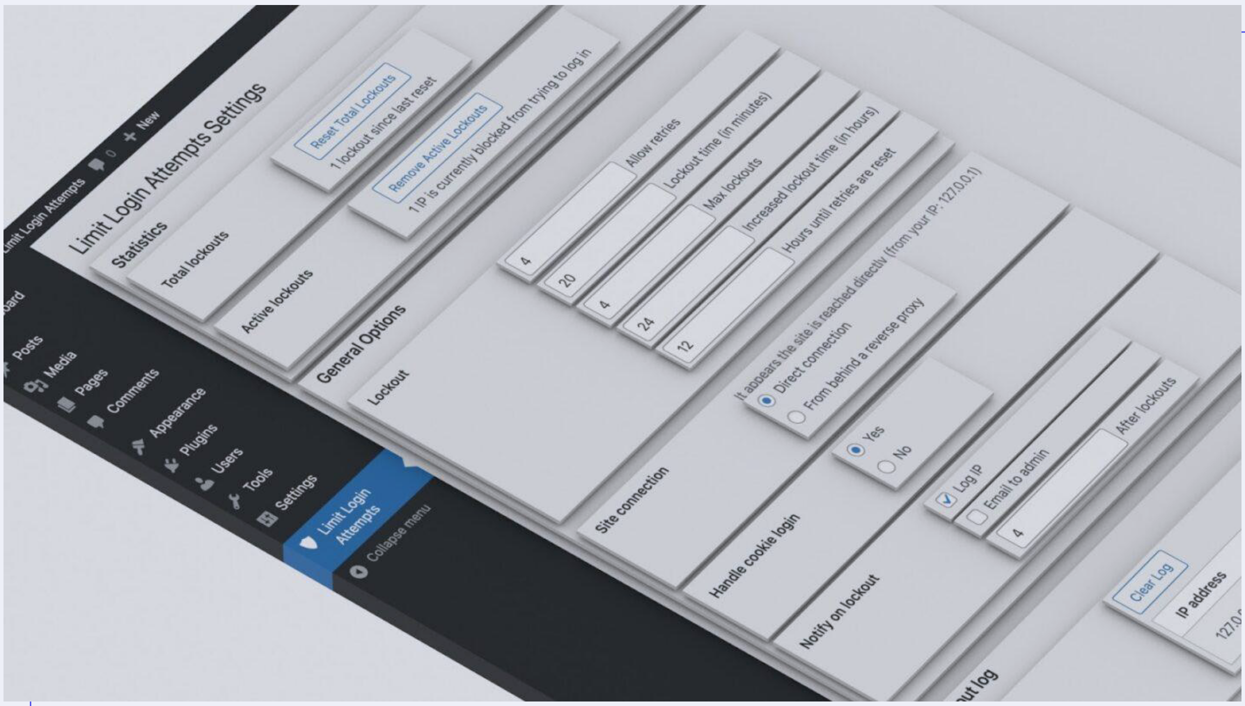
Some fields also include a description, like the “Site Connection” field, but not all of them.

The WordPress Settings API allows us to register settings pages, sections within those pages and fields within the sections:

Pages → Sections → Fields

We thought, why not add one more “layer”, the Elements, in order to make our plugin easier to extend in the future.

Pages → Sections → Fields → Elements



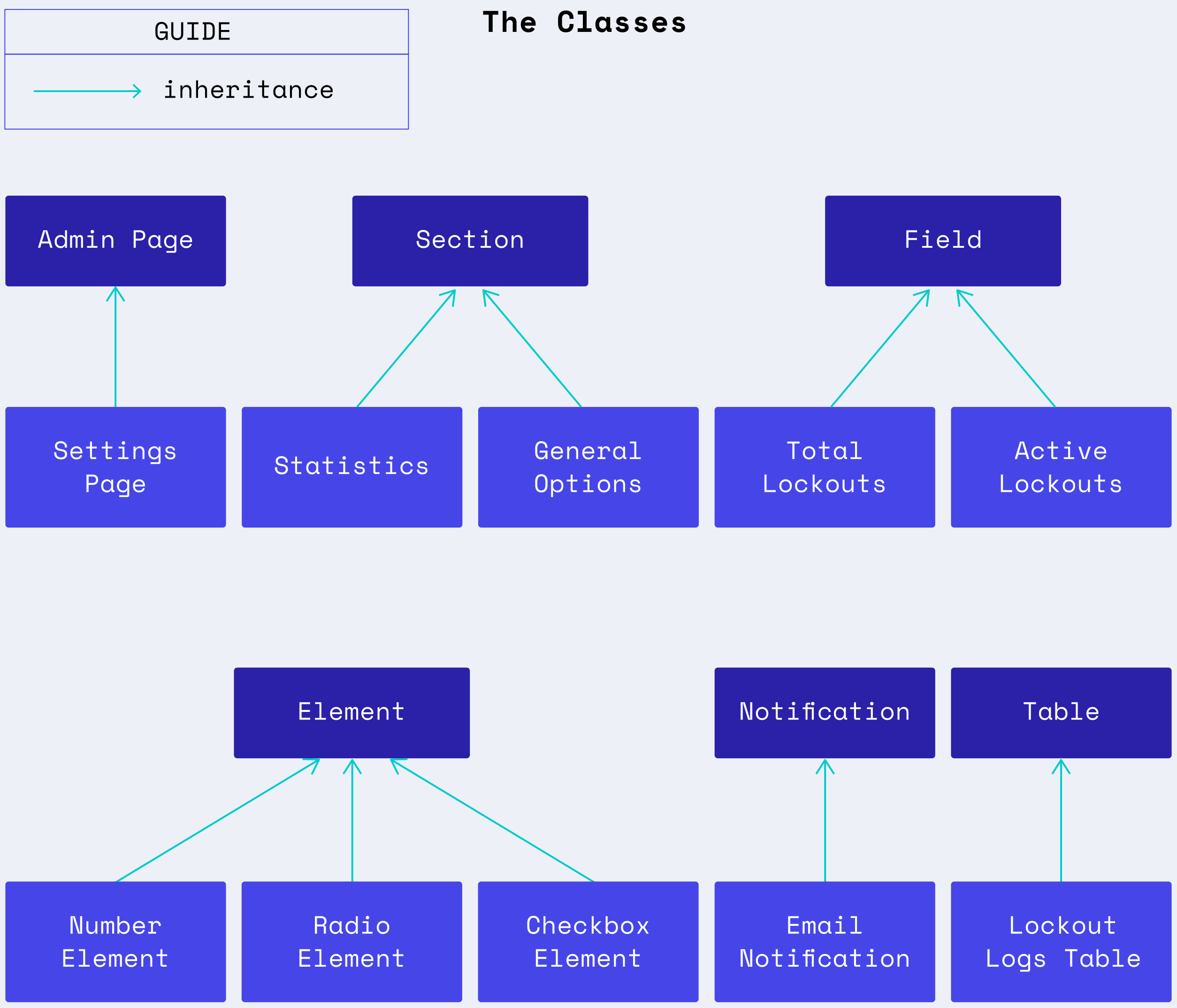
So, Pages and Sections are what we’ve already explained above, and Fields will contain Elements of any content type on the right side.

Taking under consideration all these different kinds of elements, we went with an Element class and several classes, extending it, for the checkboxes, radio buttons, numbers etc. that will render different output.

We may also need to add more pages and sections in the future. So it’s likely that we’d need to extend these admin page and section classes.

The same goes for the fields. The classes for “Total lockouts”, “Active lockouts” etc. will extend the same (parent) class.

Here’s a simplified visual that demonstrates those relations:



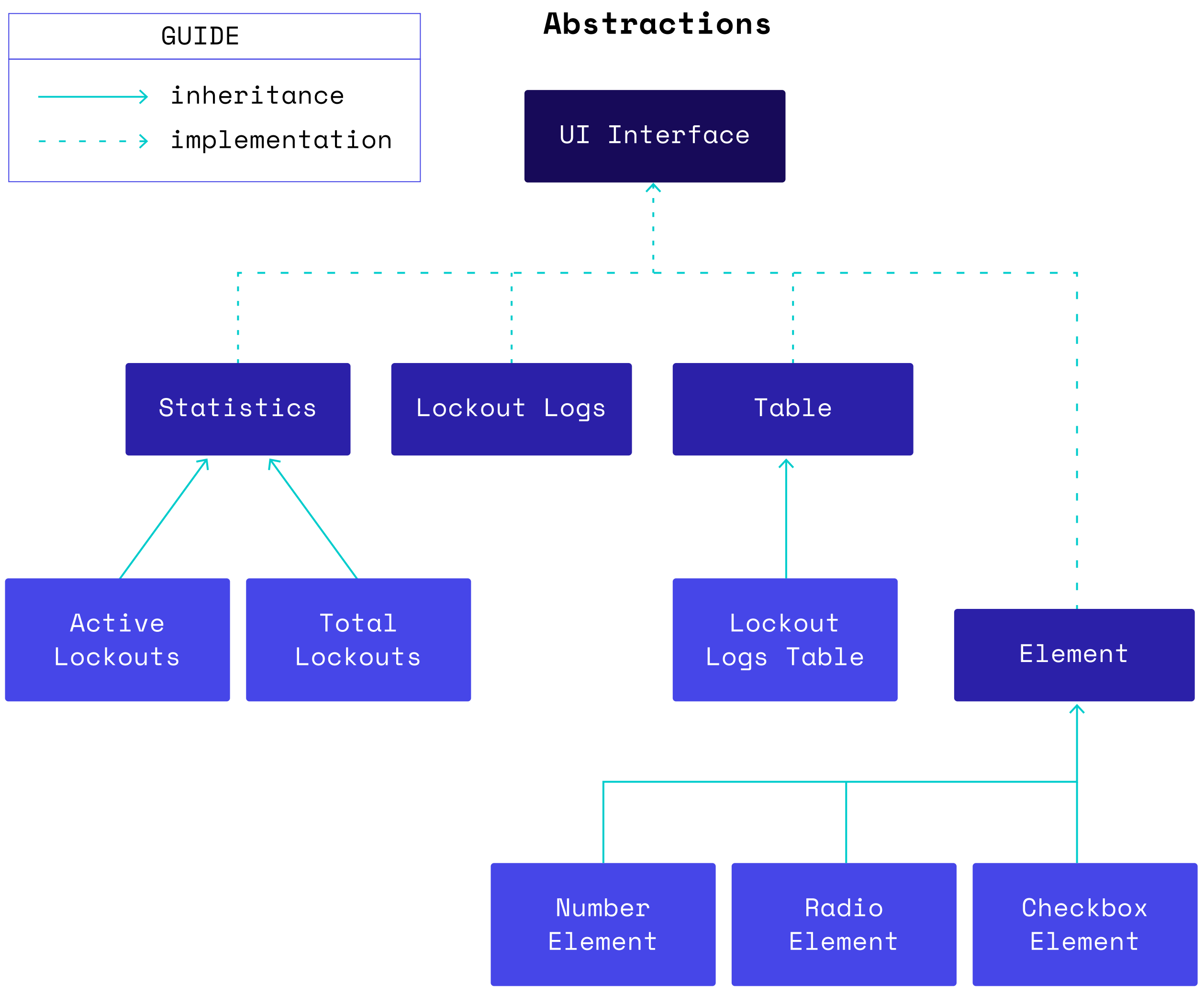
Of course, not all "components" are included in the diagram.

