Parallels® Panel

Writing Customer and Business Manager Plugins

Developer's Guide & Reference

Revision 1.0

Contents

Preface

In this section:

About This Guide

This document describes how to introduce new payment processing systems, domain name registrars, and SSL certificate providers to Business Manager so that administrators and customers are able to use them.

Who Should Read This Guide

This guide is addressed to individual developers and software vendors who plan to write pluggable modules that add the support for a certain payment system, domain name registrar, or SSL certificate provider to Business Manager. We assume that the developers have PHP programming skills as the plugins should be implemented in PHP.

Typographical Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it.

The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information	Example
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.	Go to the **System** tab.
	Titles of chapters, sections, and subsections.	Read the **Basic Administration** chapter.

Formatting convention	Type of Information	Example
Italics	Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	The system supports the so called *wildcard character* search.
`Monospace`	The names of commands, files, and directories.	The license file is located in the `http://docs/common/licenses` directory.
`Preformatted`	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	**`# ls -al /files`** `total 14470`
`Preformatted Bold`	What you type, contrasted with on-screen computer output.	**`# cd /root/rpms/php`**
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

Feedback

If you have found an error in this guide, or if you have suggestions or ideas on how to improve this guide, please send your feedback using the online form at http://www.parallels.com/en/support/usersdoc/. Please include in your report the guide's title, chapter and section titles, and the fragment of text in which you have found an error.

CHAPTER 1

Introduction

Business Manager design makes it possible to extend the system with new abilities, namely:

- Process payments from custom payment processing system
- Sell domain names using a reseller account with a registrar that is not yet supported by the system
- Sell SSL certificates using a reseller account with an SSL provider that Business Manager does not currently support.

Each of the abilities is handled by a separate part of Business Manager API. You can use the API to create pluggable modules and instruct Business Manager to use them. We explained the details on how to practically do it in the three next chapters of this document. All the technical details and uncovered places in these instructions are discussed in the **Reference** chapter (on page 57).

Before you proceed to plugins creation, we recommend that you read **Debugging** (on page 81) to find out how to efficiently and promptly diagnose errors in code.

If you wish to translate your plugin to other languages, see how to do it in **Localization** (on page 79).

CHAPTER 2

Writing Payment Gateways

Charging customers for different services is one of the most important functions of Customer & Business Manager. Typically, if a customer pays with a credit card or a bank transfer, Business Manager passes the operation to a payment processing system and receives the result. The examples of such systems are Authorize.Net and PayPal.

<bm_names> connects to payment processing systems through middleware – *payment gateways* that are modules written in PHP or Perl. The interaction between these three entities is as follows:

1. Business Manager send payment details to the corresponding gateway.
2. The gateway transforms the details into the format accepted by the related processing system and sends the operation request for processing.
3. When the processing system completes the operation, it sends the result back to the gateway that, in turn, returns it to Business Manager.

Business Manager comes with payment gateways for major payment processing systems. If you need to process payments with the system that is not yet supported by Business Manager, implement a corresponding gateway for it.

Gateway Types

Business Manager supports the following types of payment gateways:

- *On-site* payment gateways allow customers to pay by credit cards without forwarding them to third-party sites. All credit card data is stored in Business Manager.
- *Off-site* payment gateways send your customers to sites of payment processing systems, where they enter credit card data and preform payments.
- *Bank transfer* gateways give you the option to accept payments by transferring money directly from your customers' bank accounts to your bank account.

Next in this chapter you will find the instructions on writing on-site payment gateways for Business Manager.

In addition to this guide, we provide a PHP file containing the example of a gateway class with declared functions and comments that help to implement the functions. You can find it the `/sdk/samples/gateway` directory.

Localization

When you create Business Manager user interface elements such as plugin configuration parameters names or error messages, consider using locale keys instead of actual values. This will allow to easily translate these names and messages to other languages. Learn how to add new locale keys to Business Manager in the section **Localization** (on page 79).

Debugging

To make the development process easier and more efficient, you may use standard PHP or Business Manager functions that output error information and show you the calls of your gateway's methods. Find out more about using them in the section **Debugging** (on page 81).

In this chapter:

1. Creating a Gateway Class

To make your payment gateway visible to Business Manager you should meet the following requirements when you create the module:

- A plugin should be class inherited from the `Gateway` class.
- The filename part that precedes `.php` should be equal to the class name.
- The file should be located in the `opt/plesk-billing/lib/lib-mbapi/include/modules/gateway` directory. You can use gateways from this directory as examples when writing your own gateway.

The class definition should look like the following:

```
class MyGateway extends Gateway
```

2. Defining Gateway Descriptor

Each payment gatwway should have its *descriptor* - a structure containing the following information about the gateway:

- *type*. Should be `gateway` for all payment gateways.
- *status*. May have one of two values:
  - `STATUS_STABLE` for appropriately tested gateways.
  - `STATUS_TESTING` for gateways that are currently in testing state and the using of the gateway may cause some risks. Business Manager does not show gateways with this status by default in the user inteface; to see them, an administrator should click the corresponding link.
- *version.* Is shown as the gateway version in Business Manager user interface.
- *gatewayType*. May have one of the following values:
  - `TYPE_PRIMARY` for on-site gateways.
  - `TYPE_THIRD_PARTY` for off-site gateways.
- *displayName*. Is shown as the gateway name in Business Manager user interface.
- *author.* Is shown as the gateway author name in Business Manager user interface.
- *capabilities* (array). A list of supported operations such as charging or refunding. Find the full list of the gateway capabilities in the section **Gateway Capabilities List** (on page 59).
- *countryCodes*. A list of two-letter codes of countries sellers from which can use the gateway.

Define the descriptor in the global variable as shown in the sample below:

```
$className = 'MyGateway';
$GLOBALS['moduleInfo'][$className] = array(
      'type' => 'gateway',
      'status' => PaymentGateway::STATUS_STABLE,
      'version' => '1.0.0',
      'gatewaytype' => PaymentGateway:: TYPE_PRIMARY,
      'displayName' => $className,
      'author' => 'Parallels',
      'capabilities' => array(
            GATEWAY_CHARGE,
            GATEWAY_AUTHORIZE,
            GATEWAY_PRIOR_AUTH_CHARGE,
            GATEWAY_REFUND,
            GATEWAY_VOID,
            GATEWAY_IPV6_COMPLIANCE,
      ),
      'countryCodes' => array('US', 'CA'),
);
```

Then implement the `getModuleInfo` function to return the descriptor:

```
public function getModuleInfo()
{
      return $GLOBALS['moduleInfo']['MyGateway'];
}
```

3. Adding Gateway Settings

When Business Manager administrators decide to configure a certain payment gateway, they go to the **Business Setup** > **All Settings** > **Payment Gateways** page and select one of the available modules. Then Business Manager opens the gateway settings page, where administrators configure the plugin, for example, provide a merchant account ID or enable an option of accepting Visa cards.

You define what parameters Business Manager administrator will see on this page in the `addDefaultConfigParams` function that returns an array of the parameters. Find the instruction on how to define these parameters in the section **Gateway Settings Format** (on page 58).

Below is a sample of the `addDefaultConfigParams()` function.

```
public function addDefaultConfigParams()
{

      $defaultConfigParams = array(
            array(NULL, NULL, "login", "", "",
"TRANS_MYGATEWAY_LOGIN", TRANS_MYGATEWAY_LOGIN_DESC", "t", 1, 1, 0,
NULL),
            array(NULL, NULL, "password", "", "",
"TRANS_MYGATEWAY_PASSWORD",   "TRANS_MYGATEWAY_PASSWORD_DESC", "p", 1,
2, 0, NULL),
            array(NULL, NULL, "language", "en", "",
"TRANS_DISPLAY_LANGUAGE",     "TRANS_DISPLAY_LANGUAGE_DESC", "r", 1,
3, 0, array(
                        array(NULL, NULL, "en", "TRANS_ENGLISH", 0),
                        array(NULL, NULL, "sp", "TRANS_SPANISH", 1),
            )),
            return $defaultConfigParams;
}
```

4. Implementing Operations Processing

To perform payment operations processing, implement the `execute()` function.

This function does the following:

1. Determines the operation type by parsing the `$this->params['subCommand'][0]` variable.
2. Sends a request to the payment system.
3. Processes the response or outputs error information if needed.
4. Forms an array of operation results. This array must contain the following data:
   - `status`.
   - `statusName`. A name used in Business Manager to identify the status. You can get it using the `getAPIStatusName()` function.
   - `statusMessage`. The status description used in Business Manager. You can get it using the `getStatusName()` function.
   - `authorizationCode`. An authorization code returned by the payment system in case of charging and authorization operations.
   - `transactionID`. A transaction identifier returned by the payment system in case of charging and authorization operations.
   - `AVSCode`. Address verification system code returned by the system in case of charging and authorization operations.
   - `message`. A message about operation results.
5. Sets the `$this->hasExecuted` flag to 1 and returns the results in Business Manager supported XML format.

