WooCommerce for Developers

Igor Benic

WooCommerce for Developers

Extend WooCommerce sites with code

Igor Benić

This book is for sale at http://leanpub.com/woodev

This version was published on 2018-05-28

Contents

About the Book

Hi there! Thank you for purchasing this book. Not only you have supported the work of a developer but you have invested in your career!

This book will teach you how to extend WooCommerce and understand how WooCommerce is functioning. You will learn how to develop for WooCoomerce using already defined actions and filters (hooks) that are defined by WooCommerce but also how to implement your own hooks within your plugins.

The whole book will present a specific project that will cover various specific topics. We could learn each topic separately but defining a specific project will require us to focus more on the details. By using this strategy you will be able to learn how to think and develop for specific purposes.

The specific topics that will be discussed and developed here are hard to find on the internet so be sure to study them good because they could come in handy from time to time.

Code in this book will be written in PHP and JavaScript mostly but some HTML and CSS could be seen in some parts.

All the code that you will see in this book can be used outside of it. I am giving you the permission to use them as you want (for non-commercial and commercial projects).

This book was not written by a member of WooCommerce or WooThemes team but only by me (Igor Benić) who is using WooCommerce in WordPress development.

How to read and use the code

Most of chapters (articles, tutorials) here are articles with code examples.

They are written in a way that you, as a reader, can easily understand every part of it. We are always starting from nothing and then by the end of the article you will have a usable code that you can easily edit to your own needs.

Code examples are written in separate boxes from text such as:

```
1  This is a line
```

but mostly the code examples will be consisted of much more lines such as:

```
<?php
This is a code example
$var = "variable in a code example";
```

Since a lot of these examples use the existing code and many of those code lines will be wider than this book's page, you will see sometimes a symbol '\' that indicates the next line to be the part of the line before.

When you see that symbol, ignore it (delete it without adding space) and continue to insert the code along the same line.

Here is an example of a long variable value that should be written in one line:

```
<?php
$long_variable = "This is a variable with a very long value to see how the same \
line is extended in more lines in this book.";
```

In the most articles, we will create a solution through the whole article so bare in mind that some of the code will come after a previous one.

Some code will not be a new code, but actually a redefined or refactored version of a previous code in the same article.

If you apply, copy or write the code in the article into another file, please be aware that only one opening *php* tag is needed. The opening tag will be used again, only if there was a need to close the previous one, for example: to write some regular HTML.

To show you what I mean let's use an example of this two code examples:

Code Example 1

```
<?php
$variableA = "Variable A";
```

Code Example 2

```
<?php
$variableB = "Variable B";
```

If we want to place them in one file, we would use only the first opening **php** tag like this:

```php
<?php
$variableA = "Variable A";
$variableB = "Variable B";
```

WooCommerce 3.x

I have decided to write only for up-to-date WooCommerce version. If you ever need to add a part to support old WooCommerce 2.6.x code or even older, than you can use this code to wrap the features:

```
if( version_compare( WC()->version, ‘3.0.0’, '<' ) ) {
     // backward compatibility here.
}
```

The WooDev Project

WooDev is just a name combined from the title of this book “WooCommerce for Developers”. WooDev will be the main project for which we will create different children projects such as:

- Shipping Method
- Restrictions for the shipping method
- General restriction on checkout
- Product Type
- Product Type connected to other WordPress content
- Product Type connected to an external service

Requirements

To follow this book you should have WordPress installed and WooCommerce activated. Be sure to code on your local machine or development environment since everything in here should be also tested on a staging server before using it on your production server.

The book will be refreshed when a refactoring is needed. You should have the latest versions of WordPress and WooCommerce installed.

Custom Shipping Method

Creating WooCommerce Shipping methods can be really fun. But to have fun you first need to know what your shipping method can do or can't do.

This is really important before entering any code. I want to be sure that you understand what WooCommerce Shipping Class can do so we will go through all the Shipping Class code.

The basic abstract class **WC_Shipping_Method** can be found in *woocommerce/includes/abstracts/abstract-wc-shipping-method.php*

Shipping API

Since you can extend or modify each class as you want, you can even modify some methods in the Shipping Class.

We will not go into modification of some previously defined methods in the abstract class, but if you would like to see how something was modified you can go into the folder *includes/shipping* inside **woocommerce** and look at each of those methods how they were defined.

There are still some methods that should be defined when creating your custom Shipping Method. Before we go into that, we should learn about the important attributes of our shipping class:

- **$id** - A required unique identifier for the shipping class
- **$method_title** - Title that will show in the admin area
- **$title** - Title that will be shown on the cart or checkout page
- **$method_description** - Used to describe our shipping method in the admin area
- **$availablity** - Indicator if the shipping method is available or not
- **$countries** - Array of countries that this shipping method is shipping to or not. Depending on *$availability*
- **$tax_status** - if set to *taxable*, the tax will be charged is possible
- **$minimum_fee** - a fee that will be charged when using this method when there are not other fees set,
- **$enabled** - indicator if this shipping method is enabled or not
- **$instance_id** - we can have more than one shipping method from the same class
- **$instance_form_fields** - fields that we use for settings
- **$instance_settings** - instance settings from database
- **$supports** - what this shipping method supports (settings, shipping-zones, instance-settings and/or instance-settings-modal)

What are those supports settings?

- **settings** - backward compatibility for old versions of WooCommerce
- **shipping-zones** - functionality for zones + instances
- **instance-settings** - instance settings used instead of settings
- **instance-settings-modal** - settings are opened in the modal and not on separate page

Now that you know which attributes are the most important ones and which you can easily set, we should also learn about some of the methods you should also set if needed:

- **__construct** - the constructor method will set the id and some other important attribues
- **init** - used to initialise the shipping fields and get the values for that fields
- **init_form_fields** - used to define all the form fields for our shipping method
- **calculate_shipping** - this, besides the constructor method, has to be set to register or shipping method cost or costs. Use the method **add_rate** to add the rates inside *calculate_shipping* method.

Countries and Availability

As described above, if there is an array of countries the shipping method can become available or not for those countries.

This is easily set by using the attribute **$availability**. We are setting that attribute when defining our own custom Shipping Method class.

This attribute can be left as it is without setting it in our own custom class or we can define it as follows:

- **specific** - Ships only to set countries
- **including** - Ships only to set countries
- **excluding** - Ships to all other countries than the set ones

The array of countries set with their ISO Codes. To find out about the ISO Codes of the countries you have to set use a site such as https://countrycode.org/[1].

[1] countrycode.org

Defining the Shipping Method

It is very important to define your shipping method. By defining your shipping method, I mean to define how our shipping method will calculate the cost, how much weight can it ship or which is the maximum dimension of our packages.

What about the countries and the availability of our shipping method? We will create a real world shipping method. I will use a shipping company in my own country **TISAK**.

TISAK operates in zones. Each country has their own zone and each zone has its own price. There are also weight and dimension limitations so we need to be aware of that also.

TISAK ships packages so each package is defined by their maximum dimensions and weight. We will not allow our customer to choose which package to use. We will instead automatically set which package to use based on the weight and dimension since that will be the package we will have to use when shipping with TISAK.

Since the manual for TISAK is written on Croatian I will not show it here. I can only say that TISAK does ship to around 123 countries + Croatia.

The cost of shipping is differently defined when shipping inside Croatia or when shipping to other countries. We will have them inside the countries array. There are also different price tags for each zone and each package.

Packages which TISAK uses are:

- Small
- Medium
- Large

When shipping inside Croatia prices for each package are:

- Small - 15kn (~ $2)
- Medium - 20kn (~ $3)
- Large - 25kn (~ $4)

When shipping to other countries prices for each package are also defined by zones:

- Zone 0
  - Small - 95kn (~ $15)
  - Medium - 105 (~ $16)
  - Large - 135kn (~ $21)
- Zone 1
  - Small - 220kn (~ $34)

    - Medium - 250kn (∼ $39)
    - Large - 275 (∼ $43)
- Zone 2
    - Small - 260kn (∼ $40)
    - Medium - 300kn (∼ $45)
    - Large - 360kn (∼ $55)
- Zone 3
    - Small - 470kn (∼ $72)
    - Medium - 550kn (∼ $85)
    - Large - 790kn (∼ $122)

Since we will automatically assign which package will be used when calculating the shipping cost we need to know the dimensions (in cm) of each package (length x width x height):

- Small - 20x20x15
- Medium - 30x20x20
- Large - 40x30x15

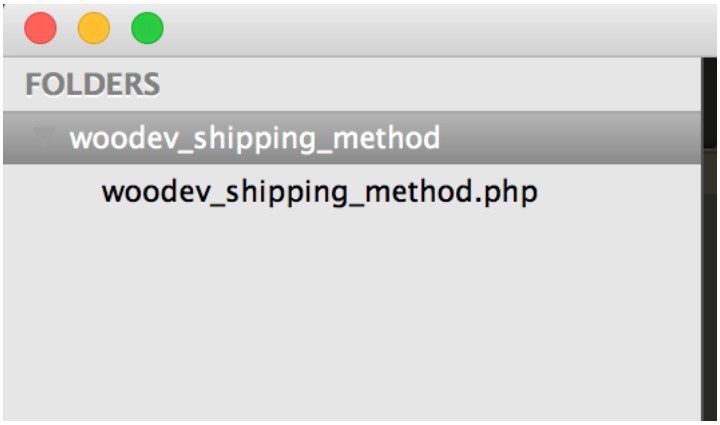
I have used the unit *cm* since that is how TISAK has defined their package dimensions. You can use a different one as long as it is allowed in WooCommerce.

The last part that we should define is the weight. TISAK does not define the cost of their shipping by weight but they do have a limit of maximum 10kg on any package. If our cart items surpass this limit in weight, then our shipping method will not be available.

Now that we have defined our shipping method, we can start coding! Prepare your developing environment and let's start coding.

Shipping Class

We will start first off with a simple WordPress plugin definition. Create your own plugin folder and give it a name to your liking. I will call it **woodev_shipping_method**. Inside that folder create a file with the same name and the *php* extension.



Folder and File for Custom Shipping Method

Let's enter the plugins' definition inside our newly created file:

```
<?php
/*
Plugin Name: WooCommerce For Developers Custom Shipping
Plugin URI: https://leanpub.com/woodev
Description: A Custom Shipping Method done in the "WooCommerce for Developers" e\
Book
Version: 1.0
Author: Igor BeniÄ‡
Author URI: http://ibenic.com
textdomain: woodev_shipping
*/
```

Go to your WordPress admin area inside the menu **Plugins** and activate your newly create plugin. By activating it, all our changes can be seen instantly on our website.