> A structure like this makes the plugin easier to extend; we're able to easily add a field, element or section if the need arises. We'll be able to easily add more components—fields, elements, or sections—by creating new child classes, without having to modify existing ones.

Thinking and Abstracting

Now it's a great time to start thinking about **what** the various components of our plugin do. During the design phase, we don't have to go into much detail about **how** something works.

For example, consider all elements, tables, statistics and pretty much anything else that is going to be displayed to the user. They might be separate components with nothing in common, but will all eventually render some output. **Therefore, some functionality will be common for components that are otherwise completely unrelated.** Of course, this extends to the rest of our components as well.



In the above visual, we see how a UI interface is implemented by multiple classes.

Pay attention to the fact that the UI interface is implemented by the Statistics, Lockout Logs, Table, and Element classes that are referred to as **parent** classes. There's no need for the Radio/Number/Checkbox Element classes to implement the interface directly, since they inherit all interfaces from their parent class. However, a child class **could** override a method of its parent class.

Since we know that our plugin's going to deal with settings, we can safely assume that we'll read and write their values. That is, being able to **get, set,** and **remove** options.

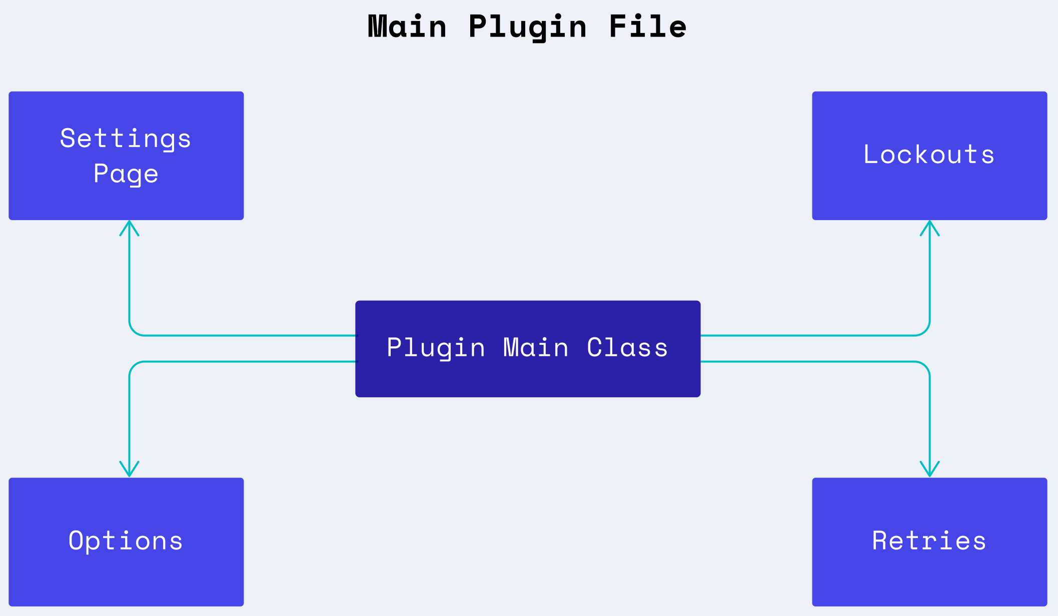
All these actions will be bundled together in a class. We'll probably store our options in the WordPress database or something like that. For now, we don't have to care about **how** or **where** we're going to store our data.

We can keep the get/set/remove options **abstraction** in our minds, simplifying things conceptually, and keep designing our plugin.

Main Plugin File

Here, we'll provide some information about the plugin to WordPress through the header comment and perform some initialization. We'll organize our code, by wrapping everything in a small class.

Depending on how the classes of our plugin will work together, the main class will have to instantiate most of them. As far as we know, this will include classes related to options, admin pages, retries and lockouts.



Potential Classes

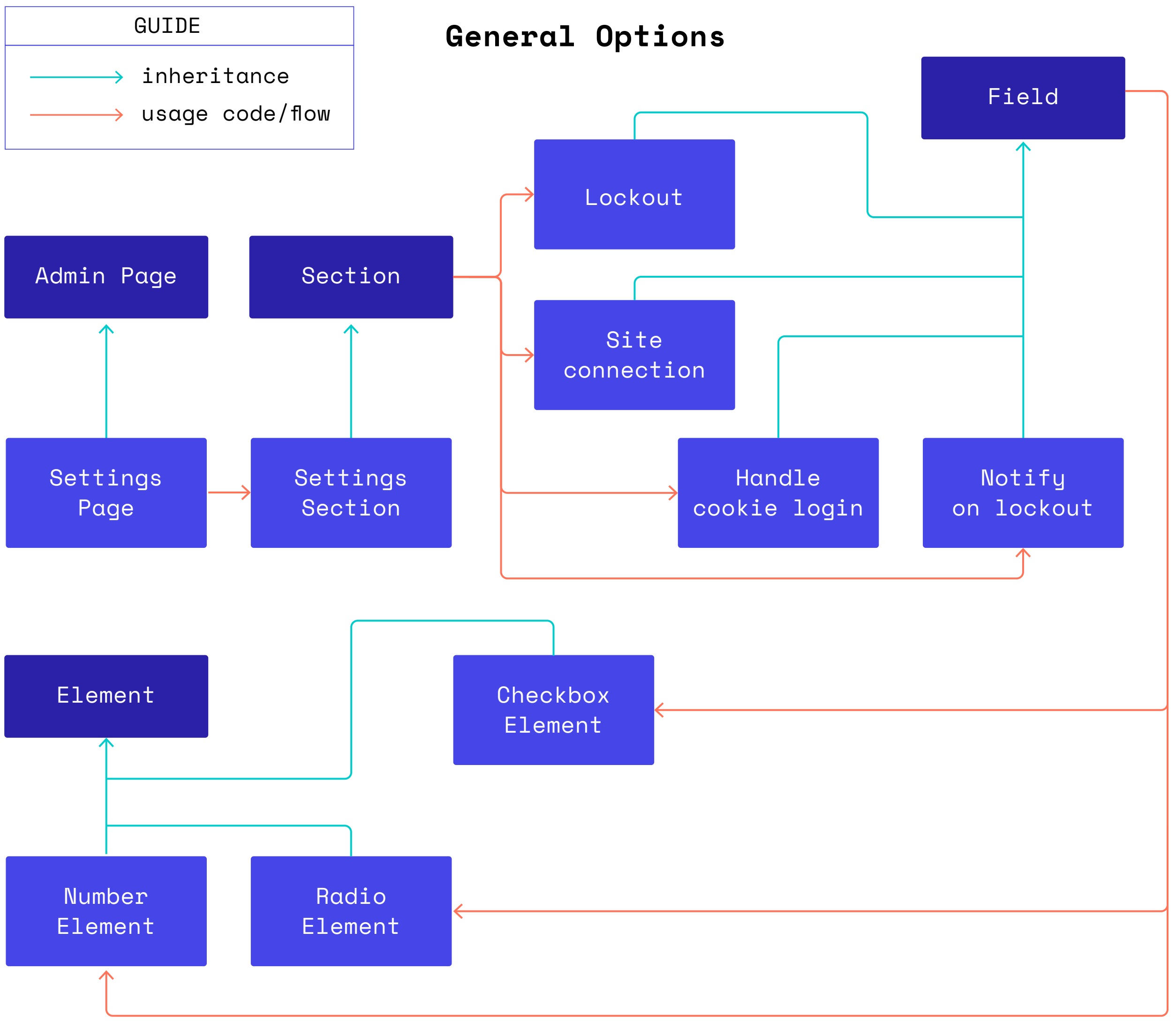
We took some time to try and figure out what classes we're going to need and we ended up with a list as follows:

- ✓ Retries
- ✓ Lockouts
- ✓ Cookies
- ✓ Error messages
- ✓ Email notifications
- ✓ Admin notices
- ✓ Buttons

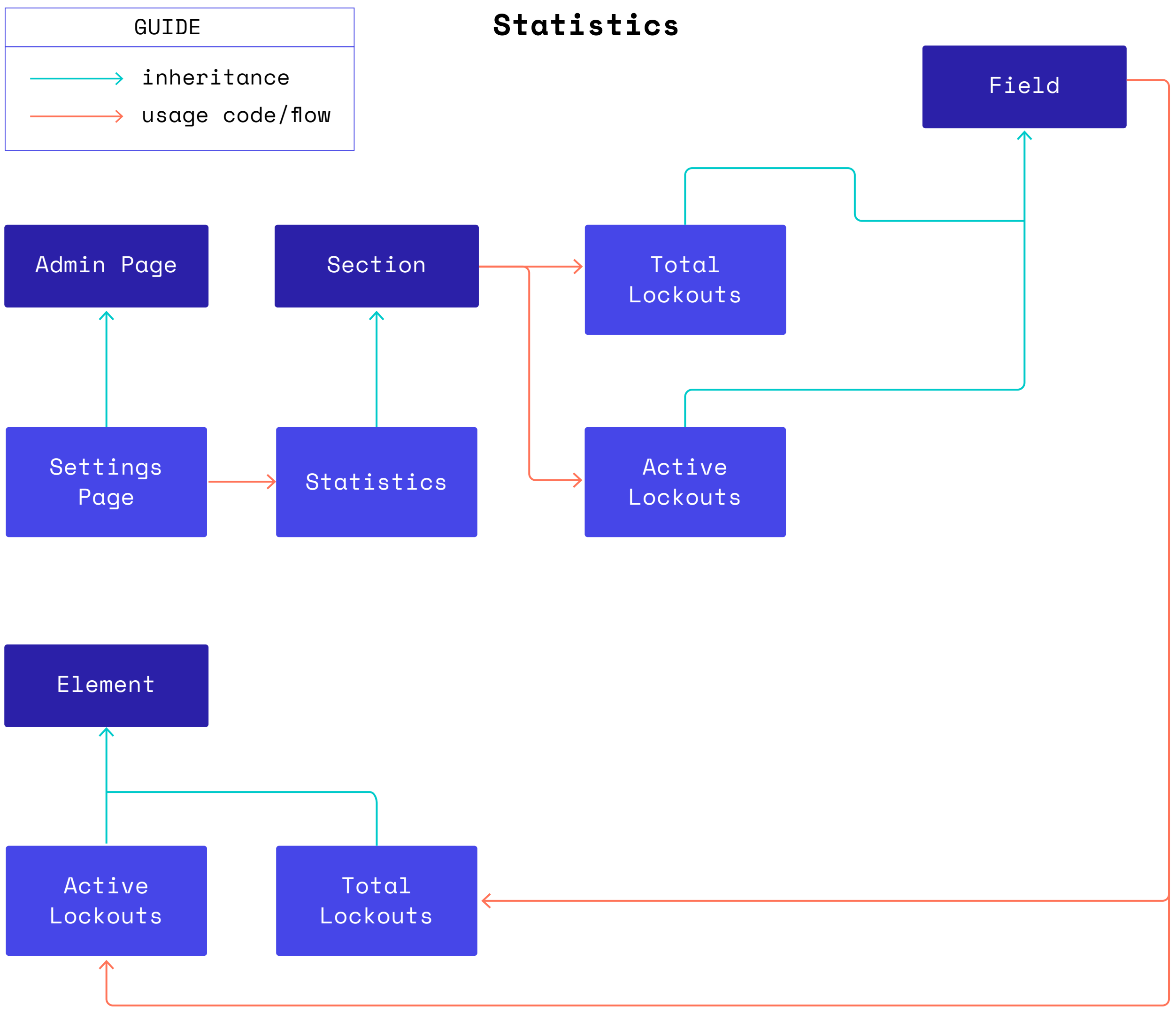
- ✓ Lockout logs
- ✓ Active/Total lockouts
- ✓ IP address

Keep in mind that there's no one single "correct" way to structure your plugin. As with most things in software development, there are multiple, equally valid, ways to solve a problem.