In case the gateway supports all the capabilities, this function looks like the following:

```
public function execute()
{
      switch ($this->params['subCommand'][0]) {
      case 'charge':
            <...>
            break;
      case 'auth':
            <...>
            break;
      case 'finalize':
            <...>
            break;
      case 'refund':
            <...>
            break;
      case 'void':
            <...>
            break;
      default:
            $this->status = GATEWAY_ERROR;
            $this->message = 'Invalid Gateway Action Requested';
      }

      $result = array(
            'status' => $this->status,
            'statusName' => getAPIStatusName('transactions', $this-
>status),
            'statusMessage' => getStatusName('transactions', $this-
>status),
            'authorizationCode' => $this->authorizationCode,
            'transactionID' => $this->transactionID,
            'AVSCode' => $this->AVSCode,
            'message' => $this->message,
      );

      $this->hasExecuted = 1;
      return $this->createOutputXML($result);
}
```

In each case, the function performs the following actions:

1. *Sends a request to the registrar.*

   In our samples we use the placeholder ***callToPaymentSystem(prepareRequest())*** to indicate this step. When you send the request, you are able to refer to the following input parameters:

   - Operation information such as bank account number or card type in the `$this->params` array. Find the description of the array elements in the section **Getting Payment Information** (on page 60).
   - Gateway configuration settings that you added on the step **3. Adding Gateway Settings** (on page 10) in the $this->configuration array.

2. *Gets a response and saves the operation status*

   A gateway saves the status in the `$this ->status` variable. The possible statues are the following:

   - `GATEWAY_SUCCESS` - the payment system has successfully performed the operation.
   - `GATEWAY_DECLINED` - the payment system has declined the request.
   - `GATEWAY_FAILURE` - the operation has failed.

3. *Saves a message containing information about the operation result.*

   A gateway saves the message in the `$this->message` variable. You can use it to transfer the information about the operation to Business Manager administrator or customers.

4. *In case of successfully completed charge and authorization actions saves the following data returned by the system to the corresponding variables*:

   - authorization code (`$this->authorizationCode`)
   - transaction identifier (`$this->transactionID`)
   - AVS code (`$this->AVSCode`)

The subsections of this section provide the detailed information about implementing capabilities listed in the section **Gateway Capabilities List** (on page 59).

In this section:

Authorization

This command blocks a certain amount of money on the payer's card to get ready to the money transfer.

To transfer the blocked amount, perform the **Capture** (on page 14) operation.

You can cancel the authorization by performing the **Voiding** (on page 16) operation. If nobody capture or voids the payment within a month, the payment system voids the operation automatically.

➢ ***To implement the authorization capability:***

1. Add the `GATEWAY_AUTHORIZE` to the gateway `'capabilities'` array. See the section **2. Defining Gateway Descriptor** (on page 9) for details.
2. Add the case of authorization operation to the `execute()` function. Learn how to refer to the operation data in the section **Getting Payment Information** (on page 60). This case may look like the following:

```
case 'auth':
      $registrarResponseArray = callToPaymentSystem(prepareRequest($this->params));
      switch($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->status = GATEWAY_SUCCESS;
                  $this->message = "Operation successfully completed";
                  break;
            case "declined":
                  $this->status = GATEWAY_DECLINED;
                  $this->message = "Operation declined";
                  break;
            case "failure":
                  $this->status = GATEWAY_FAILURE;
                  $this->message = "Operation failed";
                  break;
      }
      if (GATEWAY_SUCCESS == $this->status) {
            $this->authorizationCode =
$registrarResponseArray["authCode"];
            $this->transactionID = $registrarResponseArray["transID"];
            $this->AVSCode = $registrarResponseArray["avsCode"];
      }
      break;
```

Capturing

This command transfers the blocked amount of money from the payer's card to seller's merchant account. You can perform this operation to a payment after its **Authorization** (on page 14).

To cancel this operation and return the money to the payer, use the **Refund** (on page 16) command.

➢ ***To implement the capture capability:***

1. Add the `GATEWAY_FINALIZE` to the gateway `'capabilities'` array. See the section **2. Defining Gateway Descriptor** (on page 9) for details.
2. Add the case of authorization operation to the `execute()` function. Learn how to refer to the operation data in the section **Getting Payment Information** (on page 60). This case may look like the following:

```
case 'finalize':
      $registrarResponseArray = callToPaymentSystem(prepareRequest($this->params));
      switch($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->status = GATEWAY_SUCCESS;
                  $this->message = "Operation successfully completed";
                  break;
            case "declined":
                  $this->status = GATEWAY_DECLINED;
                  $this->message = "Operation declined";
                  break;
            case "failure":
                  $this->status = GATEWAY_FAILURE;
                  $this->message = "Operation failed";
                  break;

      }
      break;
```

Charging

Charging includes the **Authorization** (see page 14) and **Capture** (see page 14) actions: it block a certain amount of money on a payer's card and then transfers in to a seller's merchant account.

To cancel a charging operation and return the money, use the **Refund** (on page 16) command.

➢ ***To implement the charging capability:***

1. Add the `GATEWAY_CHARGE` to the gateway `'capabilities'` array. See the section **2. Defining Gateway Descriptor** (on page 9) for details.
2. Add the case of charging operation to the `execute()` function. Learn how to refer to the operation data in the section **Getting Payment Information** (on page 60). This case may look like the following:

```
case 'charge':
      $registrarResponseArray = callToPaymentSystem(prepareRequest($this->params));
      switch($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->status = GATEWAY_SUCCESS;
                  $this->message = "Operation successfully completed";
                  break;
            case "declined":
                  $this->status = GATEWAY_DECLINED;
                  $this->message = "Operation declined";
                  break;
            case "failure":
                  $this->status = GATEWAY_FAILURE;
                  $this->message = "Operation failed";
                  break;
      }
      if (GATEWAY_SUCCESS == $this->status) {
            $this->authorizationCode =
$registrarResponseArray["authCode"];
            $this->transactionID = $registrarResponseArray["transID"];
            $this->AVSCode = $registrarResponseArray["avsCode"];
      }
      break;
```

Refund

The refund command returns the money transferred within a certain payment back to a payer's card. You can perform the refund for a payment after its **Capture** (on page 14) or **Charging** (on page 15).

➢ ***To implement the refund capability:***

1. Add the `GATEWAY_REFUND` to the gateway `'capabilities'` array. See the section **2. Defining Gateway Descriptor** (on page 9) for details.
2. Add the case of authorization operation to the `execute()` function. Learn how to refer to the operation data in the section **Getting Payment Information** (on page 60). This case may look like the following:

```
case 'refund':
      $registrarResponseArray = callToPaymentSystem(prepareRequest($this->params));
      switch($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->status = GATEWAY_SUCCESS;
                  $this->message = "Operation successfully completed";
                  break;
            case "declined":
                  $this->status = GATEWAY_DECLINED;
                  $this->message = "Operation declined";
                  break;
            case "failure":
                  $this->status = GATEWAY_FAILURE;
                  $this->message = "Operation failed";
                  break;

      }
      break;
```

Voiding

This command cancels the **Authorization** (on page 14) operation and makes the blocked amount of money available to a payer.

➢ ***To implement the voiding capability:***

1. Add the `GATEWAY_VOID` to the gateway `'capabilities'` array. See the section **2. Defining Gateway Descriptor** (on page 9) for details.
2. Add the case of authorization operation to the `execute()` function. Learn how to refer to the operation data in the section **Getting Payment Information** (on page 60). This case may look like the following:

```
case 'void':
      $registrarResponseArray = callToPaymentSystem(prepareRequest($this->params));
      switch($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->status = GATEWAY_SUCCESS;
                  $this->message = "Operation successfully completed";
                  break;
            case "declined":
                  $this->status = GATEWAY_DECLINED;
                  $this->message = "Operation declined";
                  break;
            case "failure":
                  $this->status = GATEWAY_FAILURE;
                  $this->message = "Operation failed";
                  break;
      }
      break;
```

C H A P T E R 3

Writing Domain Name Registrar Plugins

Selling domain names, as well as providing hosting services is one of the most frequently used function of Customer & Business Manager. When somebody purchases a domain name from a hosting provider that uses Business Manager, the following sequence of actions happens in the background:

1. Business Manager generates and sends a request to a certain domain name registrar.
2. The registrar creates a domain name and sends back the operation status.
3. Business Manager returns the status to the requester.

The logic of sending requests to registrars and processing responses from them in stored in *domain name registrar plugins* - modules connected to Business Manager; one module for each supported registrar.

Using the plugins, you can add the support for a custom domain name registrar to Business Manager. Next in this document we will explain how to do this.

Creating Domain Name Registrar Plugins

Business Manager domain name registrar plugins are PHP classes that perform interaction between Business Manager and the registrars. The main steps of implementing your own domain name registrar plugin are the following:

1. Create the plugin file and place it in a certain directory.
2. Create a structure containing common information about the plugin - the *plugin descriptor*. See the section **Defining Plugin Descriptor** (on page 20) for instructions on creating the descriptor and the descriptor code samples.
3. Describe fields that administrators fill to configure the interaction between Business Manager and a registrar (They do it on the **Business Setup** > **All Settings** > **Registrar Modules** > ***<registrar>*** page). Learn how to do it in the section **Defining Plugin Configuration Parameters** (on page 21).
4. Define which actions will be available to Business Manager administrators and customers (for example, domain name registration or transfer) and implement the functions that perform these actions. Find the list of these actions and instructions on implementing them in the section **Setting Registrar Capabilities** (on page 22).