Make sure WooCommerce is active

The headline kind of tells you what is going to be done here, right? We will check if WooCommerce is active and then proceed with our functionality.

There is no point in adding our Shipping Method to WooCommerce if WooCommerce is not installed and active. So let's add this chunk of code next:

```
/**
 * Check if WooCommerce is active
 */
if ( in_array( 'woocommerce/woocommerce.php',
apply_filters( 'active_plugins', get_option( 'active_plugins' ) ) ) ) {

        // our code will go here because we are sure WooCommerce is active

}
```

Basically, we get all the active plugins and then check if WooCommerce is there by using *plugin_-folder/plugin_file.php* string that is saved for every activated plugin.

The Base Class & Shipping Settings

We will define a function that will hold the definition of our custom shipping method and hook it to a WooCommerce action **woocommerce_shipping_init** which is used to get all registered WooCommerce Shipping Methods.

```
function woodev_tisak_shipping() {
  if ( ! class_exists( 'WOODEV_TISAK_Shipping' ) ) {
    class WOODEV_TISAK_Shipping extends WC_Shipping_Method {

    }
  }
}
add_action( 'woocommerce_shipping_init', 'woodev_tisak_shipping' );
```

In our function *woodev_tisak_shipping* we have defined a new class **WOODEV_TISAK_Shipping** that extends the abtract shipping class.

Now we need to add the constructor method inside our new class:

```
// ...
class WOODEV_TISAK_Shipping extends WC_Shipping_Method {

/**
 * Constructor for your shipping class
 *
 * @access public
 * @return void
 */
 public function __construct() {
   $this->id        = 'woodev_tisak_shipping';
   $this->method_title        = __( 'TISAK Shipping', 'woodev_shipping' );
   $this->method_description        = __( 'TISAK Shipping Settings', 'woodev_shipping' \
);
   $this->title        = __("TISAK","woodev_shipping");
   $this->init();
   $this->enabled        = true;
 }
// ...
```

So here we have defined the unique **id** of our shipping method, title that will be displayed in the admin area, description that will be displayed in the admin area and the title that will be displayed on our cart or checkout pages.

Method *init* will be used to get all the settings and settings' fields for this shipping method. The last attribute we have defined is **enabled** which enables our shipping method to be used for shipping costs.

Let's define now our method *init* so that we can use our settings:

```
// ...
/**
 * Init your settings
 *
 * @access public
 * @return void
 */
function init() {
  $this->init_form_fields();
  $this->init_settings();
  // Save settings in admin if you have any defined
  add_action( 'woocommerce_update_options_shipping_' . $this->id,
    array( $this, 'process_admin_options' ) );
```

```
14  }
15  // ...
```

In this method we are calling another method *init_form_fields* that is used to set our shipping settings' fields. We are also calling the method *init_settings* to get all our settings that we can use when calculating the shipping cost.

Finally we are hooking the method *process_admin_options* so that all our fields values are saved to database in the admin area.

So, how to define our fields for this shipping method?

Shipping Settings' Fields

We will construct an array of field definitions and set it to our attribute **#form_fields**. Let's define our form fields:

```
1   // ...
2   /**
3    * Settings Fields
4    * @return void
5    */
6   function init_form_fields() {
7     $this->form_fields = array(
8      'enable' => array(
9         'title' => __( 'Enable', 'woodev_shipping' ),
10        'type' => 'checkbox',
11        'description' => __( 'Enable this shipping.', 'woodev_shipping' ),
12        'default' => 'no'
13        ),
14    );
15
16  }
17  // ...
```

We have set only one field. The name (id) of our field is **enable**. Everything else is self explanatory so I will not go into that.

Now that we have our field set, we can enable our shipping method within the admin area. To reflect that setting, we need to modify our contructor method so that our attribute **enabled** is getting set from our settings:

```
// ...
class WOODEV_TISAK_Shipping extends WC_Shipping_Method {

/**
 * Constructor for your shipping class
 *
 * @access public
 * @return void
 */
 public function __construct() {
   $this->id        = 'woodev_tisak_shipping';
   $this->method_title        = __( 'TISAK Shipping', 'woodev_shipping' );
   $this->method_description        = __( 'TISAK Shipping Settings', 'woodev_shipping' \
);
   $this->title        = __("TISAK","woodev_shipping");
   $this->init();
   $this->enabled        = $this->settings["enable"];
 }
// ...
```

OK, now we have defined our shipping settings and our base class. We still have to define our method to calculate the shipping cost and set all the available countries.

Let's go with the second requirement.

Shipping Countries

We will now create our own custom function that will return an associative array where keys will be the countries ISO codes and values will be the corresponding zones. Place this function below our class definition.

This is a long list:

```
/**
 * Array with all country codes where this shipping method is shipping
 * @return array country codes with zones
*/

function woodev_shipping_countries(){
    return array(
        'BA' => 0,
        'XK' => 0,
        'ME' => 0,
```

```
11          'RS' => 0,
12          'SI' => 0,
13          'AT' => 1,
14          'BE' => 1,
15          'BG' => 1,
16          'CZ' => 1,
17          'DK' => 1,
18          'FR' => 1,
19          'DE' => 1,
20          'HU' => 1,
21          'IT' => 1,
22          'NL' => 1,
23          'PL' => 1,
24          'RO' => 1,
25          'SK' => 1,
26          'GB' => 1,
27          'EE' => 2,
28          'FI' => 2,
29          'GR' => 2,
30          'IE' => 2,
31          'LV' => 2,
32          'LT' => 2,
33          'LU' => 2,
34          'PT' => 2,
35          'ES' => 2,
36          'SE' => 2,
37          'AF' => 3,
38          'AL' => 3,
39          'DZ' => 3,
40          'AS' => 3,
41          'AD' => 3,
42          'AO' => 3,
43          'AI' => 3,
44          'AG' => 3,
45          'AR' => 3,
46          'AM' => 3,
47          'AW' => 3,
48          'AU' => 3,
49          'AZ' => 3,
50          'BS' => 3,
51          'BH' => 3,
52          'BD' => 3,
```

```
53          'BB' => 3,
54          'BY' => 3,
55          'BZ' => 3,
56          'BJ' => 3,
57          'BM' => 3,
58          'BT' => 3,
59          'BO' => 3,
60          'BQ' => 3,
61          'BW' => 3,
62          'BR' => 3,
63          'VG' => 3,
64          'BN' => 3,
65          'BF' => 3,
66          'BI' => 3,
67          'KH' => 3,
68          'CM' => 3,
69          'IC' => 3,
70          'CV' => 3,
71          'KY' => 3,
72          'CF' => 3,
73          'TD' => 3,
74          'CL' => 3,
75          'CN' => 3,
76          'CO' => 3,
77          'KM' => 3,
78          'CG' => 3,
79          'CD' => 3,
80          'CK' => 3,
81          'CR' => 3,
82          'CI' => 3,
83          'CW' => 3,
84          'CY' => 3,
85          'DJ' => 3,
86          'DM' => 3,
87          'DO' => 3,
88          'TL' => 3,
89          'EC' => 3,
90          'EG' => 3,
91          'SV' => 3,
92          'GQ' => 3,
93          'ER' => 3,
94          'ET' => 3,
```

```
95          'FO' => 3,
96          'FJ' => 3,
97          'PF' => 3,
98          'GA' => 3,
99          'GM' => 3,
100         'GE' => 3,
101         'GH' => 3,
102         'GI' => 3,
103         'GL' => 3,
104         'GD' => 3,
105         'GP' => 3,
106         'GU' => 3,
107         'GT' => 3,
108         'GN' => 3,
109         'GW' => 3,
110         'GY' => 3,
111         'HT' => 3,
112         'HN' => 3,
113         'HK' => 3,
114         'IS' => 3,
115         'IN' => 3,
116         'ID' => 3,
117         'IQ' => 3,
118         'IL' => 3,
119         'JM' => 3,
120         'JP' => 3,
121         'JO' => 3,
122         'KZ' => 3,
123         'KE' => 3,
124         'KI' => 3,
125         'KR' => 3,
126         'FM' => 3,
127         'KW' => 3,
128         'KG' => 3,
129         'LA' => 3,
130         'LB' => 3,
131         'LS' => 3,
132         'LR' => 3,
133         'LY' => 3,
134         'LI' => 3,
135         'MO' => 3,
136         'MK' => 3,
```

```
137             'MG' => 3,
138             'MW' => 3,
139             'MY' => 3,
140             'MV' => 3,
141             'ML' => 3,
142             'MT' => 3,
143             'MH' => 3,
144             'MQ' => 3,
145             'MR' => 3,
146             'MU' => 3,
147             'YT' => 3,
148             'MX' => 3,
149             'MD' => 3,
150             'MC' => 3,
151             'MN' => 3,
152             'MS' => 3,
153             'MA' => 3,
154             'MZ' => 3,
155             'MM' => 3,
156             'NA' => 3,
157             'NP' => 3,
158             'KN' => 3,
159             'NC' => 3,
160             'NZ' => 3,
161             'NI' => 3,
162             'NE' => 3,
163             'NG' => 3,
164             'MP' => 3,
165             'NO' => 3,
166             'OM' => 3,
167             'PK' => 3,
168             'PW' => 3,
169             'PA' => 3,
170             'PG' => 3,
171             'PY' => 3,
172             'PE' => 3,
173             'PH' => 3,
174             'PR' => 3,
175             'QA' => 3,
176             'RE' => 3,
177             'RU' => 3,
178             'RW' => 3,
```

```
179             'WS' => 3,
180             'SM' => 3,
181             'SA' => 3,
182             'SN' => 3,
183             'SC' => 3,
184             'SL' => 3,
185             'SG' => 3,
186             'SB' => 3,
187             'ZA' => 3,
188             'LK' => 3,
189             'BL' => 3,
190             'LC' => 3,
191             'SX' => 3,
192             'MF' => 3,
193             'VC' => 3,
194             'SR' => 3,
195             'SZ' => 3,
196             'CH' => 3,
197             'TW' => 3,
198             'TJ' => 3,
199             'TZ' => 3,
200             'TH' => 3,
201             'TG' => 3,
202             'TO' => 3,
203             'TT' => 3,
204             'TN' => 3,
205             'TR' => 3,
206             'TM' => 3,
207             'TC' => 3,
208             'TV' => 3,
209             'UG' => 3,
210             'UA' => 3,
211             'AE' => 3,
212             'UY' => 3,
213             'UZ' => 3,
214             'VU' => 3,
215             'VE' => 3,
216             'VN' => 3,
217             'VG' => 3,
218             'WF' => 3,
219             'YE' => 3,
220             'ZM' => 3,
```

```
221         'ZW' => 3,
222         );
223 }
```

Now that we have our function for countries, we can assign the country codes to the attribute **$countries** and set the attribute **$availability** to *including*. We will do that in our method *init*:

```
1  function init() {
2          $this->init_form_fields();
3          $this->init_settings();
4          $country_codes = woodev_shipping_countries();
5          $this->countries = array_keys( $country_codes );
6          $this->countries[] = 'HR';
7          $this->availability = 'including';
8
9      // Save settings in admin if you have any defined
10         add_action( 'woocommerce_update_options_shipping_' . $this->id,
11                 array( $this, 'process_admin_options' ) );
12 }
```

First, we get the array of countries with their zones. After that, we get only the keys in that array which are actually only the country codes. That array of keys will be then set to the attribute **$countries**.