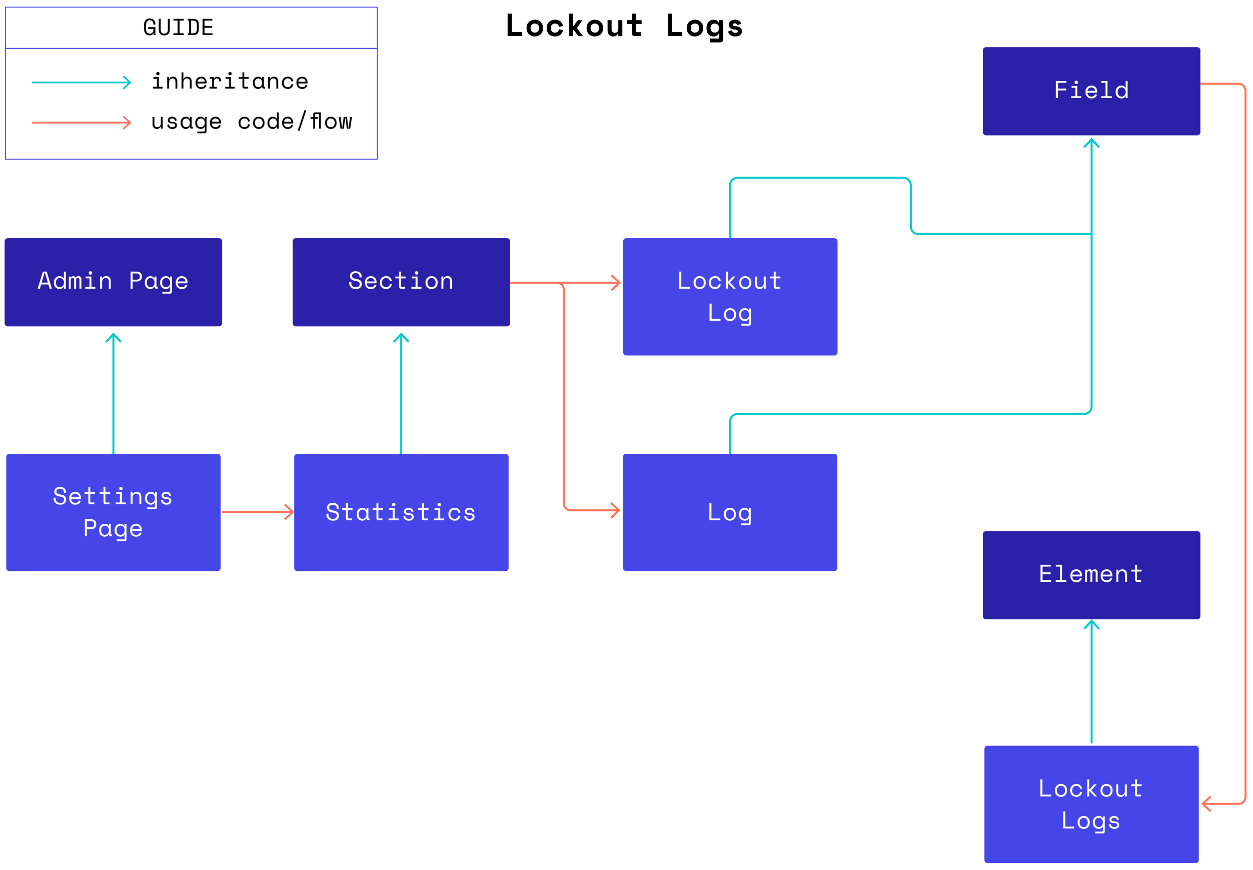
In the "General" section, for example, the relations between our classes would look like this:



The “Statistics” section will be similar to this:



Finally, the “Lockout logs” will be very similar to “Statistics”:



So far we defined our requirements and thought of a design for our new and improved plugin. We explained how we came up with our structure and also provided some simple diagrams showing how our classes will be related to each other.

Now, we’ll get into the most exciting part where we’ll dig deeper into how we implemented it!

PART FIVE

Implementation: The Administration Menu

PART FIVE

Implementation: The Administration Menu

We'll walk you through some parts of the implementation, one step at a time, talking about the very basics of object-oriented programming, PHP syntax, some core concepts, and we'll even glance over the S.O.L.I.D. principles.

Getting Started

We assume you're familiar with WordPress plugin development in general, so we'll focus on the object-oriented aspects of our plugin. If you're new to plugin development or need a refresher, you should learn how to build your first WordPress plugin first.

Let's get started like we always do, by creating a new prsdm-limit-login-attempts.php file, under our plugin directory
(i.e. /wp-content/plugins/prsdm-limit-login-attempts).

The main plugin file is going to include the Plugin Header you're already familiar with:

```
/**
 * Plugin Name: PRSDM Limit Login Attempts
 * Plugin URI: https://pressidium.com
 * Description: Limit rate of login attempts, including by way of cookies,
for each IP.
 * Author: Pressidium
 * Author URI: https://pressidium.com
 * Text Domain: prsdm-limit-login-attempts
 * License: GPL-2.0+
 * Version: 1.0.0
 */
```

And a simple if statement to prevent direct access to it.

```
if ( ! defined( 'ABSPATH' ) ) {
    exit;
}
```

That's all we need for now. We'll revisit this file later!

Building an Administration Menu

When you're developing a plugin, you often need to provide your users with a way to configure it. That's where a settings page comes in. To build one, we're going to add an administration menu that utilizes the WordPress Settings API.

So, let's start thinking about how our object-oriented API would look.

Ideally, we'd like to instantiate our `Pressidium_LLA_Settings_Page` and be done with it. To create an instance of a class, the `new` keyword must be used.

```
new Pressidium_LLA_Settings_Page();
```

Now, let's think about how our `Pressidium_LLA_Settings_Page` class would look.

We'll start by creating a new class, using the `class` keyword:

```
class Pressidium_LLA_Settings_Page {}
```

Our class name has to be prefixed with a unique identifier, Pressidium_LLA_ to prevent any naming collisions with other WordPress plugins. Prefixes prevent other plugins from overwriting and/or accidentally calling our classes. As long as our class names are unique—or we use namespaces—there won't be any conflicts with other plugins.

The Constructor

Now, we'll hook into admin_menu and admin_init. To keep things simple, we'll just call add_action() in our constructor (spoiler alert: we'll change this later).

```
class Pressidium_LLA_Settings_Page {

    /**
     * Settings_Page constructor.
     */
    public function __construct() {
        add_action( 'admin_menu', array( $this, 'add_page' ) );
        add_action( 'admin_init', array( $this, 'register_sections' ) );
    }

}
```

Classes which have a constructor, call this method when an object gets instantiated. So, the `__construct()` method is great for any initialization we might want to perform.

Let's take a closer look at our `add_action()` calls. If you've developed WordPress plugins in the past, you might have expected something like this:

```
add_action( 'admin_menu', 'my_plugin_prefix_add_page' );
```

But instead, we've got:

```
add_action( 'admin_menu', array( $this, 'add_page' ) );
```

You might be confused about the use of an array here. Whenever we want to pass a method of an instantiated object as a callback/callable, we can use an array containing an object at index 0, and a method name at index 1.

What is $this?

It's a pseudo-variable that is available when a method is called from within an object context. `$this` is the value of the calling object. In this case, `$this` is an instance of `Pressidium_LLA_Settings_Page.`

Plus, all of our "functions" are now methods, wrapped in a class, so there's no need to prefix our method names.

Namespaces

Namespaces in PHP allow us to group related classes, interfaces, functions, etc., preventing naming collisions between our code, and internal PHP or third-party classes/functions.

Let's go ahead and use them, so we don't have to prefix any of our classes moving forward.

We'll declare a namespace using the `namespace` keyword.

```
namespace Pressidium;
```

Namespaces can be defined with sub-levels.

```
namespace Pressidium\Limit_Login_Attempts;
```

Since we're building a settings page, we'll declare a "pages" sub-namespace to group anything related to administration pages together.

```
namespace Pressidium\Limit_Login_Attempts\Pages;
```

We can finally get rid of the `Pressidium_LLA_` prefix!

```
namespace Pressidium\Limit_Login_Attempts\Pages;

class Settings_Page {
    // ...
```

Another WordPress plugin containing a `Settings_Page` class isn't an issue anymore, since its class and our class won't live in the same namespace.

When instantiating our `Settings_Page` within the same namespace we can omit it:

```
namespace Pressidium\Limit_Login_Attempts\Pages;

$settings_page = new Settings_Page();
```

When instantiating our `Settings_Page` outside of its namespace, we have to specify it like this:

```
namespace Another\Namespace;

$settings_page = new
\Pressidium\Limit_Login_Attempts\Pages\Settings_Page();
```

Alternatively, we could import our class with the `use` operator:

```
use Pressidium\Limit_Login_Attempts\Pages\Settings_Page;

$settings_page = new Settings_Page();
```

Adding Hook Callbacks

Now, let's declare these `add_page()` and `register_sections()` methods.

```
class Settings_Page {

    /**
     * Settings_Page constructor.
     */
    public function __construct() {
        add_action( 'admin_menu', array( $this, 'add_page' ) );
        add_action( 'admin_init', array( $this, 'register_sections' ) );
    }

    /**
     * Add this page as a top-level menu page.
     */
    public function add_page() {
        // TODO: Implement this method.
    }

    /**
     * Register sections.
     */
    public function register_sections() {
        // TODO: Implement this method.
    }

}
```

Our add_page() method will just call the add_menu_page() WordPress function.

```
public function add_page() {
    add_menu_page(
        __( 'Limit Login Attempts Settings', 'prsdm-limit-login-attempts'
),
        __( 'Limit Login Attempts', 'prsdm-limit-login-attempts' ),
        'manage_options',
        'prsdm_limit_login_attempts_settings',
        array( $this, 'render' ),
        'dashicons-shield-alt',
        null
    );
}
```

That seems like a convoluted way to develop WordPress plugins. It’s simply calling WordPress functions, with extra steps.

Well, that’s not exactly **“reusable”**, we’d still have to write all this extra code for every administration menu/page we want to add.