Next in this chapter, we will discuss each step in details.

In addition to this guide, we provide a PHP file containing the example of a domain registrar plugin with declared functions and comments that help to implement the functions. You can find it the `/sdk/samples/domain` directory.

Localization

When you create Business Manager user interface elements such as plugin configuration parameters names or error messages, consider using locale keys instead of actual values. This will allow to easily translate these names and messages to other languages. Learn how to add new locale keys to Business Manager in the section **Localization**.

Debugging

To make the development process easier and more efficient, you may use standard PHP or Business Manager functions that output error information and show you the calls of your plugin's methods. Find out more about using them in the section **Debugging**.

In this chapter:

1. Creating a Plugin Class

To make your plugin visible to Business Manager you should meet the following requirements when you create the module:

- A plugin should be class inherited from the *Registrar* class.
- The filename part that precedes `.php` should be equal to the class name.
- The file should be located in the `opt/plesk-billing/lib/lib-mbapi/include/modules/registrar` directory. You can use registrars from this directory as examples when writing your own registrar.

The class definition should look like the following:

```
class MyRegistrar extends Registrar
```

2. Defining Plugin Descriptior

Each registrar plugin should have its *descriptor* - a structure containing the following information about the plugin:

- *type.* Should be `registrar` for all domain name registrar plugins.
- *status*. May have one of two values:
  - `STATUS_STABLE` for appropriately tested plugins.
  - `STATUS_TESTING` for plugins that are currently in testing state and the using of the plugin may cause some risks. Business Manager does not show plugins with this status by default in the user inteface; to see them, an administrator should click the corresponding link.
- *displayName*. Is shown as the registrar name in Business Manager user interface.

Business Manager will show this data on the domain name registrar choosing page **Business Setup** > **All Seetings** > **Registrar Modules**.

Define the descriptor in the global variable as shown in the sample below:

```
$className = 'MyRegistrar';
$GLOBALS['moduleInfo'][$className] = array(
    'type' => 'registrar',
    'status' => Registrar::STATUS_STABLE,
    'displayName' => 'My registrar',
);
```

Then implement the `getModuleInfo` function to return the descriptor:

```
public function getModuleInfo()
{
      return $GLOBALS['moduleInfo']['MyRegistrar'];
}
```

3. Adding Plugin Settings

When Business Manager administrators decide to configure a certain domain registrar plugin, they go to the **Business Setup** > **All Settings** > **Registrar Modules** page and select one of the available modules. Then Business Manager opens the plugin settings page, where administrators configure the plugin, for example, provide access credentials or auto renewal parameters.

You define what parameters Business Manager administrator will see on this page in the `addDefaultConfigParams` function that returns an array of the parameters. Find the instruction on how to define these parameters in the section **Plugin Settings Format** (on page 63).

Below is a sample of the `addDefaultConfigParams()` function.

```
public function addDefaultConfigParams()
{

      $defaultConfigParams = array(
            array('moduleUsername', 't', NULL, 'USERNAME', NULL, NULL,
1),
            array('modulePassword', 'p', NULL, 'PASSWORD', NULL, NULL,
1),
            array('moduleTestMode', 'b', '1', 'ENABLE_TEST_MODE',
NULL, NULL, 0),
            array('moduleAllowAutoRenew', 'r', '0', 'ALLOW_AUTORENEW',
'ALLOW_AUTORENEW_DESC',
                  array(
                    array(
                        NULL,
                        NULL,
                        '0',
                        'TRANS_OBEY_TLD',
                        1
                    ),
                    array(
                        NULL,
                        NULL,
                        '1',
                        'TRANS_REGISTRAR_AUTORENEW_OPTION',
                        2
                    ),
                    array(
                        NULL,
                        NULL,
                        '2',
                        'TRANS_MODERNBILL_RENEW_OPTION',
                        3
                    ),
                ), 0
            ),


      );
      return $defaultConfigParams;
}
```

Important: The samples in this document use the `$this->UID`, and `$this->PW` variables to refer to the username and password fields. It is possible if the corresponding fields of the configuration parameters array have the names `moduleUsername` and `modulePassword`,

4. Setting Plugin Capabilities

Each domain name registrar plugin should support a number of *capabilities* - features like domain registration, editing registrant contact information, and so on. Even if a corresponding registrar supports a certain capability, you are still able to restrict its use by the your plugin. For each capability that your plugin supports, you should implement a function that performs the corresponding action, say, sends a request to register a domain name. You will find advices on implementing these functions in the subsections of this section.

A plugin defines its capabilities in the static associative array `$moduleCapabilities`, so you are free to refer to the elements using the text constant values, for example, `$this->capabilities[REGISTRAR_REGISTERDOMAIN] = 1`.

The registrar plugin capabilities definition may look like the following:

```
private static $moduleCapabilities = array(
        REGISTRAR_REGISTERDOMAIN => 1,
    );
```

You can find the full list of supported capabilities in the section **Domain Name Registrar Capabilities List** (on page 63).

Implementing a Capability

Most of the functions that make the capabilities work contain the following main steps:

1. *Send a request to a registrar API.*

   You are free to use SOAP, cURL, or other data exchanging tools. In our samples we use the placeholder ***callToRegistrar(prepareRequest())*** to indicate this step; you should write your own function considering the API provided by a registrar. When you create the request, you are able to refer to the input parameters such as domain name or registrant contact information fields using the `$this->input` array. Find the description of the array elements in the section Getting Domain Information (on page 66).

2. *Process the response.*

   We suppose that the registrar returns an array of the operation details. If your registrar returns a plain text response, say, an XML, the function should form an array of data that the XML contains. This step usually includes writing the received parameters to the `$this->attibutes` or `$this->arrayData` arrays and saving error information if needed.

3. *Transfer the operation result to the Business Manager acceptable XML format.*

   To do this, run the function `$this->getRegistrarReturnXML()`.

See the subsections of this section to learn what particular steps you should take to implement certain domain name registrar capabilities.

In this section:

Domain Registration

This capability allows registering new domain names with a registrar.

When a customer purchases a domain name in your online store, Business Manager creates an event (task) *ProcessRegistrar of subscription NN*, where *NN* is the customer's subscription ID. After the invoice for the domain registration is paid, Business Manager runs this event. In turn, the event triggers the `registerDomain()` method.

➢ *To implement the domain name registration capability:*

1. Set value 1 for the `REGISTRAR_REGISTERDOMAIN` capability using the following line:

```
REGISTRAR_REGISTERDOMAIN => 1,
```

2. Implement the `registerDomain()` function that requests the registrar's API to register a new domain name. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   During the registrar response processing, the module must return one of the statuses:

   - `self::ACTION_STATUS_COMPLETED`. Change the status of the corresponding Business Manager event to *Completed*, make the subscription *Active.*
   - `self::ACTION_STATUS_WAITING`. Set the event status to *In progress*; it will be executed again in 5 minutes. Change the subscription status to *Work in progress*.
   - `self::ACTION_STATUS_ERROR`. Set the event and subscription statuses to *Failed.*

The result may look like the following:

```
public function registerDomain()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input , $this->UID, $this-
>PW)) ;
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus =
self::ACTION_STATUS_COMPLETED;
                  break;
            case "pending":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus = self::ACTION_STATUS_WAITING;
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
```

```
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  $this->commandStatus = self::ACTION_STATUS_ERROR;
                  break;
      }

      return $this->getRegistrarReturnXML();
}
```

Domain Transfer

This capability allows transferring the support of an existing domain to the registrar. To let Business Manager administrators protect customers domains from stealing by transfers, you can allow using *domain transfer locks*. For instructions on how to implement the corresponding ability, refer to the section **Managing Transfer Lock for Domain** (on page 28).

When a customer orders a domain name transfer in your online store, creates an event (task) *ProcessRegistrar of subscription NN*, where *NN* is the customer's subscription ID. After the invoice for the domain registration is paid, Business Manager runs this event. In turn, the event triggers the `transferDomain()` method.

➢ *To implement the domain name transfer capability:*

1. Set value *1* for the `REGISTRAR_TRANSFERDOMAIN` capability using the following line:

```
REGISTRAR_TRANSFERDOMAIN => 1,
```

2. Implement the `transferDomain()` function that that requests the registrar's API to transfer a domain name. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   During the registrar response processing, the module must return one of the statuses:

   - `self::ACTION_STATUS_COMPLETED`. Change the status of the corresponding Business Manager event to *Completed*, make the subscription *Active.*
   - `self::ACTION_STATUS_WAITING`. Set the event status to *In progress*; it will be executed again in 5 minutes. Change the subscription status to *Work in progress*.
   - `self::ACTION_STATUS_ERROR`. Set the event and subscription statuses to *Failed.*

The result may look like the following:

```
public function transferDomain()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input , $this->UID, $this->PW));
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus = 
self::ACTION_STATUS_COMPLETED;
                  break;
            case "pending":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus = self::ACTION_STATUS_WAITING;
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  $this->commandStatus = self::ACTION_STATUS_ERROR;
                  break;
      }
      return $this->getRegistrarReturnXML();
}
```

In this section:

Managing Transfer Lock

The managing transfer lock capability allows Business Manager administrator and customers (if the administrator allows this) to protect their domain from stealing by prohibiting transfer of a certain domain.