Availability is then set to *including* so that this shipping method will be available only for those countries we have set.

Getting the Right Package

Let's define a method that will be used to get the right package from our dimensions, maximum length, width or height. This method will return false if we could not get any package from provided parameters.

```php
/**
 * Get the package for TISAK
 * @param  number $dimension     Dimension volume (length x width x height)
 * @param  number $maximumLength Maximum length from all items in our cart
 * @param  number $maximumWidth  Maximum width from all items in our cart
 * @param  number $maximumHeight Maximum height from all items in our cart
 * @return mixed                 Returns false if there is no package
 *                                                                    that can be selected
 */
public function getTisakPackage(
        $dimension,
        $maximumLength,
        $maximumWidth,
        $maximumHeight ) {

                $packageS = 20 * 20 * 15;
                $packageM = 30 * 20 * 20;
                $packageL = 40 * 30 * 15;

                if( $maximumLength > 40 ){
                    return false;
                }

                if( $maximumWidth > 30 ){
                    return false;
                }

                if( $maximumHeight > 20 ) {
                    return false;
                }

                if( $dimension <= $packageS ) {

                  return 's';

                } elseif ( $dimension <= $packageM ) {

                  return 'm';

                } elseif ( $dimension <= $packageL ) {

                  return 'l';
```

```
43
44                  }
45
46                  return false;
47  }
```

In this method we are passing the mentioned parameters and define the maximum dimensions for each package (**$packageS**, **$packageM** and **$packageL**). If any maximum parameter is higher than the maximum possible (from any package) then we do not have a shippable package.

Getting the Price for Package

We have to create a method or methods to get the right price for the selected package based on the country where we need to ship.

Since each package has its own price for Croatia and a different one for every zone we will need to define two different methods for national and international shipping.

Shipping value for Croatia is easily calculated for each package since there are no zones. Here is the method for getting the price for each package when shipping in Croatia.

```
1   public function croatia_shipping_value( $package ){
2     switch ( $package ) {
3       case 's':
4         return 15;
5         break;
6       case 'm':
7         return 20;
8         break;
9       default:
10        return 25;
11        break;
12    }
13  }
```

By receiving the package we are returning the price for each package size.

To calculate the shipping price for international shipping we will have to develop helper methods that will be use to decouple our code into smaller parts and thus make it more maintainable and readable.

The first helper method will be for getting the price by zones:

```php
/**
* Returns the array with prices for a zone
* @param  number $zone zone number
* @return array       prices for packages
*/
public function get_zone_prices( $zone ) {

        $zonePrice = array(
            0 => array(
                's' => 95,
                'm' => 105,
                'l' => 135),
            1 => array(
                's' => 220,
                'm' => 250,
                'l' => 275),
            2 => array(
                's' => 260,
                'm' => 300,
                'l' => 360),
            3 => array(
                's' => 470,
                'm' => 550,
                'l' => 790),
            );
        return $zonePrice[ $zone ];
}
```

In this method we are receiving a zone as a number. We have defined an array with prices for each package for every zone. We are returning only one set of package prizes by using the zone number as the index of the array.

So now we have the method to retrieve all package prizes for a particular zone. We also have all countries with their zones assigned in the method **shipping_countries()**.

That are actually two helper methods that can be combined to get the right prize for the country and the package. Let's now create our method to calculate the shipping price when shipping internationally.

```php
/**
* Returns the prices for the provided country and package
* @param  string $country ISO country code
* @param  string $package
* @return number          Price for the package and country
*/
public function international_shipping_value( $country, $package ){
        $countries = woodev_shipping_countries();

        if( ! isset( $countries[ $country ] ) ) {
            return false;
        }

        $countryZone = (int) $countries[ $country ];

        $packagesFromZone = $this->get_zone_prices( $countryZone );

        $price = $packagesFromZone[ strtolower( $package ) ];

        return $price;
}
```

In this method we will pass two parameters: country code and package size. We will get all the countries from our method **shipping_countries()** and check if the passed country code is in that array of countries.

Afterwards, if the country is supported by our shipping method, we will get the zone that is assigned to that country by passing the country code to the array of countries. Since the country code is the **key** in the array and the zone is the **value** array, we will receive the **value** and that is the zone we require.

Once that is done, we are getting all the package prices for that specific zone using the method **get_zone_prices**. Once our prices with package sizes are returned for the passed zone, we are getting the specific prize for the package size we have passed to this method. The last thing to do is to return the price.

At this point we have our method to calculate the cost of the shipping for any country to which **TISAK** ships. We have our method to get the right package by dimensions and weight. We now need to combine all of those methods in the method **calculate_shipping** which WooCommerce uses to get the cost of that shipping.

Calculating the Shipping Cost

The first step is to define some variables which will be used to calculate the shipping cost.

```
/**
 * calculate_shipping function.
 *
 * @access public
 * @param mixed $package
 * @return void
 */
public function calculate_shipping( $package ) {

    $cost = 0;
    $weight = 0;
    $currency = get_woocommerce_currency();
    $maximumLength = 0;
    $maximumHeight = 0;
    $maximumWidth = 0;

    if( $currency != 'HRK' ){
      return false;
    }

    $dimensions = 0;

            ...
```

Variable **$weight** will be used to contain the total weight of the package.

We are getting the currency used in WooCommerce by using the function *get_woocommerce_currency()*. I am here checking if the currency is **Croatian Kuna** and I return false it is not. So if WooCoommerce uses another currency this shipping will never be displayed as available at the checkout page.

This does not have to be in your case. Or even in this case. Since all the prices here are calculated and set as they would be in **Croatian Kuna**, we could have a *flag* that is true if the currency is not **Croatian Kuna**. We will then use a conversion function or API to convert the total shipping cost into your currency.

If you are following this article by copying the code, I suggest you to not use that part where I return false if it is not HRK. Or you can set your WooCommerce currency to HRK to see the shipping method.

In this method a parameter **$package** is passed. This parameter holds all the items we have in our cart and all our shipping definitions on the checkout page. We need to see the dimensions of all those items in the package and calculate the dimensions.

```
...
foreach ( $package['contents'] as $item_id => $values )
{
  $_product = $values['data'];
  $weight = $weight + $_product->get_weight() * $values['quantity'];
  $width = floatval( wc_get_dimension( $_product->get_width(), 'cm' ) );
  if( $maximumWidth < $width ) {
      $maximumWidth = $width;
  }

  $height = floatval( wc_get_dimension( $_product->get_height(), 'cm' ) );
  if( $maximumHeight < $height ) {
      $maximumHeight = $height;
  }

  $length = floatval( wc_get_dimension( $_product->get_length(), 'cm' ) );
  if( $maximumLength < $length ) {
      $maximumLength = $length;
  }

  $dimensions = $dimensions + (( $length * $values['quantity']) * $width * $heig\
ht );
}
...
```

On the first line are getting the value the product object for each item. Then we are calculating the weight which we add to the total weight in our variable **$weight**.

After that we are getting the width of this product and we are converting it in the unit **cm** since all our methods and definitions are using **cm** for dimensions.

We are also assigning that width to the **$maximumWidth** if that width is greater then the current maximum width.

The same logic is used for both height and length. The last part of the foreach loop is calculating the dimensions. Dimension is calculated in Length * Height * Width. Since we also can have more than one of those items, we have to use the parameter quantity also.

When the dimension for that item is calculated we are adding it up to the total dimensions which is our variable **$dimensions**.

We have our total weight and our total dimensions. Let's see if we can ship our package with **TISAK**:

```
...
$weight = wc_get_weight( $weight, 'kg');

if( $weight > 10 ){
return false;
}

$tisak_package = $this->getTisakPackage( $dimensions, $maximumLength, $maximumWi\
dth, $maximumHeight );

if( $tisak_package == false ) {
   return false;
}
...
```

Since our maximum weight which can be shipped is 10 **kg** we have to set the weight to **kg** unit. Once that is done we check if the weight is under **10kg**. If not, we return false and this shipping method is not displayed.

We are also getting the package by passing the total dimensions variable, maximum length, width and height. If we do not receive a package value but we receive **false**. Then it means that the dimensions we have passed are exceeding all the packages.

Let's now define how to calculate the cost if everything until that point went well:

```
...
if( $package['destination']['country'] == 'HR' ) {

        $cost = $this->croatia_shipping_value( $tisak_package );

}else{

        $cost = $this->international_shipping_value( $package['destination']['co\
untry'], $tisak_package );
}

if( $cost == false ) {
        return false;
}
...
```

We check the destination for that package. If the country code is **HR** then it means that this package will be shipped inside Croatia. If that is the case, we are using our method **croatia_shipping_value** to get the cost.

If the package is going to be shipped internationally, then we will use our other method **international_shipping_value**.

Once that is done and if the variable **$cost** is false, then we return false and the shipping method will not be displayed.

The last part of this method is to add the rate:

```
...
$rate = array(
    'id' => $this->id,
    'label' => $this->title,
    'cost' => $cost,
    //'calc_tax' => 'per_item'
);

// Register the rate
$this->add_rate( $rate );
}
```

Adding our Shipping Class to all other Shipping Methods

To add the shipping method to other shipping method (register it), we need to use the filter *woocommerce_shipping_methods*. That filters passes an array of all registered Shipping method.