Refactoring

Let's go ahead and refactor our code a bit to take advantage of object-oriented programming and make our code reusable. We'll start by replacing our hardcoded values in `add_page()` with a few methods, like so:

```
public function add_page() {
    add_menu_page(
        $this->get_page_title(),    // page_title
        $this->get_menu_title(),    // menu_title
        $this->get_capability(),    // capability
        $this->get_slug(),          // menu_slug
        array( $this, 'render' ),   // callback function
        $this->get_icon_url(),      // icon_url
        $this->get_position()       // position
    );
}
```

We'll define these methods as `protected`, so they can be accessed only within the class itself and by its child/parent classes.

```
protected function get_page_title() { /* ... */ }
protected function get_menu_title() { /* ... */ }
protected function get_capability() { /* ... */ }
protected function get_slug() { /* ... */ }
protected function get_icon_url() { /* ... */ }
protected function get_position() { /* ... */ }
```

> **Great! We can now use this class as a reusable, generic class to extend from.**

Redesigning

We told you this was probably going to happen eventually. Here we are, rethinking the design of our class while building it.

Since this is going to be our base class, we'll rename it to a more generic name, like `Admin_Page`. So far, it looks like this:

```
class Admin_Page {

    /**
     * Admin_Page constructor.
     */
    public function __construct() {
        add_action( 'admin_menu', array( $this, 'add_page' ) );
        add_action( 'admin_init', array( $this, 'register_sections' ) );
    }

    /**
     * Add this page as a top-level menu page.
     */
    public function add_page() {
        add_menu_page(
            $this->get_page_title(),    // page_title
            $this->get_menu_title(),    // menu_title
            $this->get_capability(),    // capability
            $this->get_slug(),          // menu_slug
            array( $this, 'render' ),   // callback function
            $this->get_icon_url(),      // icon_url
            $this->get_position()       // position
        );
    }

    /**
     * Register sections.
     */
    public function register_sections() {
        // TODO: Implement this method.
    }

    protected function get_page_title() { /* ... */ }
    protected function get_menu_title() { /* ... */ }
    protected function get_capability() { /* ... */ }
    protected function get_slug() { /* ... */ }
    protected function get_icon_url() { /* ... */ }
    protected function get_position() { /* ... */ }

}
```

We can now create a separate `Settings_Page` that extends that `Admin_Page` base class.

```
class Settings_Page extends Admin_Page {
    // ...
}
```

That's a great example of inheritance, one of the core concepts of object-oriented programming. When extending a class, the child class— `Settings_Page` , in this case—inherits all of the public and protected methods, properties, and constants from the parent class.

We can make use of this and set some default values. For example, we'll set a generic icon for all menu pages, by defining our `get_icon_url()` method like this:

```
class Admin_Page {

    // ...

    /**
     * Return the menu icon to be used for this menu.
     *
     * @link https://developer.wordpress.org/resource/dashicons/
     *
     * @return string
     */
    protected function get_icon_url() {
        return 'dashicons-admin-generic';
    }

}
```

Unless a class overrides those methods, they will retain their original functionality. So, by default, all child classes are going to use that generic icon.

However, if we want to set another icon for a specific menu page, we can simply override the `get_icon_url()` method in our child class, like so:

```
class Settings_Page extends Admin_Page {

    protected function get_icon_url() {
        return 'dashicons-shield-alt';
    }

}
```

There are some values, though, that must be different for each child class. For instance, the menu slug—the fourth argument of `add_menu_page()` —should be unique for each menu page.

If we'd define this method in our `Admin_Page` base class, we'd need a way to make sure that every single child class overrides this method. Well, we can do something even better. We can declare the method's signature and completely skip its implementation.

Enter abstract methods!

Abstract Classes and Methods

Methods defined as ***abstract*** simply declare the method's signature and they cannot define its implementation.

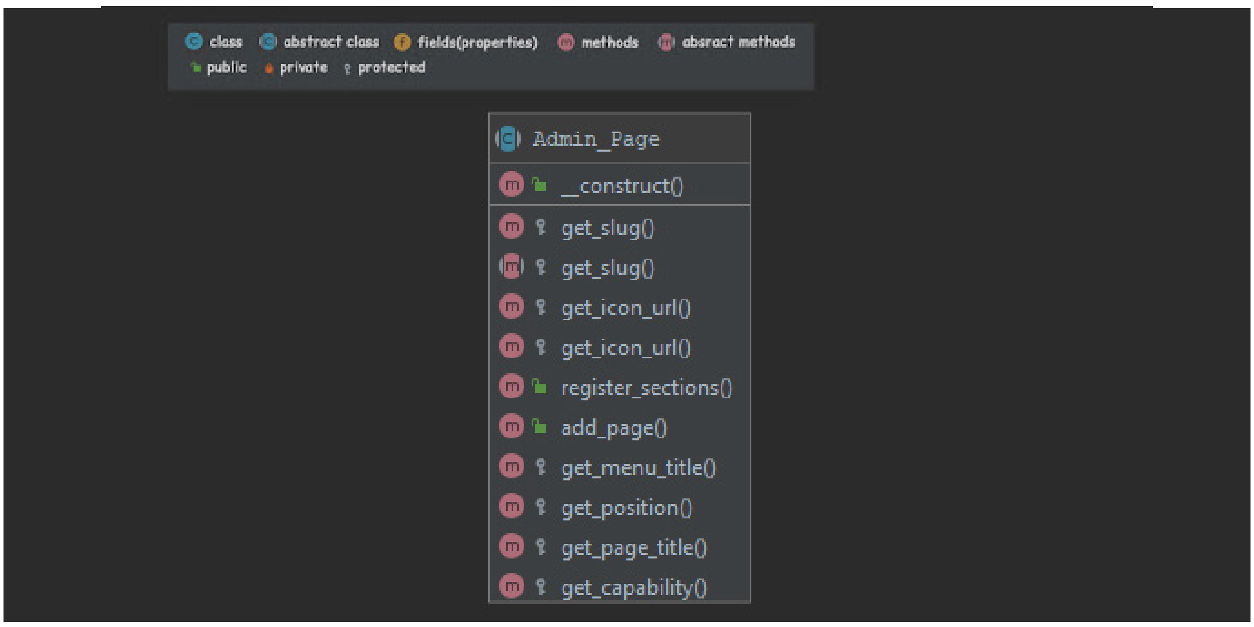
```
/**
 * Return page slug.
 *
 * @return string
 */
abstract protected function get_slug();
```

Any class that contains at least one abstract method ***must*** also be abstract. That means, our Admin_Page class should be defined as abstract as well.

```
abstract class Admin_Page {
    // ...
```

It's also important to point out here that classes defined as abstract cannot be instantiated. So, we can no longer directly instantiate `Admin_Page.`

Here's also a visualization of the class:



When inheriting from an abstract class, the child class must define all methods marked abstract in the declaration of its parent class. Meaning, that our `Settings_Page` has to implement the get_slug() method.

```
class Settings_Page extends Admin_Page {

    // ...

    protected function get_slug() {
        return 'prsdm_limit_login_attempts_settings';
    }

    // ...

}
```

In the same way, we should implement the rest of the protected methods the `add_page()` needs.

Before proceeding on how we'll register the sections and fields of the admin page and render their content, let's talk a bit about settings in WordPress.

The Settings API

We'll assume you're already familiar with the Settings API. But, just in case, here's the gist of it:

- settings_fields() — Outputs nonce, action, and option_page fields for a settings page. Basically, the hidden form fields.
- do_settings_sections() — Prints out all settings sections (and their fields) added to a particular settings page.
- add_settings_section() — Adds a new section to a settings page.
- add_settings_field() — Adds a new field to a section of a settings page.
- register_setting() — Registers a setting and its data.

If you are not already familiar with this, you can pause reading this e-book and check our related article on how to build the settings page for a custom plugin.

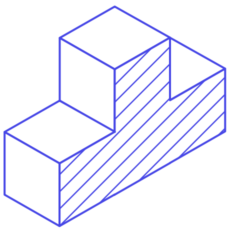
Now that we're on the same page, let's get back to our `register_sections()` method. Once again, we have to take a step back and think about our API.

Since we've defined the `add_page()` method in the `Admin_Page` class, we'll also define the `render()` method there as well. We'll pass the return values of our other methods as arguments to the WordPress functions.

```
abstract class Admin_Page {

    // ...

    /**
     * Render this admin page.
     */
    public function render() {
        ?>

        <div class="wrap">
            <form action="options.php" method="post">
                <h1><?php echo esc_html( $this->get_page_title() ); ?></h1>
                <?php
                settings_fields( $this->get_slug() );
                do_settings_sections( $this->get_slug() );
                submit_button( __( 'Change Options',
'prsdm-limit-login-attempts' ) );
                ?>
            </form>
        </div>

        <?php
    }

}
```

That way, we won't have to bother directly with these WordPress functions ever again. That's because any admin page we may add in the future will be built through a child class just like the `Settings_Page` , and its rendering will be done through the inherited `render()` method of the `Admin_Page` parent class.

> Great! We created the classes that are responsible for registering an administration menu and adding a settings page.
>
> In the next step , we'll keep building our settings page and register its sections, fields, and elements.



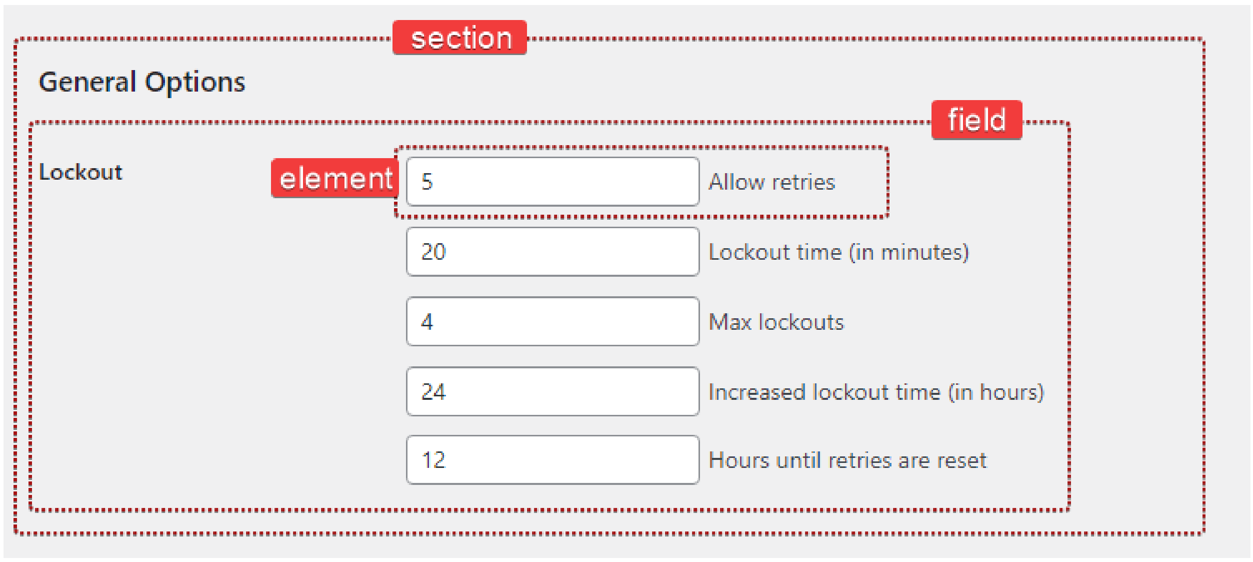
PART SIX

Implementation: Registering the Sections

PART SIX

Implementation: Registering the Sections

As we have already explained, an admin page consists of **sections,** each section contains one or more **fields,** and each of those fields contain one or more **elements.**



General Options

Lockout

5	Allow retries
20	Lockout time (in minutes)
4	Max lockouts
24	Increased lockout time (in hours)
12	Hours until retries are reset

How would that look in code?

```
public function register_sections() {
    $my_section = $this->register_section( /* ... */ );
    $my_field   = $my_section->add_field( /* ... */ );
    $my_element = $my_field->add_element( /* ... */ );
}
```

Alright, that seems easy to use and we can already tell that we'll probably need to create three new classes: `Section` , `Field`, and `Element`.

```
class Section {}
class Field {}
class Element {}
```

Let's take a moment and ask ourselves what we know so far about these classes.