➢ *To implement the retrieving domain transfer lock status capability:*

1. Set value *1* for the `REGISTRAR_GETDOMAINREGISTRARLOCKOPTIONS` capability using the following line:

```
REGISTRAR_GETDOMAINREGISTRARLOCKOPTIONS => 1
```

2. Implement the `getDomainRegistrarLockOptions()` function that retrieves the domain transfer lock status. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   If the registrar has returned the status, assign the appropriate values to the `$this->arrayData['domainIsLockable']` and `$this->arrayData['domainRegistrarLock']` attributes:

   - *1* if the domain is lockable/locked.
   - *0* if the domain is not lockable/locked.

   Otherwise, output the error information.

   The result may look like the following:

```
public function getDomainRegistrarLockOptions()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  <...>
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  break;
      }
      return $this->getRegistrarReturnXML();
}
```

➢ ***To implement the updating domain transfer lock status capability:***

1. Set value *1* for the `REGISTRAR_SETDOMAINREGISTRARLOCKOPTIONS` capability using the following line:

```
REGISTRAR_SETDOMAINREGISTRARLOCKOPTIONS => 1,
```

2. Implement the `setDomainRegistrarLockOptions()` function that updates the domain transfer lock status. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   After you get the response from the registrar, output the error information, if needed.

```
public function setDomainRegistrarLockOptions()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input));
      foreach($registrarResponseArray["errors"] as $error) {
            $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
      }
      return $this->getRegistrarReturnXML();
}
```

3. Add the parameter `moduleAllowChangeRegistrarLock` to the default configuration parameters array in the `addDefaultConfigParameters()` function either to allow or to deny Business Manager customers to use this feature:

```
array('moduleAllowChangeRegistrarLock', 'b', '0',
'ALLOW_CHANGE_REGISTRAR_LOCK', 'ALLOW_CHANGE_REGISTRAR_LOCK_DESC',
NULL, 0),
```

Domain Renewal

This capability allows renewal of domain names at the registrar.

Business Manager supports two types of automatic domain renewal: automatic renewal at the registrar side and automatic sending of renewal requests by the Business Manager. For detailed information on how to retrieve and update automatic domain renewal type, refer to the section **Managing Renewal Settings** (on page 31).

If you configure Business Manager to send renewal requests to the registrar, then the system will create an event (task) *ProcessRegistrar of subscription NN* (where *NN* is the customer's subscription ID) at the end of the subscription billing cycle and issue the invoice for the next cycle. After the invoice for the domain renewal is paid, Business Manager runs this event. In turn, the event triggers the `renewDomain()` method.

Moreover, Business Manager supports manual domain renewal by the administrator. If you want to allow administrators to renew domains manually, find the instructions in the section **Configuring Manual Renewal** (on page 32).

➢ *To implement the domain name renewal capability:*

1. Set value *1* for the `REGISTRAR_RENEWDOMAIN` capability using the following line:

```
REGISTRAR_RENEWDOMAIN => 1,
```

2. Implement the `renewDomain()` function that that requests the registrar's API to renew a domain name. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   During the registrar response processing, the module must return one of the statuses:

   - `self::ACTION_STATUS_COMPLETED`. Change the status of the corresponding Business Manager event to *Completed*, make the subscription *Active.*
   - `self::ACTION_STATUS_WAITING`. Set the event status to *In progress*; it will be executed again in 5 minutes. Change the subscription status to *Work in progress*.
   - `self::ACTION_STATUS_ERROR`. Set the event and subscription statuses to *Failed.*

The result may look like the following:

```
public function renewDomain()
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input , $this->UID, $this->PW));
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus =
self::ACTION_STATUS_COMPLETED;
                  break;
            case "pending":
                  $this->attributes = $registrarResponseArray;
                  $this->commandStatus = self::ACTION_STATUS_WAITING;
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  $this->commandStatus = self::ACTION_STATUS_ERROR;
                  break;
      }

      return $this->getRegistrarReturnXML();
}
```

In this section:

Managing Renewal Settings

Domain renewal may be performed in the following two ways:

- Business Manager sends renewal request to the registrar.
- The registrar performs the renewal automatically.

If the updating domain renewal settings capability is enabled, Business Manager administrator can change the domain renewal type in the Business Manager user interface. Moreover, the administrator can allow or restrict changing a domain renewal type by the domain owner.

➢ *To implement the retrieving domain renewal type capability:*

1. Set value *1* for the `REGISTRAR_GETDOMAINRENEWTYPEOPTIONS` capability using the following line:

```
REGISTRAR_GETDOMAINRENEWTYPEOPTIONS => 1,
```

2. Implement the `getDomainRenewTypeOptions()` function that retrieves the domain renewal type. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   If the registrar returns the renewal type, assign the appropriate values to the `$this->arrayData['domainRenewType]` attribute:

   - *1* if the automatic renewal is enabled.
   - *0* if the automatic renewal is disabled.

   Otherwise, output the error information.

```
public function getDomainRenewTypeOptions()
    {
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this-
>PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  <...>
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  break;
      }
      return $this->getRegistrarReturnXML();
}
```

➢ ***To implement the updating domain renewal type capability:***

1. Set value *1* for the `REGISTRAR_SETDOMAINRENEWTYPEOPTIONS` capability using the following line:

```
REGISTRAR_SETDOMAINRENEWTYPEOPTIONS => 1,
```

2. Implement the `getDomainRenewTypeOptions()` function that updates the domain renewal type. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   After you get the response from the registrar, output the error information, if needed.

```
public function setDomainRenewTypeOptions()
    {
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));                foreach($registrarResponseArray["errors"] as $error)
{
            $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
      }
      return $this->getRegistrarReturnXML();
}
```

3. Add the parameter `moduleAllowChangeAutorenew` to the default configuration parameters array in the `addDefaultConfigParameters()` function either to allow or to deny Business Manager customers to use this feature:

```
array('moduleAllowChangeAutorenew', 'b', '0',
'ALLOW_CHANGE_AUTORENEW', 'ALLOW_CHANGE_AUTORENEW_DESC', NULL, 0),
```

Configuring Manual Renewal

Business Manager supports manual domain renewal by the administrator. Manual domain renewal requires a set of data that you can get from the registrar using the `getDomainExtendedInfo()` function.

➢ ***To allow manual domain renewal:***

1. Set value *1* for the `REGISTRAR_LIVERENEW` and `REGISTRAR_GETDOMAINEXTENDEDINFO` capabilities using the following lines:

```
REGISTRAR_LIVERENEW => 1,
REGISTRAR_GETDOMAINEXTENDEDINFO => 1,
```

2. Implement the function `renewDomain()` as described in the section **Domain Renewal** (on page 29).

3. Implement the function `getDomainExtendedInfo()` that retrieves the domain renewal information. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

If the registrar returns the password, assign the appropriate values to the following attributes:

- `$this->arrayData['DomainExpDate']` - domain expiration date.
- `$this->arrayData['DomainExpTimestamp']` - domain expiration timestamp.
- `$this->arrayData['DomainRenewmaxYears']` - domain expiration date.
- `$this->arrayData['DomainRenewDefaultYears']` - domain expiration date.
- `$this->arrayData['DomainRenewPrice']` - domain expiration date.

Otherwise, output the error information.

```
public function getDomainExtendedInfo()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  <...>
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  break;
            }
      return $this->getRegistrarReturnXML();
}
```

Managing Contact Information

Managing contact information includes two capabilities:

- Retrieving contact information.
- Updating contact information.

Business Manager administrator can allow or prohibit customers to get or set the contact information for their domains.

The contact information contains the following sets of contacts;

- registrant contact data
- billing contact data
- auxiliary billing contact data
- technical contact data
- administrative contact data

Besides, you can allow getting information about required registrant contact information fields from the registrar. The instructions on how to do it are provided in the section **Getting Registrant Contact Information Fields Descriptor (on page 36).**

➢ *To implement the retrieving contact information capability:*

1. Set value *1* for the `REGISTRAR_GETDOMAINCONTACTDATA` capability using the following line:

```
REGISTRAR_GETDOMAINCONTACTDATA => 1,
```

2. Implement the `getDomainContactData()` function that retrieves the contact information. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   If the registrar returns the contact information, assign the appropriate values to the elements of the `$this->arrayData` array. You can find the structure of these elements in the section Getting Domain Information.