By adding our shipping id as a **key** inside that array while the value is the name of our Shipping class, we will register it.

```
/**
 * Adding our Shipping class to methods
 * @param  array $methods
 * @return array
 */
function woodev_add_shipping_methods( $methods ) {
        $methods['woodev_tisak_shipping'] = 'WOODEV_TISAK_Shipping';
        return $methods;
}
add_filter( 'woocommerce_shipping_methods', 'woodev_add_shipping_methods' );
```

That's it! You now have a shipping method that you can use in your own WooCommerce project. Let's learn how to add or remove shipping zones programmatically.

On Activation

In this chapter we will code what happens when our plugin is activated. Let's first create the base. We will hook our own function to the activation process of our plugin and we will check if there is WooCommerce and if WooCommerce is of 2.6 or above:

```
register_activation_hook( __FILE__, 'woodev_shipping_activation' );
function woodev_shipping_activation(){
 if( class_exists( 'WooCommerce' ) ){
        $wc_version = WC()->version;
        if( version_compare( $wc_version, '2.6', '>' ) ){
                // Here will our code be
        }
 }
}
```

First, we will get all the existing zones. We will retrieve them with the new class **Shipping_Zones**. We can then check if our zones are already there or not.

```
// Shipping Zone are available, add them
// Get existing ones
$available_zones = WC_Shipping_Zones::get_zones();
// Get all shipping countries from our shipping
$shipping_countries = woodev_shipping_countries();

// Get all WC Countries
$all_countries  = WC()->countries->get_countries();
//Array to store available names
$available_zones_names = array();
// Add each existing zone name into our array
foreach ($available_zones as $zone ) {
        if( !in_array( $zone['zone_name'], $available_zones_names ) ) {
                $available_zones_names[] = $zone['zone_name'];
        }
}
```

We have prepared some variables that we will use later. Those are **all_countries** and **shipping_countries**. The variable **available_zones_names** will hold only the names of all zones. We will check againts that variable if our zones are already in the database.

Zone: TISAK Croatia

```
// Check if our zone 'TISAK Croatia' is already there
if( ! in_array( 'TISAK Croatia', $available_zones_names ) ){


        // Instantiate a new shipping zone
        $new_zone_cro = new WC_Shipping_Zone();
        $new_zone_cro->set_zone_name( 'TISAK Croatia');

        // Add Croatia as location
        $new_zone_cro->add_location( 'HR', 'country' );

        // Save the zone, if non existent it will create a new zone
        $new_zone_cro->save();
        // Add our shipping method to that zone
        $new_zone_cro->add_shipping_method( 'woodev_tisak_shipping' );
}
```

So, we now check if there is a zone named *TISAK Croatia*. If not, we are instantiating a new object of **WC_Shipping_Zone**. Once that is done, we are setting a zone name to the object. We are also adding a new location to it by the country **code** and by the type of location which is **country**.

Then we save the zone. The zone is then created and saved to database with all those settings. After that, we just add our shipping method to that zone.

We will go through all of it for each of the other international zones. The only difference will be when adding locations.

Zone: TISAK Zone 0

```
if( ! in_array( 'TISAK Zone 0', $available_zones_names ) ){

        $new_zone = new WC_Shipping_Zone();
        $new_zone->set_zone_name( 'TISAK Zone 0' );
        // for each countries of TISAK, if the country is in the specific zone, add it.
        foreach ($shipping_countries as $code => $zone) {
                // If the country is not zone 0, escape it
                if( $zone != 0 ){
                        continue;
                }
                // If the country is not in WooCommerce countries, escape it
                if( ! isset( $all_countries[ $code ] ) ) {
                        continue;
                }

```

```
                // If not escaped, add it
                $new_zone->add_location( $code, 'country' );

        }
        $new_zone->save();
        $new_zone->add_shipping_method( 'woodev_tisak_shipping' );
}
```

When adding new locations, we are going through all our shipping countries. We are checking the current zone for which we want to add locations.

If the current country is not in that zone, we escape it. If the country is not in the WooCommerce countries, then we are also escaping it. If the country is not escaped, then we add it.

Here is the code for the other three zones:

Zone: TISAK Zone 1

```
if( ! in_array( 'TISAK Zone 1', $available_zones_names ) ){

        $new_zone = new WC_Shipping_Zone();
        $new_zone->set_zone_name( 'TISAK Zone 1' );
        foreach ($shipping_countries as $code => $zone) {
                if( $zone != 1 ){
                        continue;
                }
                if( ! isset( $all_countries[ $code ] ) ) {
                        continue;
                }
                $new_zone->add_location( $code, 'country' );

        }
        $new_zone->save();
        $new_zone->add_shipping_method( 'woodev_tisak_shipping' );
}
```

Zone: TISAK Zone 2

```
if( ! in_array( 'TISAK Zone 2', $available_zones_names ) ){

        $new_zone = new WC_Shipping_Zone();
        $new_zone->set_zone_name( 'TISAK Zone 2' );
        foreach ($shipping_countries as $code => $zone) {
                if( $zone != 2 ){
                        continue;
                }
                if( ! isset( $all_countries[ $code ] ) ) {
                        continue;
                }
                $new_zone->add_location( $code, 'country' );

        }
        $new_zone->save();
        $new_zone->add_shipping_method( 'woodev_tisak_shipping' );
}
```

Zone: TISAK Zone 3

```
if( ! in_array( 'TISAK Zone 3', $available_zones_names ) ){

        $new_zone = new WC_Shipping_Zone( $zone );
        $new_zone->set_zone_name( 'TISAK Zone 3' );
        foreach ($shipping_countries as $code => $zone) {
                if( $zone != 3 ){
                        continue;
                }
                if( ! isset( $all_countries[ $code ] ) ) {
                        continue;
                }
                $new_zone->add_location( $code, 'country' );

        }
        $new_zone->save();
        $new_zone->add_shipping_method( 'woodev_tisak_shipping' );
}
```

On Deactivation

For the deactivation function, we will also go through all the existing zones and if any of those zones is one from TISAK, then we will delete it.

```
function woodev_tisak_deactivate(){
        if( class_exists( 'WooCommerce' ) ){
                $wc_version = WC()->version;
                if( version_compare( $wc_version, '2.6', '>' ) ){
                        $available_zones = WC_Shipping_Zones::get_zones();
                        $woodev_zones = array(
                                'TISAK Croatia',
                                'TISAK Zone 0',
                                'TISAK Zone 1',
                                'TISAK Zone 2',
                                'TISAK Zone 3');
                        foreach ($available_zones as $zone ) {
                                if( in_array( $zone['zone_name'], $woodev_zones ) ) {
                                        $the_zone = new WC_Shipping_Zone( $zone['zone_id'] );
                                        $the_zone->delete();
                                }
                        }
                }
        }
}
register_deactivation_hook( __FILE__, 'woodev_tisak_deactivate' );
```

So, we are checking if the WooCommerce is of 2.6 or above. Then we are getting all the available zones. Once we go through all of them, we check if one of those zones is from our TISAK method. If it is, we instantiate the *WC_Shipping_Zone* with it's **zone_id** and delete it.

Custom Payment Gateway

WooCommerce has a bunch of free and premium plugins that add additional payment gateways. Some of them are really complex and some of them are actually pretty simple.

We will create a card processing payment gateway.

Payment Gateway API

Let's learn a bit more about creating custom payment gateways before we begin writing code. To start our payment gateway we will extend our class from an abstract class *WC_Payment_Gateway* that is located under **woocommerce/includes/abstracts/abstract-wc-payment-gateway.php**.

This class inhertis the same attributes and methods as the *WC_Shipping_Method* since they both are extended from the class *WC_Settings_API*.

To learn more about the Payment Gateway API, you can visit https://docs.woothemes.com/document/payment-gateway-api/[2].

Here you can learn some of the main things you need to know.

Types

There are four different types of payment gateways:

- **Form based**: user is redirected upon clicking the submit button to another website for processing payment (similar to PayPal Standard),
- **iFrame based**: the gateway seems to be loaded on the same page but it is actually an iframe loading another website where user can pay
- **Direct**: payment fields are shown directly on the checkout page and the payment is made when user places the order
- **Offline**: no online payment, for example: cheque or bank transfer which has to be completed manually.

We will create a mixed payment gateway where the main fields will be displayed as an iframe but the payment will be initialized on our part by the API.

[2] https://docs.woothemes.com/document/payment-gateway-api/

Required methods and attributes

Some of the methods and attributes defined by *WC_Settings_API* or *WC_Payment_Gateway* are required and must be defined in every custom payment gateway. Some of them were already described in the previous chapter **Custom Shipping Method**.

The required attributes are:

- **$id**: Unique ID for the custom gateway
- **$method_title**: Title that will show in the admin area
- **$title**: Title that will be shown on the checkout page
- **$method_description**: Used to describe our payment gateway in the admin area
- **$availablity**: Indicator if the payment gateway is available or not
- **$enabled**: indicator if this payment gateway is enabled or not
- **$supports**: array that contains what the payment gateway supports. This will define if the payment gateway will show. The basic values can be: *products*, *refunds*, *default_credit_card_form*.

Some recommended but not required attributes:

- **$icon**: URL to the icon/image of the payment gateway
- **$has_fields**: True if we want to show a form for processing a direct payment gateway

The required methods are:

- **__construct**: the constructor method will set the id and some other important attribues
- **init_form_fields**: used to define all the form fields for our payment gateway
- **init_settings**: get all the fields populated with values from the database
- **process_payment**: all the logic and functionality to process the payment of our payment gateway

Braintree

Braintree offers a wide range of options for accepting payments in mobile apps or online on your website. They even provide easy integrations with PayPal and Venmo so you can widen your payments options by using only one company and API.

Since we are creating a card processing payment gateway we are interested only in payments options that will accept credit cards. To learn which credit cards are available in Braintree you can visit this address: https://www.braintreepayments.com/payment-methods/accept-credit-cards[3].

To learn more about configuring them and integrating them you can visit this address: https://developers.braintreepa cards/overview?_ga=1.130665295.1769098426.1465365969[4].

[3] https://www.braintreepayments.com/payment-methods/accept-credit-cards

[4] https://developers.braintreepayments.com/guides/credit-cards/overview?_ga=1.130665295.1769098426.1465365969

Testing Braintree

To test Braintree payment options on our local server and test environment you should sign up for a sandbox account here: https://www.braintreepayments.com/sandbox[5].