- `$my_section->add_field()` → The `Section` class should be able to add (and store) a new `Field` object.
- `$my_field->add_element()` → The `Field` class should be able to add (and store) a new `Element` object.

We'll also write the `add_field()` method to create and add a new field.

```
class Section {

    /**
     * @var Field[] Section field objects.
     */
    protected $fields = array();
```

This `$fields` variable is a class member and it's what we call a **property.** Properties are PHP variables, living in a class, and they can be of any data type (`string`, `integer`, `object` , etc.).

We'll also write the `add_field()` method to create and add a new field.

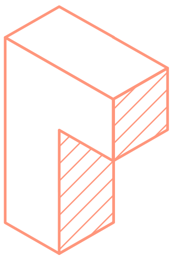
```
public function add_field() {
    $field = new Field( /* ... */ );

    $this->fields[] = $field;

    return $field;
}
```

This method creates a new `Field` object, adds it to the fields property and returns that newly-created object. Pretty straightforward.

Let's repeat the same process for the `Field` class as well.

```
class Field {

    /**
     * @var Element[] Field elements.
     */
    private $elements = array();

    /**
     * Create a new element object.
     *
     * @return Element
     */
    private function create_element() {
        return new Element( /* ... */ );
    }

    /**
     * Add a new element object to this field.
     */
    public function add_element() {
        $element = $this->create_element();

        $this->elements[] = $element;
    }
```

That’s a start! What’s next?



The Section Class

We need to call add_settings_section(), when a new section is created. Once again, the constructor method is a great way to perform our initialization. Let's add it in the class:

```
class Section {

    // ...

    public function __construct() {
        add_settings_section(
            $this->id,
            $this->title,
            array( $this, 'print_description' ),
            $this->page
        );
    }

}
```

It seems that a Section needs a slug-name to identify it (used in the id attribute of tags). It can also have a title, a description, and belongs to a specific page.

```php
class Section {

    /**
     * @var Field[] Section field objects.
     */
    protected $fields = array();

    /**
     * @var string Section title.
     */
    public $title;

    /**
     * @var string Section id.
     */
    public $id;

    /**
     * @var string Slug-name of the settings page this section belongs to.
     */
    public $page;

    /**
     * @var string Section description.
     */
    public $description;
```

We could set the title of the section, by doing something like this:

```
$section = new Section();
$section->title = __( 'Hello world', 'prsdm-limit-login-attempts' );
```

Well, that's not quite right. Even though the code above it's perfectly valid, it doesn't actually do what we expect it to do.

The constructor method is executed when a new Section object is created. So `add_settings_section()` will be called before we even get a chance to set the title. As a result, the section won't have a title.

The title needs to be available during the initialization of our object, so we need to do this in the constructor.

```
class Section {

    /**
     * @var string Section title.
     */
    private $title;

    public function __construct( $title ) {
        $this->title = $title;
        // ...
    }

    // ..
```

Beware that `$this->title` refers to the title class property, where `$title` refers to the constructor's argument.

```
class Section {

    /**
     * @var string Section title.
     */
    private $title;

    public function __construct( $title ) {
        $this->title = $title;
        // ...
    }

    // ..
```

Here, we also take advantage of the visibility. Since our `$title` property will only be accessed by the class that defined it, we can declare it `private`. Therefore, we prevent it from being accessed outside the class.

Oh, and we also have to add a `print_description()` method which is going to, well, print the section's description.

```
/**
 * Print the section description.
 */
public function print_description() {
    echo esc_html( $this->description );
}
```

Putting all together, our Section class looks like this.

```
class Section {

    /**
     * @var Field[] Section field objects.
     */
    protected $fields = array();

    /**
     * @var string Section title.
     */
    private $title;

    /**
     * @var string Section id.
     */
    private $id;

    /**
     * @var string Slug-name of the settings page this section belongs to.
     */
    private $page;

    /**
     * @var string Section description.
     */
    private $description;

    /**
     * Section constructor.
     *
     * @param string $id          Section id.
     * @param string $title       Section title.
     * @param string $page        Slug-name of the settings page.
     * @param string $description Section description.
     */
    public function __construct( $id, $title, $page, $description ) {
        $this->id          = $id;
        $this->title       = $title;
        $this->page        = $page;
        $this->description = $description;

        add_settings_section(
            $this->id,
            $this->title,
            array( $this, 'print_description' ),
            $this->page
        );
    }

    /**
     * Print the section description.
     */
    public function print_description() {
        echo esc_html( $this->description );
    }

    /**
     * Create and add a new field object to this section.
     */
    public function add_field() {
        $field = new Field( /* ... */ );

        $this->fields[] = $field;

        return $field;
    }

}
```

The Field Class

In a similar way to `Section`, we can now proceed and build the `Field` class, which is going to utilize the `add_settings_field()` WordPress function.

```
class Field {

    /**
     * @var Element[] Field elements.
     */
    private $elements = array();


    /**
     * @var string ID of the section this field belongs to.
     */
    private $section_id;

    /**
     * @var string Field description.
     */
    private $description;

    /**
     * Field constructor.
     *
     * @param string $id          Field ID.
     * @param string $label       Field label.
     * @param string $page        Slug-name of the settings page.
     * @param string $section_id  ID of the section this field belongs to.
     * @param string $description Field description.
     */
    public function __construct( $id, $label, $page, $section_id,
$description ) {
        $this->section_id  = $section_id;
        $this->description = $description;
        add_settings_field(
            $id,
            $label,
            array( $this, 'render' ),
            $page,
            $this->section_id
        );
    }

}
```

Here, we'd also like to provide default values for the ID, label, and description of the field. We can do this by passing an options array to the constructor and use the wp_parse_args() WordPress function to parse those options.

```
class Field {

    /**
     * @var int Number of fields instantiated.
     */
    private static $number_of_fields = 0;

    // ...

    /**
     * Field constructor.
     *
     * @param string $section_id  ID of the section this field belongs to.
     * @param string $page        Slug-name of the settings page.
     * @param array  $options     Options.
     */
    public function __construct( $section_id, $page, $options = array() ) 
        self::$number_of_fields++;

        $options = wp_parse_args(
            $options,
            array(
                'label'       => sprintf(
                    __( 'Field #%s', 'prsdm-limit-login-attempts' ),
                    self::$number_of_fields
                'id'          => 'field_' . self::$number_of_fields,
                'description' => ''
            )
        );

        $this->section_id  = $section_id;
        $this->description = $options['description'];

        add_settings_field(
            $options['id'],
            $options['label'],
            array( $this, 'render' ),
            $page,
            $this->section_id
        );
    }

}
```

The wp_parse_args() function will allow us to merge the user defined values (the `$options` array) with the default values.

```
array(
    'label'       => sprintf(
        __( 'Field #%s', 'prsdm-limit-login-attempts' ),
        self::$number_of_fields
),
    'id'          => 'field_' . self::$number_of_fields,
    'description' => ''
)
```

We also have to set unique labels for each field. We can handle this by setting the label to a prefix (`'field_'`) followed by a number, which will be increased every time a new Field object is created. We'll store this number in the `$number_of_fields` static property.

```
/**
 * @var int Number of fields instantiated.
 */
private static $number_of_fields = 0;
```

A static property can be accessed directly without having to create an instance of a class first.

```
'id' => 'field_' . self::$number_of_fields
```

The `self` keyword is used to refer to the current class and, with the help of the scope resolution operator `::` (commonly called "double colon"), we can access our static property.

That way, in the constructor, we always access the same `$number_of_fields` property, increasing its value each time an object is created, which results in a unique label attached to each field.

Going forward, the `render()` method, after printing the description (if one exists), iterates through all the elements and renders each one of them.

```
public function render() {
    if ( ! empty( $this->description ) ) {
        printf(
            '<p class="description">%s</p>',
            esc_html( $this->description )
        );
    }

    foreach ( $this->elements as $key => $element ) {
        $element->render();
    }
}
```

Putting it all together...

```
class Field {

    /**
     * @var int Number of fields instantiated.
     */
    private static $number_of_fields = 0;

    /**
     * @var Element[] Field elements.
     */
    private $elements = array();

    /**
     * @var string ID of the section this field belongs to.
     */
    private $section_id;
```

```
 /**
  * @var string Field description.
  */
private $description;

/**
  * Field constructor.
  *
  * @param string $section_id  ID of the section this field belongs to.
  * @param string $page        Slug-name of the settings page.
  * @param array  $options     Options.
  */
public function __construct( $section_id, $page, $options = array() ) {
    self::$number_of_fields++;

    $options = wp_parse_args(
        $options,
        array(
            'label'       => sprintf(
                /* translators: %s is the unique s/n of the field. */
                __( 'Field #%s', 'prsdm-limit-login-attempts' ),
                self::$number_of_fields
            'id'          => 'field_' . self::$number_of_fields,
            'description' => ''
        )
    );

    $this->section_id  = $section_id;
    $this->description = $options['description'];

    add_settings_field(
        $options['id'],
        $options['label'],
        array( $this, 'render' ),
        $page,
        $this->section_id
    );
}
```

```php
    /**
     * Create a new element object.
     *
     * @return Element
     */
    private function create_element() {
        return new Element( /* ... */ );
    }

    /**
     * Add a new element object to this field.
     */
    public function add_element() {
        $element = $this->create_element();

        $this->elements[] = $element;
    }

    /**
     * Render the field.
     */
    public function render() {
        if ( ! empty( $this->description ) ) {
            printf(
                '<p class="description">%s</p>',
                esc_html( $this->description )
            );
        }

        foreach ( $this->elements as $key => $element ) {
            $element->render();
        }
    }

}
```

The Element Class

Going forward, we'll build the `Element` class in a similar fashion!