   Otherwise, output the error information.

```
public function getDomainContactData()
    {
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this-
>PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
      case "success":
            <...>
            break;
      default:
            foreach($registrarResponseArray["errors"] as $error) {
                  $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
```

```
            }
            break;
      }
      return $this->getRegistrarReturnXML();
}
```

➢ *To implement the updating contact information capability:*

1. Set value *1* for the `REGISTRAR_SETDOMAINCONTACTDATA` capability using the following line:

```
REGISTRAR_SETDOMAINCONTACTDATA => 1,
```

2. Implement the `setDomainContactData()` function that updates the contact information. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   After you get the response from the registrar, output the error information, if needed.

```
public function setDomainContactData()
    {
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      foreach($registrarResponseArray["errors"] as $error) {
            $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
      }
      return $this->getRegistrarReturnXML();
}
```

3. Add the parameter `moduleAllowChangeContactInfo` to the default configuration parameters array in the `addDefaultConfigParameters()` function either to allow or to deny Business Manager customers to use this feature:

```
array('moduleAllowChangeContactInfo', 'b', '1',
'ALLOW_CHANGE_CONTACT_INFO', 'ALLOW_CHANGE_CONTACT_INFO_DESC', NULL,
0),
```

In this section:

Getting Contact Information Fields Descriptor

To define what contact information a customer should specify on an online store when registering or transferring a domain name, you should implement the `getRegistrantParams()` function.

This function has the following input parameters:

- `$action`. May be *register* or *transfer.*
- `$domain_sld`.
- `$domain_tld`.
- `$contact`. Always `NULL`.
- `$withTLDParams`. Always `false`.

First, you should get the standard registrant contact form fields by calling the `parent::getRegistrantParams($action, $domain_sld, $domain_tld, $contact, $withTLDParams)`.

Then, if you need, you can add more fields described in the following format:

- `id`. The unique name of the field.
- `type`. Find the available values in the the table below.
- `title`. The name to display in Business Manager user interface.
- `required`. - *1* if required, *0* if optional.
- `default`. Default value.

The `type` may be one of the following:

- `b` - checkbox.
- `s` - drop-down list.
- `t` - text field.
- `r` - radio button.
- `c` - checkbox.
- `x` - HTML
- `a` - text area.
- `p` - password.

```
public function getRegistrantParams($action, $domain_sld, $domain_tld,
ClientContact $contact = NULL, $withTLDParams = false)
{
      $registrantParams = parent::getRegistrantParams($action,
$domain_sld, $domain_tld, $contact, $withTLDParams);
      $inputs = array (
            array("id" => "<domain_fieldID>",
                  "type" => "t",
                  "title" => blmsg("<APPROPRIATE LOCALE KEY NAME>"),
                  "required"  => 0,
                  "default" => "", ),
            );
      $registrantParams = array_merge($registrantParams, array("title"
=> "",
                         "description" => "", "inputs" => $inputs,
));,
      ));
      return $registrantParams;
}
```

Finding out the registrant type

To learn if the registrant is an individual person, commercial company, or an individual entrepreneur, see the value of the `$contact->Client->client_type` variable. These are the possible values of this variable:

- *Client::TYPE_INDIVIDUAL_PERSON* or *0* - individual person
- *Client::TYPE_COMMERCIAL_COMPANY* or *1* - commercial company
- *Client::TYPE_INDIVIDUAL_ENTERPRENEUR* or *2* - individual entrepreneur.

Managing Access Password

Managing domain access password includes the following capabilities:

- Retrieving the password.
- Updating the password.

Business Manager administrator can allow or prohibit customers to get or set the access password for their domains.

➢ *To implement the retrieving domain access password capability:*

1. Set value *1* for the `REGISTRAR_GETDOMAINACCESSPASSWORD` capability using the following line:

```
REGISTRAR_GETDOMAINACCESSPASSWORD => 1,
```

2. Implement the `getDomainAccessPassword()` function that retrieves the password. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   If the registrar returns the password, assign the appropriate value to `$this->arrayData['domainAccessPassword']` attribute:

   Otherwise, output the error information.

```
public function getDomainAccessPassword()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  <...>
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error) {
                        $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
                  }
                  break;
      }
      return $this->getRegistrarReturnXML();
}
```

➢ ***To implement the updating domain access password capability:***

1. Set value *1* for the `REGISTRAR_SETDOMAINACCESSPASSWORD` capability using the following line:

```
REGISTRAR_SETDOMAINACCESSPASSWORD => 1,
```

2. Implement the `setDomainAccessPassword()` function that retrieves the password. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   After you get the response from the registrar, output the error information, if needed.

```
public function setDomainAccessPassword()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      foreach($registrarResponseArray["errors"] as $error) {
            $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
      }
      return $this->getRegistrarReturnXML();
}
```

3. Add the parameter `moduleAllowChangeUpdateDomainPassword` to the default configuration parameters array in the `addDefaultConfigParameters()` function either to allow or to deny Business Manager customers to use this feature:

```
array('moduleAllowChangeUpdateDomainPassword', 'b', '0',
'ALLOW_CHANGE_UPDATE_DOMAIN_PASSWORD',
'ALLOW_CHANGE_UPDATE_DOMAIN_PASSWORD_DESC', NULL, 0),
```

Managing DNS Settings

Depending on the domain name registrar, Business Manager administrator may have different level of permissions to manage DNS settings of the domain:

- *Lower*. Allows the administrator to see and to edit the list of the domain DNS servers.
- *Higher*. Allows to see and to edit the list of the domain DNS servers and resource records.

➢ ***To implement the retrieving domain DNS settings capability:***

1. Set value *1* for the `REGISTRAR_GETDOMAINDNS` capability using the following line:

```
REGISTRAR_GETDOMAINDNS => 1,
```

2. Implement the `getDomainDNS()` function that retrieves the domain DNS settings. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information section.

If the registrar returns the domain DNS information, the function should do the following:

- In case of the lower permissions, assign the appropriate values to the `$this->arrayData['useDNS']` (registrar DNS servers usage) and `$this->arrayData['listDNS']` (a list of custom DNS servers) variables.
- If the permissions are higher, you should assign the appropriate values to the variables mentioned above plus the following:
  - `$this->arrayData['dnsOptions']` - available DNS records types.
  - `$this->arrayData['host']` - an array of the records stores as arrays with the following elements: `type`, `name`, `address`.

Otherwise, output the error information.

```
public function getDomainDNS()
    {
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
      case "success":
            <...>
            break;
      default:
            foreach($registrarResponseArray["errors"] as $error) {
                  $this->addError(blmsg("TRANS_REGISTRARERROR"), $error);
            }
            break;
      }
      return $this->getRegistrarReturnXML();
}
```

➢ *To implement the updating domain DNS settings capability:*

1. Set value *1* for the `REGISTRAR_SETDOMAINDNS` capability using the following line:

```
REGISTRAR_SETDOMAINDNS => 1,
```

2. Implement the `setDomainDNS()` function that updates the domain DNS settings. Find the descriptions of common steps you should take in this function in the section **4. Setting Registrar Capabilities** (on page 22).

   When performing a call to the registrar, use the domain attributes and the registrant information fields described in the Getting Domain Information (on page 66) section.

   After you get the response from the registrar, assign the appropriate value to the `$this->arrayData["result"]` variable or output the error information, if needed.

```
public function setDomainDNS()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      $this->arrayData = array();
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->arrayData["result"] =
"TRANS_REGISTRAR_MANAGE_DNS_UPDATED";
                  break;
            default:
                  foreach($registrarResponseArray["errors"] as $error)
{
                        $this->addError(blmsg("TRANS_REGISTRARERROR"),
$error);
                  }
                  break;
            }
      return $this->getRegistrarReturnXML();
}
```

3. Add the parameter `moduleAllowChangeDNS` to the default configuration parameters array in the `addDefaultConfigParameters()` function either to allow or to deny Business Manager customers to use this feature:

```
 array('moduleAllowChangeDNS', 'b', '0', 'ALLOW_CHANGE_DNS',
'ALLOW_CHANGE_DNS_DESC', NULL, 0),
```

Managing Supported TLDs List

For each registrar, you can specify a static list of supported TLDs, or retrieve the list from the registrar when needed.

➢ *To specify a static TLDs list:*

1. Set value *0* for the `REGISTRAR_GETTLDLIST` capability using the following line:

```
REGISTRAR_GETTLDLIST => 0,
```

2. Enter a list of supported TLD into the `REGISTRAR_TLDS` capability.

```
REGISTRAR_TLDS => array('com', 'net', 'org', 'biz', 'in', 'co.in','
firm.in', 'gen.in', 'ind.in', 'net.in', 'org.in', 'info', 'me',
'mobi', 'us', 'ws', 'es', 'com.es', 'org.es', 'nom.es', 'com.mx','
tv', 'bz', 'com.bz', 'net.bz',),
```

➢ *To implement the retrieving supported TLDs list capability:*

1. Set value *1* for the `REGISTRAR_GETTLDLIST` capability using the following line:

```
REGISTRAR_GETTLDLIST => 1,
```

2. Implement the `getTLDList()` function that retrieves the list of available TLDs from the registrar.

   Use the `$this->UID` (username) and `$this->PW` (password) attributes, in the request you send to the registrar.