This is required to do because you will have to have a merchant id, public and secret keys for the Braintree API to work.

Getting API Credentials

I will not go into further detail about getting the API credentials for Braintree because they are well described here: https://articles.braintreepayments.com/control-panel/important-gateway-credentials#api-credentials[6].

For the busy ones:

1. Login into production/sandbox control panel on Braintree
2. Go to **Account** > **My user**
3. Under **API Keys**, **Tokenization Keys**, **Encryption Keys**, click **View Authorizations** and if there are no API keys, create one by clicking **Generate New API Key**
4. Click **View** under the **Private Key** to see your public and private keys, merchant ID and environment

PCI Compliance - be legit the easy way

PCI is a security standard for organizations that handle credit cards processing. Your store must have some level of PCI Compliance to process credit cards of some brand (exp. Master Card, Visa).

This could be a hassle to implement on your own store, especially if you are looking for something simple. So, to make it easy for store owners, other companies have implemented several ways for being PCI compliant.

PayPal has a simple way of just redirecting users to their website to process payments. This way, store owners do not have to worry about being PCI Compliant since PayPal does it all for them.

With Braintree there are two simple ways. They are not redirecting users but they are offering two different ways into implementing their own fields as iframes on your website thus being PCI compliant.

The first is **Dropin** which creates the whole form on your website. This does not leave us with much customization options but it is the easiest way to have a card processing form that works like a charm.

[5] https://www.braintreepayments.com/sandbox

[6] https://articles.braintreepayments.com/control-panel/important-gateway-credentials#api-credentials

The second one is **Hosted Fields** which "hosts" our fields, each separately as an iframe and with some simple JavaScript enables us to be PCI compilant and have a fully customizable form on our side.

Both of them enable us to do the whole payment logic on our server side. We will go with the second option **Hosted Fields**.

To learn more about them, you can visit this link: https://developers.braintreepayments.com/guides/hosted-fields/overview/javascript/v2[7].

Braintree PHP SDK

We need the Braintree SDK (Software Development Kit) to work with the Braintree API. Since we are working with WordPress and WooCommerce which operate on PHP, we need the PHP SDK.

You can download it here: https://developers.braintreepayments.com/start/hello-server/php[8]. For now, just download it so that we can later include it in our code.

Payment Class Base

Before we begin the code for our class we will create a plugin that will hold our code for the payment gateway.

Go to the **wp-content/plugins** folder and create a new folder. I will name mine *woodev_payment_gateway* and also create a file with the same name under that folder *woodev_payment_gateway.php*.

Add this at the beginning of that file and customize it to your needs:

```
1  <?php
2  /*
3  Plugin Name: WooCommerce For Developers Payment Gateway
4  Plugin URI: https://leanpub.com/woodev
5  Description: A Custom Payment Gateway done in the "WooCommerce for Developers" e\
6  Book
7  Version: 1.0
8  Author: Igor BeniÄ‡
9  Author URI: http://ibenic.com
10 textdomain: woodev_payment
11 */
```

Let's first define our constants that will be used when including files or enqueueing scripts or styles. We will also create the base class in which we will define later on our methods and attributes for this payment gateway.

[7] https://developers.braintreepayments.com/guides/hosted-fields/overview/javascript/v2
[8] https://developers.braintreepayments.com/start/hello-server/php

```
define( 'WOODEV_PAYMENT_DIR', plugin_dir_path( __FILE__ ));
define( 'WOODEV_PAYMENT_URI', plugin_dir_url( __FILE__ ));

add_action( 'plugins_loaded', 'woodev_payment_gateway' );

function woodev_payment_gateway() {
        class WooDev_Braintree extends WC_Payment_Gateway {
        }
}
```

Including Payment Method in WooCommerce

Before we begin with our payment gateway class, let's add that to the woocommerce filter for gateways so that WooCommerce can include it. Add this at the bottom:

```
function woodev_add_gateway_class( $methods ) {
        $methods[] = 'WooDev_Braintree';
        return $methods;
}

add_filter( 'woocommerce_payment_gateways', 'woodev_add_gateway_class' );
```

Let's define every script and style we need and also the includes folder that will hold our Braintree PHP SDK before we define our class.

Assets

Create a folder **assets** inside our plugin's folder. Inside that folder create two other folders **css** and **js**.

- assets/
- css/
- js/

CSS

Create a new file *braintree.css* inside the folder **css** and add this inside that file which holds some basic styles. Feel free to change it for you needs:

```
.braintree-input {
        display: block;
    height: 33px;
    width: 100%;
    padding: 3px 6px;
    background-color: white;
    color: #666;
    border: none;
    border-radius: 5px;
    margin-bottom: 12px;
}

.braintree-input-small {
        display: inline-block;
        width: 30%;
        vertical-align: middle;
}
.woocommerce-checkout #payment #woodev_braintree-cc-form div.form-row {
        padding: 3px;
}
```

JavaScript

Create a new file *braintree.js* inside the folder **js** and add the following code inside that file. This is only a wrapper that will hold our JavaScript later on.

```
(function($){
        $(document).ready(function(){

        })
})(jQuery);
```

Includes

Now create a new folder **inc** inside our plugin's folder. Under that folder create a new folder **braintree** and paste the files from the Braintree SDK you have downloaded. The structure of the braintree folder should be something like this:

- braintree/
- Braintree/
- ssl/

- autoload.php
- Braintree.php

Now let's just modify the *autoload.php* so that the autoload will load the PHP files from the right directory. Open *autoload.php* and change:

```
$fileName = dirname(__DIR__) . '/lib/';
```

to

```
$fileName = dirname(__DIR__) . '/braintree/';
```

Now we are prepared for some real stuff! Let's begin:)

Constructor Method

First, we will define some basic and requires attributes so that our payment gateway can be seen:

```
...
class WooDev_Braintree extends WC_Payment_Gateway {

        /**
         * Constructor for your shipping class
         *
         * @access public
         * @return void
         */
        public function __construct() {
            $this->id                           = 'woodev_braintree';
            $this->method_title                 = __( 'WooDev Braintree', 'woodev_payment' );
            $this->method_description           = __( 'Braintree Payment Gateway made for WooDev\
 eBook', 'woodev_payment' );
            $this->title                        = __( 'Braintree', 'woodev_payment' );
                ...
        } // End of Constructor
        ...
} // End of payment class
...
```

We have defined the ID of our payment gateway, the gateway titles and description. Nothing too special.

We will now define two attributes required to show the fields of our payment method and also to include it at the checkout of our products:

```
$this->has_fields = true;

$this->supports = array(
        'products',
        'default_credit_card_form' );
```

Now we will add the inherited methods from the *WC_Settings_API*:

```
// Load the settings.
$this->init_form_fields();
$this->init_settings();
$this->enabled                = $this->get_option('enabled');
```

The fields are now loaded and also the settings from the database. We are then assigning the option **enabled** to define if our payment gateway is enabled or not.

The last part of our constructor method will have the required hooks for enqueueing scripts and styles in the header, footer and also right under our credit card form (this could be also loaded in footer if wanted). We will also define a hook that will be used to save our settings for this payment gateway.

```
        add_action( 'woocommerce_credit_card_form_end', array( $this, 'add_braintree_sc\
ript' ));
        add_filter( 'woocommerce_credit_card_form_fields', array( $this, 'braintree_fie\
lds' ));
        add_action( 'wp_enqueue_scripts', array( $this, 'braintree_style') );

        // Save settings
        if ( is_admin() ) {

                add_action( 'woocommerce_update_options_payment_gateways_' . $this->id, array(\
 $this, 'process_admin_options' ) );

        }
} // End of Constructor
...
```

All these methods will be defined as we progress in this chapter. Hooks are used for:

- **woocommerce_credit_card_form_end**: Add HTML or anything else at the end of the credit card form
- **woocommerce_credit_card_form_fields**: Filters the fields of the credit card form
- **wp_enqueue_scripts**: Enqueue the scripts and styles

Settings Fields

Our contructor method calls the method *init_form_fields()* which is used to define the fields for this payment gateway.

Our fields will be:

- **enabled**: check to enable or disable this payment gateway
- **mode**: select sandbox or production environment
- **merchant_id**: Braintree Merchant ID
- **public_key**: Braintree Public Key
- **secret_key**: Braintree Secret Key

Add this to our class:

```
public function init_form_fields() {
        $this->form_fields = array(
                'enabled' => array(
                        'title' => __( 'Enable', 'woodev_payment' ),
                        'type' => 'checkbox',
                        'label' => __( 'Enable WooDev Braintree', 'woodev_payment' ),
                        'default' => 'yes'
                ),
                'mode' => array(
                        'title' => __( 'Mode', 'woodev_payment' ),
                        'type' => 'select',
                        'default' => 'sandbox',
                        'options' => array(
                                'sandbox' => 'Sandbox',
                                'production' => 'Production (Live)')
                ),
                'merchant_id' => array(
                        'title' => __( 'Merchant ID', 'woodev_payment' ),
                        'type' => 'text',
                        'default' => ''
                ),
                'public_key' => array(
                        'title' => __( 'Public Key', 'woodev_payment' ),
                        'type' => 'text',
                        'default' => ''
                ),
                'secret_key' => array(
```

```
                        'title' => __( 'Secret Key', 'woodev_payment' ),
                        'type' => 'password',
                        'default' => ''
                ),
        );
}
```

Configuring Braintree

Since our fields are now defined we have define a method that will be used to start the Braintree API with our mode, merchant id, public and secret key. Add this to our class:

```
private function load_braintree_config(){
 require_once WOODEV_PAYMENT_DIR .'inc/braintree/Braintree.php';

 Braintree_Configuration::environment( $this->get_option('mode') );
 Braintree_Configuration::merchantId( $this->get_option('merchant_id') );
 Braintree_Configuration::publicKey( $this->get_option('public_key') );
 Braintree_Configuration::privateKey( $this->get_option('secret_key') );
}
```

This will configure our Braintree so that we can use all methods from the API.