We'll start writing the class like this:

```
class Element {

    /**
     * @var int Number of elements instantiated.
     */
    private static $number_of_elements = 0;

    /**
     * @var string Element label.
     */
    private $label;

    /**
     * @var string Element name.
     */
    private $name;

    /**
     * @var mixed Element value.
     */
    private $value;

    /**
     * Element constructor.
     *
     * @param string $section_id Section ID.
     * @param array  $options    Options.
     */
    public function __construct( $section_id, $options = array() ) {
        self::$number_of_elements++;
```

```php
        $options = wp_parse_args(
            $options,
            array(
                'label' => sprintf(
                    /* translators: %s is the unique s/n of the element. */
                    __( 'Element #%s', 'prsdm-limit-login-attempts' ),
                    self::$number_of_elements
                ),
                'name'  => 'element_' . self::$number_of_elements
            )
        );

        $this->label = $options['label'];
        $this->name  = $options['name'];
        $this->value = '';
    }

    /**
     * Render the element.
    */
    public function render() {
        ?>

        <fieldset>
            <label>
                <input
                    type="number"
                    name="<?php echo esc_attr( $this->name ); ?>"
                    id="<?php echo esc_attr( $this->name ); ?>"
                    value="<?php echo esc_attr( $this->value ); ?>"
                />
                <?php echo esc_html(); ?>
            </label>
        </fieldset>

        <?php
    }

}
```

Make sure you're escaping your output—like we're doing here, using the esc_attr() and esc_html() WordPress functions—to prevent any cross-site scripting attacks. Even though we're rendering our elements only in admin pages, it's still a good idea to always escape any output data.

NOTE: Cross-site scripting (or XSS) is a type of security vulnerability typically found in web applications. XSS enables attackers to inject client-side code into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.

When we were gathering the plugin's requirements, we noticed that there are multiple element types—checkboxes, radio buttons, number fields etc. When we came up with our design, we made the decision to build an `Element` class meant to be extended. So, we know we're going to end up with a child class for each element type.

The output should differ depending on the element type, so we'll turn `render()` into an abstract method. That means, of course, that the class itself should also be abstract.

```
abstract class Element {

    /**
     * @var int Number of elements instantiated.
     */
    private static $number_of_elements = 0;
```

```php
/**
 * @var string Element label.
 */
protected $label;

/**
 * @var string Element name.
 */
protected $name;

/**
 * @var mixed Element value.
 */
protected $value;

/**
 * Element constructor.
 *
 * @param string $section_id Section ID.
 * @param array  $options    Options.
 */
public function __construct( $section_id, $options = array() ) {
    self::$number_of_elements++;

    $options = wp_parse_args(
        $options,
        array(
            'label' => sprintf(
                /* translators: %s is the unique s/n of the element. */
                __( 'Element #%s', 'prsdm-limit-login-attempts' ),
                self::$number_of_elements
            ),
            'name'  => 'element_' . self::$number_of_elements
        )
    );

    $this->label = $options['label'];
    $this->name  = $options['name'];
    $this->value = '';
}
```

```
    /**
     * Render the element.
     */
    abstract public function render();

}
```

For example, a `Number_Element` class would look like this:

```
class Number_Element extends Element {

    /**
     * Render the element.
     */
    public function render() {
        ?>

        <fieldset>
            <label>
                <input
                    type="number"
                    name="<?php echo esc_attr( $this->name ); ?>"
                    id="<?php echo esc_attr( $this->name ); ?>"
                    value="<?php echo esc_attr( $this->value ); ?>"
                />
                <?php echo esc_html(); ?>
            </label>
        </fieldset>

        <?php
    }

}
```

> Notice that we're building our classes so they can all be used in the same way. Calling the `render()` method on any child of Element will output some HTML.
>
> That's an example of polymorphism, one of the core concepts of object-oriented programming.

Polymorphism

"Polymorphism" means literally **"many forms"** (from the greek words "poly" meaning "many", and "morphe" meaning "form"). An Element child class can have **many forms**, since it can take any form of a class in its parent hierarchy.

We can use a `Number_Element` , a `Checkbox_Element` , or any other **sub-type** in any place an `Element` object is expected, since all child objects can be **used** in the exact same way (i.e. calling their `render()` method), while still being able to **behave** differently (the output will differ for each element type).

As you can probably tell, **polymorphism and inheritance are closely related concepts.**

Substitutability

The **Liskov Substitution Principle (or LSP), the "L" in S.O.L.I.D.**, states:

"In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program."

In layman's terms, **you should be able to use any child class in place of its parent class without any unexpected behavior.**

Factories

Let's go back to our `Field` class, where we currently have a `create_element()` method creating a new `Element`.

```
/**
 * Create a new element object.
 *
 * @return Element
 */
private function create_element() {
    return new Element( /* ... */ );
}

/**
 * Add a new element object to this field.
 */
public function add_element() {
    $element = $this->create_element();

    $this->elements[] = $element;
}
```

A method that returns a new object is often called a simple factory (not to be confused with “factory method”, which is a design pattern).

Knowing that any subtype is usable in place of the `Element` parent class, we’ll go ahead and modify this factory, so it will be able to create objects of any child class.

```php
/**
 * Create a new element object.
 *
 * @throws Exception If there are no classes for the given element type.
 * @throws Exception If the given element type is not an `Element`.
 *
 * @param string $element_type
 * @param array  $options
 *
 * @return Element
 */
private function create_element( $element_type, $options ) {
    $element_type = __NAMESPACE__ . '\\Elements\\' . $element_type;

    if ( ! class_exists( $element_type ) ) {
        throw new Exception( 'No class exists for the specified type' );
    }

    $element = new $element_type( $this->section_id, $options );

    if ( ! ( $element instanceof Element ) ) {
        throw new Exception( 'The specified type is invalid' );
    }

    return $element;
}
```

```
/**
 * Add a new element object to this field.
 *
 * @param string $element_type
 * @param array  $options
 */
public function add_element( $element_type, $options ) {
    try {
        $element = $this->create_element( $element_type, $options );
        $this->elements[] = $element;
    } catch ( Exception $e ) {
        // Handle the exception
    }
}
```

We start by prefixing the element type with the current name:

```
$element_type = __NAMESPACE__ . '\\Elements\\' . $element_type;
```

The `__NAMESPACE__` magic constant contains the current namespace name.

Then, we make sure that there’s a class for the specified element type:

```
if ( ! class_exists( $element_type ) ) {
    throw new Exception( 'No class exists for the specified type' );
}
```

Next, we create a new object:

```
$element = new $element_type( $this->section_id, $options );
```

And lastly, we make sure that the newly-created object is indeed an instance of Element:

```
$element = new $element_type( $this->section_id, $options );
```

Extending

It's worth pointing out that we've built our plugin to be extensible. Adding different kinds of pages, sections, elements is as easy as creating a new class that extends `Admin_Page` , `Section` , `Element` etc. These base classes do not include any code that needs to be changed to add a new page, section, or element.

The **Open/Closed Principle (or OCP), the "O" in S.O.L.I.D.**, states:

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

This means that we should be able to extend a class like `Admin_Page` and reuse it, but we shouldn't have to modify it to do that.

> **Now that we registered our sections, fields, and elements, we'll take a closer look at how we can improve the way we manage our WordPress hooks.**

PART SEVEN

Implementation: Managing WordPress Hooks

PART SEVEN

Implementation: Managing WordPress Hooks

Up to this point, interacting with the Plugin API meant calling `add_action()` and `add_filters()` in the constructor of each class.

So far that approach was good enough, as it kept things simple and allowed us to focus on learning more about object-oriented programming with WordPress. However, it's not ideal.

If an object registers all of its hooks when it's created, things like unit testing become tricky.

NOTE: Unit tests should test each "unit" in isolation. Even if you're not writing unit tests at the moment, writing testable code will save you a lot of time refactoring later, if you ever decide to write tests.

The Hooks Manager

Let’s take this one step further and introduce a new class to manage our hooks, we’ll call it `Hooks_Manager`. This class is going to be responsible for the registration of all of our hooks. So, we’ll create a new class with a `register()` method.

```
class Hooks_Manager {

    /**
     * Register the hooks of the given object.
     *
     * @param object $object
     */
    public function register( $object ) {
        // Register the hooks the specified object needs
    }

}
```

Next, we create a new object:

```
interface Hooks {

    /**
     * Return the actions to register.
     *
     * @return array
     */
    public function get_actions();

}
```

You can think of an interface as a **contract**, where a class that implements that interface is “contractually bound” to implement all methods defined in that interface.

For example, a `Login_Error` class that hooks into the `login_head` action, **must** implement the `get_actions()` method of our `Hooks` interface.

```php
class Login_Error implements Hooks {

    public function get_actions() {
        return array(
            'login_head' => array( 'add_errors', 10, 1 ),
        );
    }

}
```

The `register()` method of `Hooks_Manager` accepts an object, calls its `get_actions()` method and registers all of its actions.

```php
public function register( $object ) {
    $actions = $object->get_actions();

    foreach ( $actions as $action_name => $action_details ) {
        $method        = $action_details[0];
        $priority      = $action_details[1];
        $accepted_args = $action_details[2];

        add_action(
            $action_name,
            array( $object, $method ),
            $priority,
            $accepted_args
        );
    }
}
```

Let’s add a `get_filters()` method to our interface, so we can register both actions and filters.

```
interface Hooks {

    /**
     * Return the actions to register.
     *
     * @return array
     */
    public function get_actions();

    /**
     * Return the filters to register.
     *
     * @return array
     */
    public function get_filters();

}
```

Back to our `Login_Error` class, we need to implement this new `get_filters()` method.

```
class Login_Error implements Hooks {

    public function get_actions() {
        return array(
            'login_head' => array( 'add_errors', 10, 1 ),
        );
    }
    public function get_filters() {
        return array(
            'authenticate'      => array( 'track_credentials', 10, 3 ),
            'shake_error_code' => array( 'add_error_code', 10, 1 ),
            'login_errors'      => array( 'format_error_message', 10, 1 ),
        );
    }

}
```

We'll rename the `register()` method of our `Hooks_Manager` to `register_actions()`. We'll also add a `register_filters()` method. These two methods will be responsible for registering actions and filters respectively.

```
class Hooks_Manager {

    /**
     * Register the actions of the given object.
     *
     * @param object $object
     */
    private function register_actions( $object ) {
        $actions = $object->get_actions();
```

```php
        foreach ( $actions as $action_name => $action_details ) {
            $method        = $action_details[0];
            $priority      = $action_details[1];
            $accepted_args = $action_details[2];

            add_action(
                $action_name,
                array( $object, $method ),
                $priority,
                $accepted_args
            );
        }
    }

    /**
     * Register the filters of the given object.
     *
     * @param object $object
     */
    private function register_filters( $object ) {
        $filters = $object->get_filters();

        foreach ( $filters as $filter_name => $filter_details ) {
            $method        = $filter_details[0];
            $priority      = $filter_details[1];
            $accepted_args = $filter_details[2];

            add_filter(
                $filter_name,
                array( $object, $method ),
                $priority,
                $accepted_args
            );
        }
    }

}
```

Now we can add a `register()` method again, which is simply going to call both `register_actions()` and `register_filters()` .

```
class Hooks_Manager {

    /**
     * Register an object.
     *
     * @param object $object
     */
    public function register( $object ) {
        $this->register_actions( $object );
        $this->register_filters( $object );
    }

    // ...
```

What if a class doesn't need to register both actions and filters? The `Hooks` interface contains two methods: `get_actions()` and `get_filters()`. All classes that implement that interface will be forced to implement both methods.

```
class Cookie_Login implements Hooks {

    public function get_actions() {
        return array(
            'auth_cookie_bad_username' => array( 'handle_bad_username',
10, 1 ),
            'auth_cookie_bad_hash'     => array( 'handle_bad_hash',
10, 1 ),
            'auth_cookie_valid'        => array( 'handle_valid',
10, 2 ),
        );
    }

    public function get_filters() {
        return array();
    }

}
```

For example, the `Cookie_Login` class has to register only actions, but it's now forced to implement the `get_filters()` method just to return an empty array.