   The function should return the array of available TLDs.

```
public function getTLDList()
{
      $registrarResponseArray = callToRegistrar(prepareRequest($this->input, $this->UID, $this->PW));
      switch ($registrarResponseArray["returnCode"]) {
            case "success":
                  $this->capabilities[REGISTRAR_TLDS] = 
$registrarResponseArray["tlds"];
                  return $this->capabilities[REGISTRAR_TLDS];
            default:
                  return array();
      }
}
```

CHAPTER 4

Writing SSL Certificate Provider Plugins

One of the functions of Business Manager is to offer SSL certificates as service plans or add-ons. The system itself does not issue certificates. After a customer fills in information about a certificate they would like to order, Business Manager forwards these data to a helping *SSL certificate provider plugin* (for simplicity, we will refer to them as *plugins*.). The plugin interacts with an SSL issuer to receive the certificate and returns the operation status back to Business Manager.

This chapter explains how to create a typical plugin. In other words, if you wish to add support for a new SSL authority or a wholesaler to Business Manager and you know their API, you can accomplish this task in 6 steps we provide later in this chapter.

A plugin is normally a PHP file, thus we assume that you have experience in PHP programming to get most out of this chapter.

We accompanied the description with two PHP files. You can find them in the `/sdk/samples/ssl/` directory. The first one, `ssl_plugin_template.php`, is the template of your featured plugin. Throughout the guide we will update this template to a working plugin and give explanations to our actions. The second file, `examplessl.php`, is result of our updates, a working plugin you can refer to and adjust to your needs.

Overview to Steps of Plugin Implementation

Here we will briefly describe each of the steps you should complete to ultimately have a full-featured plugin.

1. *Provide information about a plugin.*

   At this step you provide the details Business Manager will use to identify your plugin.

2. *Declare the plugin settings.*

   At this step you define the configuration settings administrators will use to configure your plugin in Business Manager .

3. *Provide information about certificates.*

   Here you will describe the provider's certificate plans your plugin will work with.

4. *Request extra details from customers.*

   At this step you add the feature to ask customers additional details as needed.

5. *Complete certificate issuing request.*

   This step assumes implementing the certificates issuing procedure, or in other words, interaction between Business Manager and the provider's API.

6. *Add your plugin to Business Manager.*

   Finally, you should make the system discover your new plugin.

For details on each of these steps, refer to the sections of this chapter.

Useful Information

During the plugin code writing, you might want Business Manager to provide you with debugging information such as the calls of your plugin's methods or errors. For instructions on how to receive this information, see **Debugging** (on page 81).

If you wish to translate your plugin to other languages, see how to do it in **Localization** (on page 79).

If you would like to get information about the Business Manager your plugin can use and learn more details about a particular method, consult **SSL Certificate Plugins: Reference** (on page 70).

In this chapter:

1. Providing Information About a Plugin

We assume you have extracted the plugin template `ssl_plugin_template.php`from the SDK package and opened it in a text editor. This file contains a single class handling interaction with a certain SSL authority or a certificates provider. Feel free to rename it as needed. When you design a class and file name, follow this convention: *The class name in the lower case should be the same as the SSL plugin file name.*

In our example, the class name is `ExampleSSL` and hence the SSL plugin file name will be `examplessl.php`.

After you are done with naming, provide the information about your plugin by modifying the code inside your class:

```
public function getModuleInfo()
{
      return array(
            "type"                  => "sslmodule",
            "version"         => "1.0.0",
            "displayName"     => "ExampleSSL",
            "author"          => "Parallels",
      );
}
```

When you describe your plugin, follow these rules:

- *displayName* must be equal to your class name.
- type should remain *sslmodule*.

You are free to include any other information in the remaining fields.

2. Declaring the Plugin Settings

Plugin settings are the parameters the administrators specify in Business Manager, on the plugin settings page, to start working with a certificate provider's API. For example, these can be a username to a reseller account and a password. To view the settings of an existing plugin from Business Manager, go to **All Settings** > **SSL Certificate Providers** and select a certain plugin from the list. Our sample plugin will declare the minimum settings - only username, password, and the API URL.

To specify plugin settings, substitute the empty *drawConfigParams* function with the following one. The new function adds three fields, two text fields and one password field, to the settings page of the ExampleSSL plugin. Refer to the reference for other available input types.

```
      protected function drawConfigParams(UIForm $form, array $config)
      {
            $form->addInputPanel(
                  $panel = UIFactory()-
>InputPanel(blmsg("TRANS_SSL_PLUGIN_CONFIG"))
            );
            $panel->addInputControl(UIFactory()->InputConfig(array(
                  "id"        => "url",
                  "type"            => SSLModule::CIT_TEXT,
                  "default"   => isset($config["url"]) ? $config["url"] :
"http://example-ssl.net",
                  "required"  => true,
                  "title"           =>
"TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_URL",
            )));
            $panel->addInputControl(UIFactory()->InputConfig(array(
                  "id"        => "login",
                  "type"            => SSLModule::CIT_TEXT,
                  "default"   => isset($config["login"]) ? $config["login"]
: "",
                  "required"  => true,
                  "title"           =>
"TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_LOGIN",
            )));
            $panel->addInputControl(UIFactory()->InputConfig(array(
                  "id"        => "password",
                  "type"            => SSLModule::CIT_PASSWORD,
                  "default"   => isset($config["password"]) ?
$config["password"] : "",
                  "required"  => true,
                  "title"           =>
"TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_PASSWORD",
            )));
      }
```

When one clicks **Save** on the settings page, we should receive the field values and save them to database. For this, add the *getCollectedConfigParams* function that returns the IDs you would like to save. Here is the function body (the first two lines are required):

```
      protected function getCollectedConfigParams(UIForm $form, array
$config)
      {
            $values = array();
            $form->storeValues($values);
            return array(
                  "url"       => $values["url"],
                  "login"           => $values["login"],
                  "password"  => $values["password"]
            );
      }
```

Tip: As you might have noticed from the code, the stored values become available as elements of the second array, f.e., `$config["password"]`.

Finally, to make Business Manager keep the password encrypted, instruct the *cryptedConfigParams* function to secure the *password* field.

```
public function cryptedConfigParams()
{
      return array("password");
}
```

We are done with the plugin settings. You can fetch them anytime using the *getConfig* function. Also, you can see the settings from the UI and save them.

Tip: If you view the settings page of you plugin, you will find localization keys next to each fields (like *TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_URL*). To assign proper names to these fields, add the corresponding key-value pairs to the `/opt/plesk-billing/lib/lib-billing/include/translations/` file. Learn more about the localization in the **Localization** chapter (on page 79).

3. Providing Information About Certificates

Normally a provider of SSL certificates offers several certificate types, for example, *SSL with EV* and *SSL Pro*, and assigns different prices to them. In Business Manager each of these types is a separate SSL certificate plan. This section explains how your plugin should store information about certificate types offered by a provider. Once your plugin is equipped with the information, administrators can see the prices list on the plugin settings page and choose one of these types for SSL certificate plans.

Plugins typically receive available certificate types by calling a certificate provider's API. In our example we will use hard-coded values. Say, if we have two types, *Rapid SSL* and *Comodo Instant*, we should add their names to the *getCertificates* function output and optionally provide certificate prices or *false*.

```
public function getCertificates()
{
      return array(
            'RapidSSL-RapidSSL' => '50.99',
            'Comodo-Instant' => false
      );
}
```

For each of the types, it is mandatory to set supported validity periods (in years) in the *getPeriod* function.

```
public function getPeriod($certificate)
{
      switch ($certificate) {
            case 'RapidSSL-RapidSSL':
                  return 1;
            case 'Comodo-Instant':
                  return 2;
            default:
                  return 0;
      }
}
```

Business Manager keeps a certificate type and validity period as plan properties; these properties are included into subscriptions. As a plugin developer, you should just specify some unique parameters in the output of *getCertificateAttributeName* and *getPeriodAttributeName*. We will return to these parameters later.

```
public function getCertificateAttributeName()
{
      return 'examplessl_certificate';
}

public function getPeriodAttributeName()
{
      return 'examplessl_period';
}
```

Finally you should provide the list of provider servers and let customer decide on which server to buy a certificate. This requirement came from earlier versions of Business Manager and is still actual, so you are still required to have the *webServerTypes* function. If you do not want to let customers choose between different servers, add only one item with some ID, say, 28 and make this item default (by providing 28 as the first parameter). The code should look like the following:

```
public static function webServerTypes($attributes)
{
      return array(
            28,
            array(
                  "28" => "ExampleSSl.com"
            )
      );
}
```

Now the plugin contains enough information to configure an SSL certificate plan and display it in online stores. Let's get to the essence of the plugin - write a logic of SSL certificates issuing.

4. Requesting Extra Details from Customers

An SSL providers may require additional information from a customer before issuing SSL certificates. For example, they can extract e-mails of domain administrators and let a customer choose which of the e-mails to use as the approver's e-mail. From a customer's perspective, such query is a UI notification with a link to a form for submitting extra details. If you do not need this feature, just skip this chapter.

Say, your plugin also requires the approver's e-mail to issue a certificate, so we will request each customer to choose between a list of e-mails. For sample purposes, we wrote the mock function *getApproverEmails* that returns an array of e-mails.

```
private function getApproverEmails($config, $orderId) {
      return array(
            array("admin@example.com", "admin@example.com"),
            array("root@example.com", "root@example.com")
      );
}
```

To start asking customers, you should activate the *CAPABILITY_USER_PARAMS* capability by adding it to the *__construct* function.

```
      $this->capabilities = array(
            SSLModule::CAPABILITY_USER_PARAMS
      );
```

The requests that you are up to implement (or UI notifications) can be shown in two control panels:

- In Parallels Panel
- In the legacy customer panel, *Client Panel*.