Enqueueing Style and Scripts

For our Braintree API to function correctly, we also need some JavaScript. We will enqueue the scripts and also the style that will define how our hosted fields look:

```
public function braintree_style(){
        if ( ! is_checkout() || ! $this->is_available() ) {
                return;
        }
        wp_enqueue_script( 'woodev-braintree-js', 'https://js.braintreegateway.com/js/b\
raintree-2.24.1.min.js', array(), '1.0', true );
        wp_enqueue_style( 'woodev-braintree-style', WOODEV_PAYMENT_URI . '/assets/css/b\
raintree.css', array(), '1.0', 'screen' );
        wp_enqueue_script( 'woodev-braintree-custom-js', WOODEV_PAYMENT_URI . '/assets/\
js/braintree.js', array('jquery'), '1.0', true );
}
```

Before enqueueing, we are checking also if we are on the checkout page and if this payment gateway is available.

Credit Card Fields

Since Braintree Hosted Field are added as iframes, we will not use the classic form elements such as *input*. We need to define those as regulars **divs** so that our iframe can be appended to them. Add this to filter those fields:

```
public function braintree_fields( $fields ){

        $fields['card-number-field'] = '<div class="form-row form-row-wide">
                <label for="' . esc_attr( $this->id ) . '-card-number">' . __( 'Card Number', \
'woocommerce' ) . ' <span class="required">*</span></label>
                <div id="' . esc_attr( $this->id ) . '-card-number" class="braintree-input"></\
div>
                </div>';

        $fields['card-expiry-field'] = '<div class="form-row form-row-first">
                <label for="' . esc_attr( $this->id ) . '-card-expiry">' . __( 'Expiry (MM/YY)\
', 'woocommerce' ) . ' <span class="required">*</span></label>
                <div id="' . esc_attr( $this->id ) . '-card-expiry-month" class="braintree-inp\
ut braintree-input-small" ></div> / <div id="' . esc_attr( $this->id ) . '-card-\
expiry-year" class="braintree-input braintree-input-small"></div>
                </div>';

        $fields['card-cvc-field'] = '<div class="form-row form-row-last">
                <label for="' . esc_attr( $this->id ) . '-card-cvc">' . __( 'Card Code', 'wooc\
ommerce' ) . ' <span class="required">*</span></label>
                <div id="' . esc_attr( $this->id ) . '-card-cvc" class="braintree-input braint\
ree-input-small"></div>
                </div>';



        return $fields;
}
```

If you look closely, our fields are actually divs with the IDs generated from this payment gateway:

- **#woodev_braintree-card-number**
- **#woodev_braintree-card-expiry-month**
- **#woodev_braintree-card-expiry-year**
- **#woodev_braintree-card-cvc**

All those divs have the classes we have defined in our *braintree.css* for display. These IDs will be used in the script we will append at the end of the credit card form to include the hosted fields.

Credit Card Script

Braintree JavaScript will handle everything for us to be PCI compliant as we learned before. To learn more about how to handle hosted fields with Braintree JavaScript, visit this link: https://developers.braintreepayment fields/setup-and-integration/javascript/v2[9].

We hooked our method *braintree_script* to action **woocommerce_credit_card_form_end**. Let's add it:

```
public function add_braintree_script( $id ){
        if( $this->id == $id ){
                $this->load_braintree_config();
                $clientToken = Braintree_ClientToken::generate();
                ?>
                <script>
                (function($){
                        $(document).ready(function(){
                                var $formCheckout = $("form.checkout"),
                                        $formCheckoutID = $formCheckout.attr("id");

                                if( ! $formCheckoutID ){
                                        $formCheckout.attr("id", "checkout-form");
                                }

                                braintree.setup("<?php echo $clientToken; ?>", "custom", {
                                id: "checkout-form",
                                hostedFields: {

                                                styles: {

                                                        // Styling element state
                                                        ":focus": {
                                                                "color": "blue"
                                                        },
                                                        ".valid": {
                                                                "color": "green"
                                                        },
```

[9] https://developers.braintreepayments.com/guides/hosted-fields/setup-and-integration/javascript/v2

```
                                                    ".invalid": {
                                                            "color": "red"
                                                    },

                                                },
                                    number: {
                                      selector: "#<?php echo $this->id; ?>-card-number"
                                    },
                                    cvv: {
                                      selector: "#<?php echo $this->id; ?>-card-cvc"
                                    },
                                    expirationMonth: {
                                                selector: "#<?php echo $this->id; ?>-card-expiry-m
                                        },
                                        expirationYear: {
                                              selector: "#<?php echo $this->id; ?>-card-expiry-y
                                        }
                                  }
                                });
                            });
                    })(jQuery);

                </script>
                    <?php
            }
}
```

Here we load our Braintree configuration and create a client token that is needed for our JavaScript to work.

Since our checkout form does not have an ID attribute which is needed for our hosted fields to work, we are creating a custom ID attribute and add it to the checkout form. We have named it **checkout-form**.

In setting our Braintree using the function *setup* we pass our client token, parameter '**custom**' to define that we work with hosted fields and our options object.

Object with options contains selector for each part of our credit form, the id of our form and also some styles that will show if the card is correct or invalid.

Customizing WooCommerce Form Submittion

Since Braintree JavaScript hijacks the form submittion because it has to process all those hosted fields and create a hashed string that will be appended to the form on which we called our Braintree

in the script before.

Since WooCommerce uses WC_AJAX to submit the form by default, that hashed string is not sent and we need to stop the AJAX from performing.

We will have to use the default form submittion. Due to the fact that our Braintree hijacks the form submittion, the element of type *submit* is ignored. That element has the name **woocommerce_checkout_place_order**.

This one has to be added as a hidden input field so that WooCommerce can trigger the checkout payment process if our payment gateway is selected.

Open our *braintree.js* that we have made before and add this new JavaScript inside our wrapper:

```
...
var $form = $("form.checkout");

$form.append("<input type='hidden' name='woocommerce_checkout_place_order' value\
='1' />");
$form.on('checkout_place_order_woodev_braintree', function(){
        return false;
});
...
```

This will append the hidden input to the form and also disable the AJAX process with the trigger **checkout_place_order_woodev_braintree** returning *false*.

Process Payment

Now we have customized our form submittion and added every script, style and functionality that we need. The last part is the method to process payment in our class. Let's start:

```
public function process_payment( $order_id ) {

        global $woocommerce;
        $order = new WC_Order( $order_id );


        $this->load_braintree_config();
        $payment_method_nonce = $_POST['payment_method_nonce'];
        $customerArray = array();

        ...

}
```

We are getting the order from the **$order_id** and load the braintree configuration. After that, we are getting the hashed string we have talked about which is appended to the form with the name **payment_method_nonce**.

We also start the customer array which will hold the customer information, if any. We will populate the customer array like this:

```
if( isset( $_POST['billing_first_name'] ) && $_POST['billing_first_name'] != '' \
){
        $customerArray['firstName'] = $_POST["billing_first_name"];
}

if( isset( $_POST['billing_last_name'] ) && $_POST['billing_last_name'] != '' ){
        $customerArray['lastName'] = $_POST["billing_last_name"];
}

if( isset( $_POST['billing_phone'] ) && $_POST['billing_phone'] != '' ){
        $customerArray['phone'] = $_POST["billing_phone"];
}

if( isset( $_POST['billing_email'] ) && $_POST['billing_email'] != '' ){
        $customerArray['email'] = $_POST["billing_email"];
}
```

Now that we have our hashed input and the customer information, we can create a sale using Braintree API:

```
$result = Braintree_Transaction::sale([
   'amount' => $order->order_total,
   'customer' => $customerArray,
   'paymentMethodNonce' => $payment_method_nonce,
   'options' => [
    'submitForSettlement' => True
   ]
]);
```

We have defined the option *submitForSettlement* as **true** so that our credit card is being processed automatically. Now we need to check if the transaction was succesfull to complete the payment, empty the cart and return the array with the URL address of the order.

We will also add a notice if for some reason, the transaction did happen but was declined. The last thing is to add some WooCommerce notices if there is any error while processing the payment. This is all done like this:

```
if( $result->success ){

        // Payment complete
        $order->payment_complete( $result->transaction->id );
        // Add order note
        $order->add_order_note( sprintf( __( '%s payment approved! Trnsaction ID: %s', \
'woocommerce' ), $this->title, $result->transaction->id ) );

        // Remove cart
        $woocommerce->cart->empty_cart();

        // Return thankyou redirect
        return array(
                'result' => 'success',
                'redirect' => $this->get_return_url( $order )
        );
} elseif( $result->transaction) {
        // Transaction was made but declined or failed
        $order->update_status('failed', sprintf( __( '%s payment declined! Trnsaction I\
D: %s', 'woocommerce' ), $this->title, $result->transaction->id ) );
} else {
        foreach($result->errors->deepAll() AS $error) {
          wc_add_notice(  $error->message . '(' . $error->code . ')', 'error' );
        }
}
return array(
 'result' => 'failure',
 'redirect' => ''
);
```

Congratulations! You have created a custom payment method!

Printing Order Details from the Admin area

Order details could be something that you would want to print in a custom solution. So, how to do it?

We will add a button to the “Actions” column in the order list. This button will then open a new window and automatically print the content of that window.

This new window will have the order details. The order details that we will print will be displayed through an action hook that WooCommerce uses in the Account page when viewing an order.

You can add this code in a theme also, but I would suggest that you add it to the a custom plugin.

Adding the Print Button

We will add a simple print button with an icon using the Dashicons. First, we need to hook to an action that is used for adding other buttons or HTML elements to it.

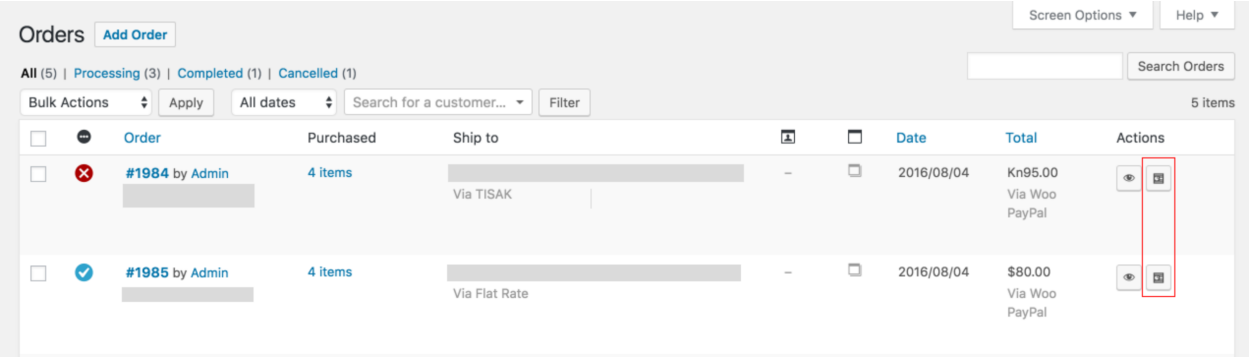
```
<?php

add_action('woocommerce_admin_order_actions_end', 'woodev_print_button' );
```

We have hooked a function **woodev_print_button**. Let’s define it:

```
<?php

function woodev_print_button( $order ) {
        $post_id = $order->get_id();
        echo '<a class="button dashicons dashicons-analytics order-print"
                        data-order="' . $post_id . '"
                        href="' . admin_url( 'post.php?post=' . $post_id . '&action=edit' ) . '"><
';

}
```

In this button we have added some new classes. Since WordPress is using Dashicons, we have applied the classes to show a Dashicon icon. Another class here is *print-order*. We will use that class to style our button so it looks good.