The **Interface Segregation Principle (ISP), the "I" in S.O.L.I.D.**, states:

"No client should be forced to depend on methods it does not use."

Meaning that what we're doing now is exactly what we shouldn't be doing.

Interface Segregation

We can fix this by splitting our interface into smaller, more specific ones so our classes will only have to know about the methods that are of interest to them.

```
interface Actions {


    /**
     * Return the actions to register.
     *
     * @return array
     */
    public function get_actions();

}

interface Filters {


    /**
     * Return the filters to register.
     *
     * @return array
     */
    public function get_filters();

}
```

We don't need both `get_actions()` and `get_filters()` anymore, we can implement only the `Actions` interface and get rid of `get_filters()`

```
class Cookie_Login implements Actions {

    public function get_actions() {
        return array(
            'auth_cookie_bad_username' => array( 'handle_bad_username', 10,
1 ),
            'auth_cookie_bad_hash'     => array( 'handle_bad_hash', 10, 1 ),
            'auth_cookie_valid'        => array( 'handle_valid', 10, 2 ),
        );
    }

}
```

On the other hand, `Login_Error` , which needs actions and filters, just has to implement both interfaces. Classes may implement more than one interface by separating them with a comma.

```
class Login_Error implements Actions, Filters {

    public function get_actions() {
        return array(
            'login_head' => array( 'add_errors', 10, 1 ),
        );
    }

    public function get_filters() {
        return array(
            'authenticate'     => array( 'track_credentials', 10, 3 ),
            'shake_error_code' => array( 'add_error_code', 10, 1 ),
            'login_errors'     => array( 'format_error_message', 10, 1 ),
        );
    }

}
```

Now that we've segregated our interface, we just have to update the `register()` method of `Hooks_Manager` to reflect our changes.

```
class Hooks_Manager {

    /**
     * Register an object.
     *
     * @param object $object
     */
    public function register( $object ) {
        if ( $object instanceof Actions ) {
            $this->register_actions( $object );
        }

        if ( $object instanceof Filters ) {
            $this->register_filters( $object );
        }
    }

    // ...
```

That way, we conditionally call only `register_actions()` , only `register_filters()` , or both, based on the interface(s) the specified object implements.

To actually use the hooks manager:

```
$hooks_manager = new Hooks_Manager();
$hooks_manager->register( $login_error );
$hooks_manager->register( $cookie_login );
```

That's it! We can now use that object to manage hooks across the entire codebase.

Of course, there are several ways to manage your hooks in an object-oriented way, we just showed you one of them. You should experiment and find one that fits your needs.

> Next, in the last part of this document, we will see how we can handle options in an object-oriented way, talk about encapsulation, abstraction and how to decouple your classes to create a flexible plugin that's easy to extend!

PART EIGHT

Implementation: Options

PART EIGHT

Implementation: Options

So far we only needed to store user-defined options, so we utilized the Settings API. However, our plugin has to be able to read/write options itself to “remember” how many times an IP address has attempted to login unsuccessfully, if it’s currently locked out, etc.

We need an object-oriented way to store and retrieve options. During the “Design” phase, we briefly discussed this, but abstracted away some of the implementation details, focusing solely on the actions we’d like to be able to perform—**getting**, **setting**, and **removing** an option.

We’ll also sort of “group” options together based on their section to keep them organized. That’s purely based on personal preference

Let’s turn this into an interface:

```
interface Options {

    /**
     * Return the option value based on the given option name.
     *
     * @param string $name Option name.
     * @return mixed
     */
    public function get( $name );
    /**
     * Store the given value to an option with the given name.
     *
     * @param string $name       Option name.
     * @param mixed  $value      Option value.
     * @param string $section_id Section ID.
     * @return bool              Whether the option was added.
     */
    public function set( $name, $value, $section_id );

    /**
     * Remove the option with the given name.
     *
     * @param string $name       Option name.
     * @param string $section_id Section ID.
     */
    public function remove( $name, $section_id );

}
```

Ideally, we'd be able to interact with the [WordPress Options API](), by doing something like this:

```
$options = new WP_Options();
$options->get( 'retries' );
```

At this point, you might be wondering why we don't just use the `get_option()` WordPress function, instead of going into the trouble of creating our own interface and class. While using WordPress functions directly would be a perfectly acceptable way of developing our plugin, by going a step further and creating an interface to depend on, we stay flexible.

Our `WP_Options` class is going to implement our `Options` interface. That way, we'll be ready if our needs change in the future. For instance, we might need to store our options in a custom table, in an external database, in memory (e.g. Redis), you name it. By depending on an abstraction (i.e. interface), changing something in the implementation, is as simple as creating a new class implementing the same interface.

WP_Options

Let's start writing our WP_Options class, by retrieving all options using the get_option() WordPress function in its constructor.

```
class WP_Options {

    /**
     * @var array Stored options.
     */
    private $options;

    /**
     * WP_Options constructor.
     */
    public function __construct() {
        $this->options = get_option( Plugin::PREFIX );
    }

}
```

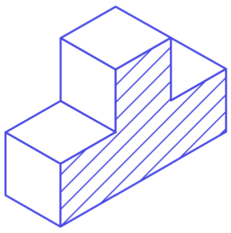
Since the `$options` property will be used internally, we'll declare it `private` so it may only be accessed by the class that defined it, the `WP_Options` class.

Now, let's implement our `Options` interface by using the `implements` operator.

```
class WP_Options implements Options {
    // ...
```

Our IDE is yelling at us to either declare our class abstract or implement the `get()`, `set()`, and `remove()` methods, defined in the interface.

So, let's start implementing these methods!



Getting an option

We'll start with the `get()` method, which is going to look for the specified option name in our `$options` property, and either return its value or `false` if it doesn't exist.

```
class WP_Options implements Options {

    private $options;

    public function __construct() {
        $this->options = get_option( Plugin::PREFIX );
    }

    /**
     * Return the option value based on the given option name.
     *
     * @return mixed
     */
    public function get( $option_name ) {
        if ( ! isset( $this->options[ $option_name ] ) ) {
            return false;
        }

        return $this->options[ $option_name ];
    }

}
```

Now it's a good time to think about **default options.**

Default options

As mentioned previously, we'd like to group options together, based on their section. So, we'll probably split the options into a couple of sections. The "General Options" section and another one for the data we need to keep track of. Lockouts, retries, lockout logs, and total number of lockouts—we'll arbitrarily call this state.

We'll use a **constant** to store our default options. The value of a constant can't be changed while our code is executing, which makes it ideal for something like our default options. Class constants are allocated once per class, and not for each class instance.

NOTE: The name of a constant is in all uppercase by convention.

```
const DEFAULT_OPTIONS = array(
    'general_options' => array(
        'allowed_retries'                 => 4,
        'normal_lockout_time'             => 1200,  // 20 minutes
        'max_lockouts'                    => 4,
        'long_lockout_time'               => 86400, // 24 hours
        'hours_until_retries_reset'       => 43200, // 12 hours
        'site_connection'                 => 'direct',
        'handle_cookie_login'             => 'yes',
        'notify_on_lockout_log_ip'        => true,
        'notify_on_lockout_email_to_admin' => false,
        'notify_after_lockouts'           => 4
    ),
    'state' => array(
        'lockouts'       => array(),
        'retries'        => array(),
        'lockout_logs'   => array(),
        'total_lockouts' => 0
    )
);
```

In the `DEFAULT_OPTIONS` nested array, we've set a default value for all of our options.

What we'd like to do next, is store the default option values in the database once the plugin is initialized, by using the `add_option()` WordPress function.

```
class WP_Options {

    public function __construct() {
        $all_options = array();

        foreach ( self::DEFAULT_OPTIONS as $section_id =>
$section_default_options ) {
            $db_option_name  = Plugin::PREFIX . '_' . $section_id;
            $section_options = get_option( $db_option_name );

            if ( $section_options === false ) {
                add_option( $db_option_name, $section_default_options );
                $section_options = $section_default_options;
            }

            $all_options = array_merge( $all_options, $section_options );
        }

        $this->options = $all_options;
    }

}
```

Let's take a closer look at this snippet. First, we iterate the default options array and retrieve the options using the `get_option()` WordPress function.

```
foreach ( self::default_options as $section_id => $section_default_options
) {
    $db_option_name  = Plugin::PREFIX . '_' . $section_id;
    $section_options = get_option( $db_option_name );
    // ...
```

Then, we check whether each option already exists in the database, and if not, we store its default option.

```
if ( $section_options === false ) {
    add_option( $db_option_name, $section_default_options );
    $section_options = $section_default_options;
}
```

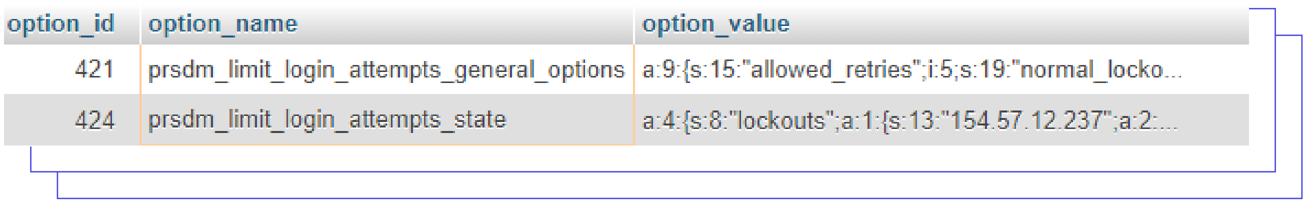
Finally, we collect the options of all sections.

```
$all_options = array_merge( $all_options, $section_options );
```

And store them in the `$options` property so we'll be able to access them later on.

```
$this->options = $all_options;
```

The WordPress options table in the database is going to have a couple of rows, where the `option_name` consists of the plugin's prefix concatenated to the section name.



option_id	option_name	option_value
421	prsdm_limit_login_attempts_general_options	a:9:{s:15:"allowed_retries";i:5;s:19:"normal_locko...
424	prsdm_limit_login_attempts_state	a:4:{s:8:"lockouts";a:1:{s:13:"154.57.12.237";a:2:...