It is a good practice to let customers get notifications from both.

The Client Panel form is rendered by *drawUserParams*.

```
protected function drawUserParams(UIForm $form, array $config, array
$attributes)
{
      $form->addInputPanel(
            $panel = UIFactory()-
>InputPanel(blmsg("TRANS_SSL_PLUGIN_ADDITIONAL_DATA"))
      );
      $emails = $this->getApproverEmails($config,
$attributes["examplessl_order_id"]);
      $panel->addInputControl(UIFactory()->InputConfig(array(
            "id"        => "ApproverEmail",
            "type"            => SSLModule::CIT_SELECT,
            "default"   => isset($attributes["ApproverEmail"]) ?
$attributes["ApproverEmail"] : "",
            "required"  => true,
            "title"           =>
"TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_APPROVER_EMAIL",
            "items"           => $emails
      )));
}
```

Do not be confused by *$attributes["examplessl_order_id"]*, this variable will contain the odrer ID we will set later on.

As you can see, this function is very similar to the one that rendered the plugin settings page. As with settings, we also need to save the input to database. The *getCollectedUserParamsFromUIForm* function does this job.

```
protected function getCollectedUserParamsFromUIForm(UIForm $form, array
$config, array $attributes)
{
      $values = array();
      $form->storeValues($values);
      return array('ApproverEmail' => $values["ApproverEmail"]);
}
```

In Parallels Panel, the same form is rendered by *getParamGroupsMap*.

```
protected function getUserParamsDescriptor(array $config, array
$attributes)
{
      $emails = $this->getApproverEmails($config,
$attributes["examplessl_order_id"]);
      return array(
            "additional_data" => array(
                  array(
                        "id"        => "ApproverEmail",
                        "type"            => SSLModule::CIT_SELECT,
                        "title"           =>
blmsg("TRANS_SSL_PLUGIN_CONFIG_EXAMPLE_SSL_APPROVER_EMAIL"),
                        "required"  => true,
                        "default"   => $attributes["ApproverEmail"],
                        "items"           => $emails
                  )
            )
      );
}
```

This form is saved by *getCollectedUserParamsFromArray.*

```
protected function getCollectedUserParamsFromArray(array $config, array
$attributes, array $values)
{
      return $values;
}
```

Both forms save user input to the *ApproverEmail* key of the subscription attributes array that is available in all functions that have the *array $attributes* argument.

5. Completing Certificate Issuing Requests

Once a customer has ordered an SSL certificate, Business Manager adds an event (task) to complete the certificate issuing to the events list. The event name is *ProcessSSLCertificate of subscription NN*, where *NN* is the customer's subscription ID. This event will periodically trigger the *pluginAdd* method of your plugin until it either successfully receives the certificate or completes with an error. In terms of statuses, Business Manager expects one of the following statuses from *pluginAdd*:

- SSLModule::ACTION_STATUS_COMPLETED
- SSLModule::ACTION_STATUS_ERROR

The *pluginAdd*, in its turn, is fully responsible for issuing an SSL certificate using the data provided by a customer. Next in this section we will realize the simplest certificate issuing scenario that includes these steps:

1. Plugin sends the certificate type and the validity period to a certificate provider's API. The API responds with an order ID.
2. Plugin inquires for an approver's e-mail from the customer.
3. Plugin sends the provided details to the provider's API and either receives the certificate or signals about an error.

View the complete code of the scenario in the `examplessl.php` file, method *pluginAdd*.

The code of the first step is:

```
protected function pluginAdd(array $config, array $attributes) {
            if (!isset($attributes["examplessl_order_id"])) {
                  $orderId = $this->createOrder(
                        $config,
                        $attributes["examplessl_certificate"],
                        $attributes["examplessl_period"]
                  );
                  return array(
                        "status" => SSLModule::ACTION_STATUS_PENDING,
                        "status_reason" =>
"TRANS_EXAMPLESSL_STATUS_REASON__WAITING_FOR_ADDITIONAL_DATA",
                        "data" => array(
                              "examplessl_order_id" => $orderId
                        )
                  );
            }
}
```

Here the *$attributes["examplessl_certificate"]* and *$attributes["examplessl_period"]* will contain one of the values you defined in section **3. Providing Information About Certificates** (on page 48). Additionally, the code assigns the newly received order ID to the new *examplessl_order_id* key.

The code assumes that the *createOrder* function implements the call to the provider's API. For sample purposes, add this mock to your class:

```
private function createOrder($config, $certificate, $period) {
      return $config['login'] . '-' . $certificate . '-' . $period . '-' .
time();
}
```

The request to choose the approver's e-mail runs asynchronously, and the result of the inquiry is the *$attributes["ApproverEmail"]* value. For more details about the inquiring, see **4. Requesting Extra Details from Customers** (on page 50). The next code snippet sends the approver's e-mail to the provider's API using our mock *uploadAdditionalData* function. The code uses the *examplessl_configured* flag to determine that the additional input is successfully sent to the provider. Add the snippet to the end of your IF statement provided earlier.

```
else if (!isset($attributes["examplessl_configured"])) {
                  $orderId = $attributes["examplessl_order_id"];
                  $this->uploadAdditionalData(
                        $config,
                        $orderId,
                        $attributes["ApproverEmail"]
                  );
                  return array(
                        "status" => SSLModule::ACTION_STATUS_PENDING,
                        "status_reason" =>
"TRANS_EXAMPLESSL_STATUS_REASON__WAITING_FOR_COMPLETE_PURCHASE",
                        "data" => array(
                              "examplessl_configured" => 1
                        )
                  );
}
```

The body of the *uploadAdditionalData* mock is:

```
private function uploadAdditionalData($config, $orderId, $approverEmail) {
      return true;
}
```

After the provider receives the e-mail, the code finally realizes the third step:

```
else {
                  if ($certificate = $this->downloadCertificate($config,
$orderId)) {
                        return array(
                              "status" =>
SSLModule::ACTION_STATUS_COMPLETED,
                              "data" => array(
                                    "CSR" => $attributes["CSR"],
                                    "SSLCertificate" => $certificate
                              )
                        );
                  } else {
                        return array(
                              "status" => SSLModule::ACTION_STATUS_PENDING,
                              "status_reason" =>
"TRANS_ENOMSSL_STATUS_REASON__WAITING_FOR_COMPLETE_PURCHASE"
                        );
                  }
```

Important: The certificates are recognized by Business Manager and Parallels Panel only if you associate them with the *SSLCertificate* field.

The *downloadCertificate* is a mock, you should rewrite it according to the certificate provider's API. The body of a sample *downloadCertificate* is:

```
private function downloadCertificate($config, $orderId) {
      return 'certificate-body';
```

}

As we see from the code, *pluginAdd* finally returns the *ACTION_STATUS_COMPLETED* if everything is fine. You can surround the IF statement with the TRY and CATCH block and return *ACTION_STATUS_ERROR* in case of an error in the workflow. Now you have the fully-functional configurable plugin that requires user input. Congratulations!

Preliminarily Verification of a Request

Business Manager lets you verify that you have enough information to issue the certificate before calling *pluginAdd*. This capability is provided by the *pluginCheckAttributes* function that is called just before *pluginAdd.* If the *pluginCheckAttributes* returns a non-empty array, the *pluginAdd* is not called.

For example, check if CSR is provided by adding the following code to your plugin:

```
protected function pluginCheckAttributes(array $config, array $attributes,
$action)
{
      $errors = array();
      switch ($action) {
            case "add":
                  if (!isset($attributes["CSR"])) {
                        $errors["CSR"] = "No CSR defined";
                  }
      return $errors;
}
```

If you want to get other fields that a customer fills during the certificate ordering, find the keys that correspond to field names in the billing database and get the values by calling the *$attributes* array with the retrieved keys. In more detail:

1. Create a subscription to a plan that uses your plugin.

2. Open the *billing* database and run this query:

   ```
   SELECT package_attribute_name, package_attribute_value FROM
   package_attributes WHERE package_id = NNN;
   ```

   Here NNN stands for the subscription ID, say, 1. The result fragment can look like:

```
| countryName              | RU
```

3. Access the values you want by calling the *$attributes* array with the fetched keys. For example, a country name is available in *$attributes["countryName"]*.

6. Publishing the Plugin to Business Manager

Business Manager discovers and initializes a new plugin when you put the plugin file into this directory: `/opt/plesk-billing/lib/lib-mbapi/include/modules/sslmodule/`. Once your plugin is ready, copy it into the directory and perform a test certificate purchase to make sure the plugin works as planned.

If you encountered an error and wish to update the plugin code, simply remove the plugin from the Business Manager GUI and replace the plugin file with the new one. Business Manager should re-read it in runtime.

CHAPTER 5

Reference

This chapter contains method and field references to the API that Business Manager provides.

In this chapter:

Payment Gateways: Reference

In this section you will find references to Business Manager payment gateways configuration settings, capabilities, and payment data fields.