You should see this now:



Print Button

The attribute **data-order** will have the order ID so that we can access the ID value with JavaScript.

We will use JavaScript to get the Order details once we click on the print button.

Adding the Style and the JavaScript

The style that we will have to add is a simple one that will just make some small adjustments to the button. JavaScript, on the other hand, will be an AJAX call to WordPress that will get the HTML to print.

We will add them both in the footer. The action hook for the footer is:

```
<?php

add_action('admin_footer', 'woodev_print_footer');
```

Now we will also define that function:

```
<?php

function woodev_print_footer(){
        ?>
        <style>
                .button.order-print {
                        width:2em;
                        height:2em !important;
                        position: relative;
                }

                .button.order-print:before {
                        position: absolute;
                        left: 0;
                        right:0;
                        top:0;
                        bottom:0;
                        margin:auto;
                }
        </style>
        <script>

                jQuery(document).ready(function($){

                        $(".order-print").on('click', function(e){
                                e.preventDefault();
                                var $orderID = $(this).attr("data-order");

                                $.ajax({
                                        'url': ajaxurl,
                                        data: { action: 'woodev_print_order', order: $orderID },
                                        dataType: 'html',
                                        success: function( resp ){
                                                // Create a pop-up
                                                var popupWIndow = window.open('','','width=400,hei
                                                // Add HTML
                                                popupWIndow.document.write(resp);
                                                // Print
                                                popupWIndow.window.print();
                                                // Close the pop-up
                                                popupWIndow.window.close();
                                        }
```

```
                                });

                            });

                    });
            </script>
            <?php
}
```

The first CSS properties are to define the display of our button. The JavaScript is a bit more complex.

We are here attaching a click event to the button using the class *order-print*. On each click, we are preventing the browser from going on the URL defined in the attribute **href** (if any). Then we are getting the Order ID from the attribute **data-order**.

After that, we are creating an AJAX request. We are sending the AJAX request to the URL specified in the parameter *url*. The variable **ajaxurl** holds the URL to the file that is used to perform AJAX request. This file is located in *wp-admin/ajax-admin.php*.

WordPress itself creates that global variable on the top of the *head* element inside the admin area.

The data that are sending with the AJAX request are the **action** and the **order**. The **action** is used so that we can hook a function to it that will return the HTML. The **order** parameter will contain the Order ID.

We are also specifying that we request an HTML response. Once we get the response, we are creating a pop-up window and add that HTML to the pop-up. After that we are printing the content of that pop-up and then close the window.

Now we only have to create the HTML and return it.

Creating the HTML for AJAX Response

This is the last step in this chapter. In order to have the Order details displayed in the same way we would have them displayed in the Account page, we will have to use the same *action* hook that WooCommerce uses to display the details.

This action is *woocommerce_view_order*. When we pass a parameter **action** to the admin-ajax.php file, we will create a dynamic action hook that will contain also the value of our parameter. We need to hook a function to that action hook and we will do it like this:

```
<?php

add_action( 'wp_ajax_woodev_print_order', 'woodev_print_order_ajax' );
```

We have used only the action that is created if the user is logged in. If the user is not logged in, we would had to use the action:

```
wp_ajax_nopriv_YOURACTION
```

Since we are in the admin area, we don't need that. So, now we hooked a function to the action hook. Let's define it:

```
<?php

function woodev_print_order_ajax(){
        $order = $_GET['order'];

        // Remove the Order Again Button
        remove_action( 'woocommerce_order_details_after_order_table', 'woocommerce_orde\
r_again_button' );
        // Remove Link from Product
        add_filter('woocommerce_order_item_permalink', 'woodev_remove_permalink_from_pr\
oduct', 99);
        ?>
        <style type="text/css">
                * {
                        margin:0;
                        padding:0;
                        box-sizing: border-box;
                }

                h1 {
                        font-size: 2em;
                        margin-bottom:2cm;
                }

                h2 {
                        font-size: 1.5em
                }

                body {
                        margin: 1cm 2cm;
                }

                table {
                        width: 100%;
                        table-layout: fixed;
                        border-collapse: collapse;
                        margin-bottom:1cm;
```

```
                }

                tfoot {
                        text-align: right;
                }
                th,td {
                        padding:5pt;
                        border-bottom:1px solid;
                }

                .product-name {
                        text-align: left;
                }

                .product-total {
                        text-align: right;
                        width: 150px;
                }

                .shop_table.customer_details {
                        text-align: left;
                }

                .shop_table.customer_details th {
                        width: 150px;
                }

                .col2-set:after {
                        display: table;
                        clear: both;
                }

                .col2-set .col-2,
                .col2-set .col-1{
                        float:left;
                        width: 50%;
                }
        </style>
        <h1>Order #<?php echo $order; ?></h1>
        <?php
        do_action( 'woocommerce_view_order', $order );
```

```
        wp_die();
}


function woodev_remove_permalink_from_product( $permalink ) {
        return '';
}
```

The first thing we do here is getting out Order ID. We are also removing a function that is hooked on another action **woocommerce_order_details_after_order_table**. This action is called inside the file that shows the order details. That is in our main action **woocommerce_view_order**.

We are removing the button to link again from our print since it is not usable. After that we are also adding a filter that will remove the link from our product title. This is also not neccessary in a print file. We could make make a CSS property that would display the link after the Product title. I will leave that decision to you.

Once we have done all that, we are adding some simple CSS for printing purposes. You can easily add your own styles here.

The next thing is to display the main title of our document. After that, we are calling the action **woocommerce_view_order** and passing the Order ID we got earlier.

Function *wp_die()* is used to make sure other code does not get executed after that.

Congrats! You now have the ability to print the order details!

You could make a WooCommerce object from the ID using WC_Order class and make a custom HTML for your order details, but that is up to you.

Managing Stock with 3rd Party Service

If you are looking for various WooCommerce projects, one of the frequent ones will be managing their stock levels for products.

Some of those projects will require you to connect with a 3rd party service to handle stock levels. This may be a company that actually holds the physical products and manages the stock levels. This company has an API that provides you with enough information about the products.

This chapter will be a small representation on how to do it. Since there are different services with different APIs, I will show you how to do it on the WooCommerce level.

To have everything fully working, you will need to implement the API calls. Let's first create our simple plugin "WooDev Stock Manager".

Create a folder **woodev_stock** and add a file in there with the same name **woodev_stock.php**. After that create another folder *inc* inside and create two new files in there: *functions-api.php* and *functions-wc.php*.

The file *functions-api.php* will contain functions that will be used to interact with our Service/API. The second file, *functions-wc.php* will contain all functions related to WooCommerce.

Now that we are done, add this to the file *woodev_stock.php*.

```
<?php
/*
Plugin Name: WooDev Stock Manager
Plugin URI: https://leanpub.com/woodev
Description: An abstract plugin that can be used as a starting point to manage s\
tock
Version: 1.0
Author: Igor Benić
Author URI: http://ibenic.com
textdomain: woodev_stock
*/

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

```
if( ! class_exists('WooCommerce') ) {
        return;
}

define( 'WOODEV_STOCK_DIR', plugin_dir_path( __FILE__ ));
define( 'WOODEV_STOCK_URI', plugin_dir_url( __FILE__ ));
```

After you have added it, go to WordPress Admin dashboard under *Plugins* and activate it.

Starting our Stock Manager

We will have a final class that can't be extended any further. In this class we will include the needed files and also hook our functions.

Add this to the file *woodev_stock.php*:

```
final class WooDev_Stock {

        /**
         * We will include files and add hooks when WooCommerce Loads
         */
        public function __construct() {
                $this->includes();
                $this->hooks();
        }

        /**
         * Including files
         * @return void
         */
        public function includes() {
                require_once WOODEV_STOCK_DIR . 'inc/functions-api.php';
                require_once WOODEV_STOCK_DIR . 'inc/functions-wc.php';
        }

        /**
         * Adding Hooks
         * @return void
         */
        public function hooks() {
                // We will have the hooks defined here
        }
```

```
}

add_action( 'woocommerce_init', 'woodev_stock_load' );

/**
 * Load the Stock Manager after WooCommerce has been loaded
 * @return void
 */
function woodev_stock_load() {
        new WooDev_Stock();
}
```

By using the action **woocommerce_init**, we are making sure that the WooCommerce plugin is active and initiated. In the hooked function, we are initializing our class *WooDev_Stock*.

Creating the Product Inventory Field

To get the stock level of one of our products, we need an identifier of the product from the service. This Service ID will be used when requesting the stock level for that product.

We need a field for that. This *Service ID* field will be rendered in the **Inventory** tab when editing a product. Let's add a hook that will render this field.

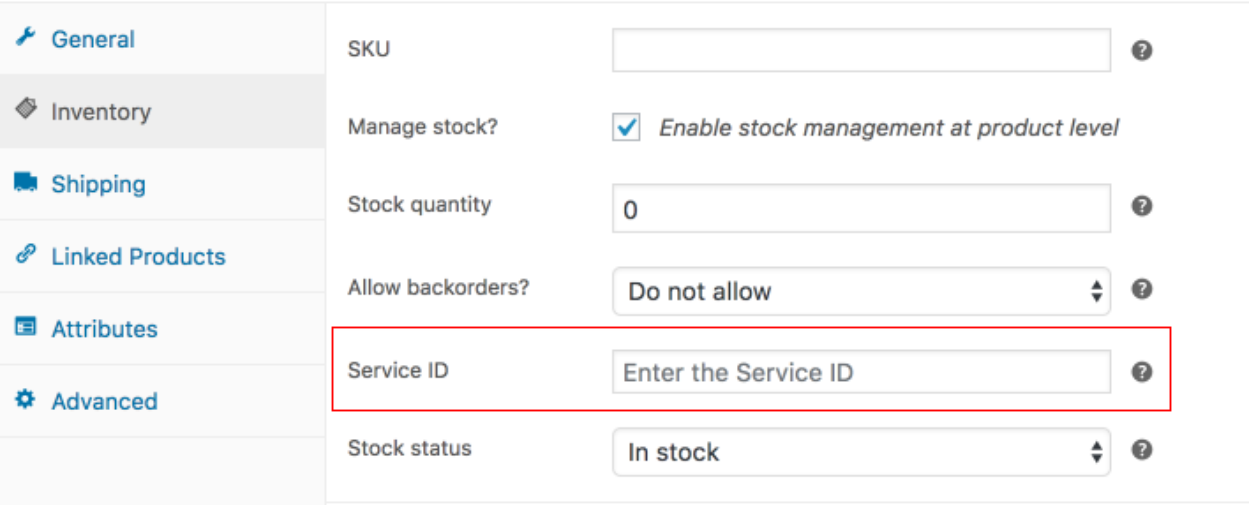
```
/**
 * Adding Hooks
 * @return void
 */
public function hooks() {