Storing an option

Similarly, we'd like to easily store a new option in the database, and overwrite any previous value, like this:

```
$options = new Options();
$options->set( 'retries', 4 );
```

So, let's implement the set() method, which is going to use the

```
/**
 * Store the given value to an option with the given name.
 *
 * @param string $name       Option name.
 * @param mixed  $value      Option value.
 * @param string $section_id Section id. Defaults to 'state'.
 * @return bool              Whether the option was added.
 */
public function set( $name, $value, $section_id = 'state' ) {
    $db_option_name = Plugin::PREFIX . '_' . $section_id;
    $stored_option  = get_option( $db_option_name );

    $stored_option[ $name ] = $value;

    return update_option( $db_option_name, $stored_option );
}
```

Removing an option

Lastly, we'll implement the `remove()` method, which is going to set the option to its initial value:

```
/**
 * Remove the option with the given name.
 *
 * @param string $name       Option name.
 * @param string $section_id Section id. Defaults to 'state'.
 * @return bool              Whether the option was removed.
 */
public function remove( $name, $section_id = 'state' ) {
    $initial_value = array();

    if ( isset( self::DEFAULT_OPTIONS[ $section_id ][ $name ] ) ) {
        $initial_value = self::DEFAULT_OPTIONS[ $section_id ][ $name ];
    }

    return $this->set( $name, $initial_value, $section_id );
}
```

> We've bundled everything together in a single class. All options-related data (i.e. our properties) and the implementation details (i.e. the methods we just implemented) are **encapsulated** in the `WP_Options` class.

Encapsulation/Abstraction

Wrapping everything in a single class, enclosing the internals (as if in a capsule), essentially “hiding” them from the outside world, is what we call **encapsulation.** Encapsulation is another core concept of object-oriented programming.

Using the `Options` interface, we focused on **what we do** with our options instead of **how we do it,** abstracting the idea of options, simplifying things conceptually. This is what we call **abstraction**, another core concept of object-oriented programming.

> **Encapsulation and abstraction are completely different concepts, but clearly, as you can see, highly-related. Their main difference is that encapsulation exists in the implementation level, while abstraction exists in the design level.**

Dependencies

Let’s consider the following scenario:

There’s a `Lockouts` class, responsible for determining whether an IP address should get locked out, what should be the duration of that lockout, if an active lockout is still valid or has expired etc. That class contains a `should_get_locked_out()` method, responsible for determining whether an IP address should get locked out. That method would need to read the maximum number of allowed retries before an IP address gets locked out, which is a configurable value, meaning it’s stored as an **option.**

So, the code we just described would look similar to this:

```
class Lockouts {

    // ...

    /**
     * @var WP_Options An instance of `WP_Options`.
     */
    private $options;

    /**
     * Lockouts constructor
     */
    public function __construct() {
        $this->options = new WP_Options();
    }

    /**
     * Return the number of retries.
     *
     * @return int
     */
    private function get_number_of_retries() {
        // ...
    }

    /**
     * Check whether this IP address should get locked out.
     *
     * @return bool
     */
    public function should_get_locked_out() {
        $retries         = $this->get_number_of_retries();
        $allowed_retries = $this->options->get( 'allowed_retries' );
        return $retries % $allowed_retries === 0;
    }

    // ...

}
```

Basically, we're creating a new instance of `WP_Options` in the constructor, and then use that instance to retrieve the value of the `allowed_retries` option.

That's absolutely fine, but we have to keep in mind that our `Lockouts` class now depends on `WP_Options` . We call WP_Options a **dependency.**

If our needs change in the future, for example, we need to read/write options on an external database, we'd need to replace the `WP_Options` with a `DB_Options` class. That doesn't seem so bad, if we need to retrieve options in only one class. However, it may get a bit tricky when there are many classes with multiple dependencies. Any changes to a single dependency will likely ripple across the codebase, forcing us to modify a class if one of its dependencies changes.

> We can eliminate this issue by rewriting our code to follow the **Dependency Inversion Principle.**

Decoupling

The **Dependency Inversion Principle (DIP), the “D” in S.O.L.I.D.**, states:

- *High-level modules should not import anything from low-level modules. Both should depend on abstractions.*
- *Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.*

In our case, the `Lockouts` class is the “high-level module” and it depends on a “low-level module”, the `WP_Options` class.

We’ll change that, using Dependency Injection, which is easier than it may sound. Our `Lockouts` class will receive the objects it depends on, instead of creating them.

```
class Lockouts {

    // ...

    /**
     * Lockouts constructor.
     *
     * @param WP_Options $options
     */
    public function __construct( WP_Options $options ) {
        $this->options = $options;
    }

    // ...

}
```

So, we **inject** a dependency:

```
$options  = new WP_Options();
$lockouts = new Lockouts( $options );
```

We just made our `Lockouts` class easier to maintain since it's now loosely coupled with its `WP_Options` dependency. Additionally, we'll be able to mock the dependencies, making our code easier to test. Replacing the `WP_Options` with an object that mimics its behavior will allow us to test our code without actually executing any queries on a database.

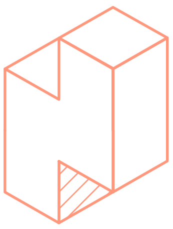
```
/**
 * Lockouts constructor.
 *
 * @param WP_Options $options
 */
public function __construct( WP_Options $options ) {
    $this->options = $options;
}
```

Even though we have given the control of `Lockouts` ' dependencies to another class (as opposed to `Lockouts` controlling the dependencies itself), `Lockouts` still expects a `WP_Options` object. Meaning, that it still depends on the concrete `WP_Options` class, instead of an abstraction. As previously mentioned, both modules should depend on abstractions.

Let's fix that!

```
/**
 * Lockouts constructor.
 *
 * @param Options $options
 */
public function __construct( Options $options ) {
    $this->options = $options;
}
```

And by simply changing the type of the `$options` argument from the `WP_Options` class to the `Options` interface, our `Lockouts` class depends on an abstraction and we're free to pass a `DB_Options` object, or an instance of any class that implements the same interface, to its constructor.



Single Responsibility

It's worth noting that we used a method called `should_get_locked_out()` to check whether the IP address should get locked out or not.

```php
/**
 * Check whether this IP address should get locked out.
 *
 * @return bool
 */
public function should_get_locked_out() {
    $retries         = $this->get_number_of_retries();
    $allowed_retries = $this->options->get( 'allowed_retries' );

    return $retries % $allowed_retries === 0;
}
```

We could easily write a one-liner like this:

```php
if ( $this->get_number_of_retries() % $this->options->get(
'allowed_retries' ) === 0 ) {
```

However, moving that piece of logic into its own little method, has a lot of benefits.

- If the condition to determine whether an IP address should get locked out ever changes, we'll only have to modify this method (instead of searching for all occurrences of our if statement)
- Writing unit tests becomes easier when each "unit" is smaller
- Improves the readability of our code a lot

Reading this:

```
if ( $this->should_get_locked_out() ) {
    // ...
```

seems to us way easier than reading that:

```
if ( $this->get_number_of_retries() % $this->options->get(
 'allowed_retries' ) === 0 ) {
    // ...
```

We've done this for pretty much every method of our plugin. Extracting methods out of longer ones till there's nothing else to extract. The same goes for classes, each class and method should have a single responsibility.

The **Single Responsibility Principle (SRP), the "S" in S.O.L.I.D.,** states:

"Every module, class, or function in a computer program should have responsibility over a single part of that program's functionality, and it should encapsulate that part."

Or, as *Robert C. Martin ("Uncle Bob")* says:

"A class should have one, and only one, reason to change."

Revisiting the main plugin file

At the moment, our main plugin file contains only this:

```
/**
 * Plugin Name: PRSDM Limit Login Attempts
 * Plugin URI: https://pressidium.com
 * Description: Limit rate of login attempts, including by way of cookies,
   for each IP.
 * Author: Pressidium
 * Author URI: https://pressidium.com
 * Text Domain: prsdm-limit-login-attempts
 * License: GPL-2.0+
 * Version: 1.0.0
 */

if ( ! defined( 'ABSPATH' ) ) {
    exit;
}
```

Once again, we'll wrap everything in a Plugin class, this time just to avoid naming collisions.

```
namespace Pressidium\Limit_Login_Attempts;

if ( ! defined( 'ABSPATH' ) ) {
    exit;
}

class Plugin {

    /**
     * Plugin constructor.
     */
    public function __construct() {
        // ...
    }

}
```

We'll instantiate this `Plugin` class at the end of the file, which is going to execute the code in its constructor.

```
new Plugin();
```

In the constructor we'll hook into the plugins_loaded action, which fires once activated plugins have loaded.

```php
public function __construct() {
    add_action( 'plugins_loaded', array( $this, 'init' ) );
}

public function init() {
    // Initialization
}
```

We'll also call a `require_files()` method to load all of our PHP files.

```php
public function __construct() {
    $this->require_files();
    add_action( 'plugins_loaded', array( $this, 'init' ) );
}

private function require_files() {
    require_once __DIR__ . '/includes/Sections/Section.php';
    require_once __DIR__ . '/includes/Pages/Admin_Page.php';
    require_once __DIR__ . '/includes/Pages/Settings_Page.php';
    // ...
}
```

Finally, we'll initialize our plugin by creating some objects in our `init()` method.

NOTE: The following snippet contains only a small part of the main plugin file. You can read the actual file at the plugin's GitHub repository.

```
public function init() {
    $options       = new Options();
    $hooks_manager = new Hooks_Manager();

    $settings_page = new Settings_Page( $options );
    $hooks_manager->register( $settings_page );
    // ...
}
```

Organizing the files

Keeping your files organized is vital, especially when working on large plugins with lots of code. Your folder structure should group similar files together, helping you and your teammates stay organized.

We've already defined a namespace (`Pressidium\Limit_Login_Attempts`), containing several sub-namespaces for `Pages` , `Sections` , `Fields` , `Elements` , etc. Following that hierarchy to organize our directories and files, we ended up with a structure similar to this:

```
.
├── includes
│   ├── Hooks
│   │   ├── Actions.php
│   │   ├── Filters.php
│   │   └── Hooks_Manager.php
│   ├── Pages
│   │   ├── Admin_Page.php
│   │   └── Settings_Page.php
│   ├── Sections
│   │   ├── Fields
│   │   │   ├── Elements
│   │   │   │   ├── Checkbox_Element.php
│   │   │   │   ├── Custom_Element.php
│   │   │   │   ├── Element.php
│   │   │   │   ├── Number_Element.php
│   │   │   │   └── Radio_Element.php
│   │   │   └── Field.php
│   │   └── Section.php
│   └── WP_Options.php
├── prsdm-limit-login-attempts.php
└── uninstall.php
```

Each file contains a single class. Files are named after the classes they contain, and directories and subdirectories are named after the (sub-)namespaces.

> There are multiple architecture patterns and naming schemes you may use. It's up to you to pick one that makes sense to you and suits the needs of your project. When it comes to structuring your project, the important thing is to be **consistent.**

Conclusion

Congratulations! You've completed our e-Book about WordPress and object-oriented programming.

Hopefully you learned a few things and are excited to start applying what you learned on your own projects!

Here's a quick recap of what we covered in this e-book:

- **Requirements gathering:** We decided on what the plugin should do.
- **Design:** We thought about how the plugin will be structured, the relationships between our potential classes, and a high-level overview of our abstractions.
- **Implementation:** We wrote the actual code of some key parts of the plugin. While doing that, we introduced you to several concepts and principles.

However, we barely scratched the surface of what OOP is and has to offer. Getting good at a new skill takes practice, so go ahead and start building your own object-oriented WordPress plugins.

Happy coding!

PRESSIDIUM®

Visit **www.pressidium.com** to find out more about the award winning technology behind our High Availability WordPress hosting.

If you'd like to take your website to the next level then you can **sign up online**. Alternatively, **contact us**, and one of the team will help select the perfect plan for your specific requirements.

@Pressidium