        // Creating the product field
        add_action( 'woocommerce_product_options_stock_fields', 'woodev_stock_product_s\
ervice_field' );

}
```

This action hook *woocommerce_product_options_stock_fields* is used to display fields that are only available when the *Manage Stock* field is checked.

Let's define the function *woodev_stock_product_service_field* inside the file *functions-wc.php*:

```php
<?php

/**
 * Hooked Functions for WooCommerce
 */

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

/**
 * Service Field for the product
 * @return void
 */
function woodev_stock_product_service_field() {
        $description = sanitize_text_field( 'The Service Product ID used to sync the st\
ock level with the Service' );
    $placeholder = sanitize_text_field( 'Enter the Service ID' );

    $args = array(
        'id'            => '_service_product_id',
        'label'         => sanitize_text_field( 'Service ID' ),
        'placeholder'   => $placeholder,
        'desc_tip'      => true,
        'description'   => $description,
    );

    woocommerce_wp_text_input( $args );
}
```

The key that we are using is *_service_product_id*. This will be also used as a meta key.



Service ID field

Saving the Field

We still can't save the data provided inside that field. Let's enable that by adding this first inside the file *woodev_stock.php*:

```
/**
 * Adding Hooks
 * @return void
 */
public function hooks() {

        // Creating the product field
        add_action( 'woocommerce_product_options_stock_fields', 'woodev_stock_product_s\
ervice_field' );

        // Saving the product field
        add_action( 'woocommerce_process_product_meta', 'woodev_stock_product_service_f\
ield_save' );

}
```

This action hook *woocommerce_process_product_meta* is a global action hook that will trigger everytime a product gets published or updated. Open the file *functions-wc.php* and add this new function:

```
/**
 * Saving the Product Service ID
 * @param  integer $post_id
 * @return void
 */
function woodev_stock_product_service_field_save( $post_id ) {

    if ( ! ( isset( $_POST['woocommerce_meta_nonce'], $_POST[ '_service_product_\
id' ] ) || wp_verify_nonce( sanitize_key( $_POST['woocommerce_meta_nonce'] ), 'w\
oocommerce_save_data' ) ) ) {
        return false;
    }

    $service_product_id = sanitize_text_field(
        wp_unslash( $_POST[ '_service_product_id' ] )
    );

    update_post_meta(
        $post_id,
        '_service_product_id',
        esc_attr( $service_product_id )
    );
}
```

On each call, we are checking the WooCommerce nonce. If that is verified, then we are sanitizing the field and saving the Service ID in the meta key *_service\product_id*.

Checking the Stock Level

The *Service ID* field is now fully functional on the admin part. We still need to check the stock level with that ID. First, let's hook in the filter that returns the stock amount inside WooCommerce.

```
/**
 * Adding Hooks
 * @return void
 */
public function hooks() {

        // Creating the product field
        add_action( 'woocommerce_product_options_stock_fields', 'woodev_stock_product_s\
ervice_field' );
```

```
        // Saving the product field
        add_action( 'woocommerce_process_product_meta', 'woodev_stock_product_service_f\
ield_save' );

        // Filtering the Stock Quantity
        add_filter( 'woocommerce_get_stock_quantity', 'woodev_stock_get_stock_quantity'\
, 10, 2 );

}
```

The filter *woocommerce_get_stock_quantity* is called in several parts of WooCommerce. It will be checked when the product is updated or published, when the product is added to the cart, on the checkout etc.

Filtering the Stock Level

Let's now define the function *woodev_stock_get_stock_quantity* inside the file *functions-wc.php*:

```
/**
 * Item has been added to cart, check the stock level
 * @return void
 */
function woodev_stock_get_stock_quantity( $amount, $product ) {

        $stock_amount = woodev_stock_get_level_for_product( $product->id );

        if( is_wp_error( $stock_amount ) ) {
                // API returned an Error, what do to? You can return the current amount or ret\
urn 0 to be sure
                // I will return 0 because I don't want to sell something that I can't ship
                return 0;
        }

        if( false === $stock_amount ) {
                // The product is not a service product
                return $amount;
        }

        // Perform Other checks if needed

        $amount = $stock_amount;
```

```
        return $amount;
}
```

We are getting the stock level by the function *woodev_stock_get_level_for_product*. This function will use the Service API to get the stock level. It will return:

- WP_Error, if the request failed for a reason,
- false, if the product does not have the Service ID (not connected to Service)
- integer, if the request was made without any issues and returns the stock level

If he product does not have the *Service ID*, we will return the current stock amount. If there is a WP_Error, we will return 0 as stock level, because we want to be sure we have the product. With an error, we can't be sure of it.

Now we just need to define the function *woodev_stock_get_level_for_product* and others to connect with our Service/API.

Getting the Stock Level from Service

To handle the stock level from a service, we need to have a few functions for performing the API call, get he product ID from the Service etc.

```
/**
 * Get the Stock level for the product ID
 * @param  integer $product_id
 * @return mixed                    It will return the stock amount, WP Error on reques\
t error or false if the product is not connected to service
 */
function woodev_stock_get_level_for_product( $product_id ) {

        $product_api_id = woodev_stock_get_product_api_id( $product_id );

        // This product is not handled by our service
        if( ! $product_api_id ) {
                return false;
        }

        $args = array(
                'body' => array(
                        'product_id' => $product_api_id
```

```
        ));

        $stock_response = woodev_stock_perform_request( 'stock', $args );
        $stock_level = 0;

        if( is_wp_error( $stock_response ) ) {
                return $stock_response;
        }
        // If the response will be retrieved
        if( isset( $stock_response['response'] ) && $stock_response['response']['code']\
 != 200 ) {
                return new WP_Error( 'api_error', $stock_response['response']['message'] );
        }

        $stock_response_body = wp_remote_retrieve_body( $stock_response );

        // I am assuming this will return only the Stock Number.
        if( is_integer( $stock_response_body ) ) {
                $stock_level = $stock_response_body;
        }

        /**
         * If the API returns an XML response, JSON or something else,
         * you will need to parse the response and get the stock level
         */
        return apply_filters( 'woodev_stock_get_level_for_product', $stock_level, $prod\
uct_id );
}
```

First, we get the Service Product ID. If there is no ID, we will return *false*. That is why we are returning the current stock amount in the function *woodev_stock_get_stock_quantity* before.

If there is a Service Product ID, the function *woodev_stock_perform_request* will perform the API request. We will return the *WP_Error* if the response if an error or if the response code is not **200**.

The function *wp_remote_retrieve_body* will return the body of the response and if it's an *integer* we will assing the value to the variable **$stock_level**.

The returned value from our function can be also filtered out to include other checks. You can also extend this function by performing other checks.

The product will become **out of stock** if the returned value is 0. You can try it by defining this function to return 0 and see what happens.

Let's now define our other two functions that are called here.

Getting the Service Product ID

We have already mentioned the function *woodev_stock_get_product_api_id* inside the code above.

```
/**
 * Returns the Product ID stored in API/Service
 * @param  integer $product_id
 * @return mixed
 */
function woodev_stock_get_product_api_id( $product_id ) {
        return get_post_meta( $product_id, '_service_product_id', true );
}
```

Performing the API request

The function *woodev_stock_perform_request* will be able to handle several requests. I will leave that to you to define. In the code example you will see how you can define it for getting the Stock Level.

```
/**
 * Performing the Remote Get
 * @return WP_HTTP
 */
function woodev_stock_perform_request( $service, $args = array() ) {

        $url = '';

        // Build the header if needed
        $args['header'] = array(
                'SERVICE_HEADER_API_KEY' => 'SERVICE_HEADER_API_KEY_VALUE'
        );

        switch ( $service ) {
                case 'stock':
                        $url = 'https://api.yoursite.com/path/for/stock';
                        break;

                default:
                        # code...
                        break;
        }

        if( ! $url ) {
```

```
                return new WP_Error( 'api_error', __( 'There is no URL provided to perform a r\
equest' ) );
        }

        return wp_remote_get( $url, $args );
}
```

For some APIs, you will also need to handle POST requests, but as we are only *getting* the stock level here I have implemented only the function *wp_remote_get*.

Courses

This chapter is a link to all the courses I have made for this eBook. The courses are hosted on a separate platform where Iâ€™ll create many other courses.

Since the videos had to be encoded, edited and some other work has to be done, I am offering this course at $35.

Because you have bought this eBook, I am offering you a coupon that will lower the price of the courses by $10.

For now, these is the list of courses for this eBook with coupon codes applied to the link:

- Custom WooCommerce Shipping Method[10]

[10] http://practicalwp.teachable.com/p/create-a-custom-shipping-method-in-woocommerce/?product_id=261080&coupon_code=REDUCE10

Changelog

28.05.2018

- Refactored the code and rewritten the text in Shipping Method chapters. Removed old code. Product attributes are now retrieved with methods rather than direct access.
- Made a decision to write only up to date code with WC 3.x.

11.04.2017

- Refactored code in “WooCommerce 2.6 new Shipping features” > “On Activation”
- Refactored code in “Printing Order Details from the Admin area” > “Adding the Print Button”