In this section:

Gateway Settings Format

Each payment gateway configuration parameter that Business Manager administrator will see on the **Business Setup** > **All Settings** > **Payment Gateways** > ***<gateway>*** page is defined by the following values:

1. Always *NULL*.
2. Always *NULL*.
3. *Name*. A name that will identify the parameter in the gateway code.
4. *Default value*.
5. *Group name*. Specify it if you want to group some parameters in a fieldset. Otherwise, leave the parameter empty.
6. *Label*. The name of the parameter in the Business Manager user interface.
7. *Description*. The description of the parameter in the Business Manager interface.
8. *Input type*:
   - *t* - one line text input (text field).
   - *a* - multiple lines text input (text area).
   - *b* - checkbox.
   - *p* - password input.
   - *s* - select input (dropdown list).
   - *r* - radio input (radio button).
   - *h* - hidden input.
9. *Required or no*t: *1* for required fields, *0* - for optional.
10. *Serial number*.
11. Always *0*.
12. *Additional items*. This option is used for parameters with `r` and `s` types. Leave this option as `NULL` if parameter is not radio or select input. Otherwise put an array with additional items with following item's options:
    1. Always *NULL*.
    2. Always *NULL*.
    3. Parameter value.
    4. Parameter name to display in the Business Manager user interface.
    5. Display order.

Gateway Capabilities List

The table below contains a list of payment systems capabilities that Business Manager supports and corresponding values of the `$this->params['subCommand'][0]` variable that the gateways uses to define what operation it should perform.

Capability	Variable	Operation Type	Implementation Details
Charging	`GATEWAY_CHARGE`	*charge*	**Charging** (on page 15)
Authorization	`GATEWAY_AUTHORIZE`	*auth*	**Authorization** (on page 14)
Capture	`GATEWAY_PRIOR_AUTH_CHARGE`	*finalize*	**Capture (on page 14)**
Refund	`GATEWAY_REFUND`	*refund*	**Refund (on page 16)**
Partial refund	`GATEWAY_PARTIAL_REFUND`	*refund*	**Refund (on page 16)**
Void	`GATEWAY_VOID`	*void*	**Voiding (on page 16)**
Periodical charging	`GATEWAY_RECURRING_CHARGE`		
E-check operations support	`GATEWAY_ECHECK`		
IPv6 support. Enable this capability if you want your gateway to accept IPv6 addresses of the hosts from which the payment operation is initiated.	`GATEWAY_IPV6_COMPLIANCE`		

Getting Payment Information

<will write the $this->params reference here>

To perform payment operations, you need the payment information such as card type, payment currency, charge amount, and so on. This information is available to your gateway in the `$this->params` array. This array contains the following elements:

Note: Each element is stored in the $this->params as an array, hence, to refer to the elements, use the constuctions like `$paymentAmount  = $this->params["paymentAmount"][0];`

Note: In this list we do not describe elements that have self-explanatory names.

---

1. Credit card data:
   - `billingCardExpMonth`
   - `billingCardExpYear`
   - `billingCardIssueNum`
   - `billingCardNameOnCard`
   - `billingCardStartDate`
   - `billingCardType`. May be one of the following:
     - *0* - Visa.
     - *1* - MasterCard.
     - *2* - American Express.
     - *3* - Discover.
     - *4* - JCB.
     - *5* - Enroute.
     - *6* - Diners.
     - *7* - Solo.
     - *8* - Maestro.
     - *9* - Visa-Delta.
     - *10* - Bankcard.
     - *11* - Electron.
   - `cardCVV2`. Available if the payer provides it; not available in automatic payments.
   - `cardNum` (`billingCardNum`)
2. Payment data:
   - `currencyID`. ISO 4127 code of the payment currency.
   - `invoiceID`. An ID of invoice, used if the payment
   - `invoiceName`

- `invoiceNumber`
- `paymentAmount`
- `paymentDescription`
- `paymentMethod`. May be one of the following:
  - *CC* - credit card payment.
  - *ECHECK* - e-check payment.
- `subCommand`. May be one of the following:
  - *auth* - check the availability of the `paymentAmount` on the payer's card and block this amount.
  - *finalize* - perform the authorized payment: transfer the `paymentAmount` from the payer's card to seller's merchant account.
  - *charge* - perform *auth* and *finalize* successively.
  - *refund* - return the `paymentAmount` to the payer's card.
  - *void* - cancel the *auth* action: free the blocked `paymentAmount` on the payer's card.

3. Payment transaction data:
   - `authCode`. Authorization code that the gateway had received from the payment system earlier, available for *finalize*, *refund*, and *void* operations.
   - `transactionCode`. Transaction identifier that the gateway had received from the payment system earlier, available for *finalize*, *refund*, and *void* operations.
4. Customer contact data:
   - `contactFirstName`
   - `contactLastName`
   - `contactEmail`
   - `contactAddress1`
   - `contactAddress2`
   - `contactCity`
   - `contactCounty`
   - `contactState`
   - `contactZip`
   - `contactPhone1`
   - `contactPhone2`
   - `contactFax`
   - `contactMobilePhone`
   - `countriesISO2`. Two-letter ISO 3166-1 code of the payer's country.
5. E-check data:
   - `billingBankAbaCode`

- `billingBankAccountNum`
- `billingBankDob`
- `billingBankLicenseNum`
- `billingBankName`
- `billingBankState`
- `billingBankType`. May be one of the following:
  - *0* - Checking.
  - *1* - Savings.
  - *2* - Checking (Business).
  - *3* - Savings (Business).

Domain Name Registrar Plugins: Reference

This section provides information about domain name registrar modules capabilities, methods and their input and output data.

In this section:

Plugin Settings Format

Each plugin configuration parameter that Business Manager administrator will see on the **Business Setup** > **All Settings** > **Registrar Modules** > ***<registrar>*** page is defined by the following values:

1. *Name*. A name that will identify the parameter in the plugin code.
2. *Type*:
   - *t* - text.
   - *p* - password - a text field with hidden content.
   - *r* - radio button - an option to choose one of the multiple values.
3. *Default value*.
4. *Label*. The name of the parameter in the Business Manager user interface.
5. *Description*. The description of the parameter in the Business Manager interface.
6. *Available values for radio button parameters*. Each parameter is an array of the following values:
   1. Always *NULL*.
   2. Always *NULL*.
   3. Parameter value.
   4. Parameter name to display in the Business Manager user interface.
   5. Display order.
7. *Required or no*t. *1* for required fields; *0* - for optional.

Available Capabilities

The full list of registrar capabilities that Business Manager supports is provided in the table below. You can find short descriptions of these capabilities and the functions you should implement to get the capabilities work in the base class `Registrar.php` located in the `opt/plesk-billing/lib/lib-mbapi/include/modules/registrar` directory.

Capability	Variable	Function	Implementation Details
Register domain names	`REGISTRAR_REGISTERDOMAIN`	`registerDomain()`	**Domain Registration (on page 24)**
Transfer domain names	`REGISTRAR_TRANSFERDOMAIN`	`transferDomain()`	**Domain Transfer (on page 26)**
Renew domain name	`REGISTRAR_RENEWDOMAIN`	`renewDomain()`	**Domain Renewal (on page 29)**

Return domain extended information	REGISTRAR_GETDOMAINEXTENDEDINFO	getDomainExtendedInfo()	**Configuring Manual Renewal (on page 32)**
Perform a live renewal using Business Manager	REGISTRAR_LIVERENEW	Auxiliary variable, used for configuring manual domain renewal.	**Configuring Manual Renewal (on page 32)**
Return registrant contact info	REGISTRAR_GETDOMAINCONTACTDATA	getDomainContactData()	**Managing Contact Information (on page 34)**
Update registrant contact info	REGISTRAR_SETDOMAINCONTACTDATA	setDomainContactData()	**Managing Contact Information (on page 34)**
Return domain renewal type (manual or automatic)	REGISTRAR_GETDOMAINRENEWTYPEOPTIONS	getDomainRenewTypeOptions()	**Managing Renewal Settings (on page 31)**
Update domain renewal type (manual or automatic)	REGISTRAR_SETDOMAINRENEWTYPEOPTIONS	setDomainRenewTypeOptions()	**Managing Renewal Settings (on page 31)**
Return domain transfer lock options	REGISTRAR_GETDOMAINREGISTRARLOCKOPTIONS	getDomainRegistrarLockOption()	**Managing Transfer Lock (on page 28)**
Update domain transfer lock options	REGISTRAR_SETDOMAINREGISTRARLOCKOPTIONS	setDomainRegistrarLockOption()	**Managing Transfer Lock (on page 28)**
Return domain access password	REGISTRAR_GETDOMAINACCESSPASSWORD	getDomainAccessPassword()	**Managing Access Password (on page 38)**
Update domain access password	REGISTRAR_SETDOMAINACCESSPASSWORD	setDomainAccessPassword()	**Managing Access Password (on page 38)**
Return domain DNS settings	REGISTRAR_GETDOMAINDNS	getDomainDNS()	**Managing DNS Settings (on page 39)**
Update domain DNS settings	REGISTRAR_SETDOMAINDNS	setDomainDNS()	**Managing DNS Settings (on page 39)**
Return a list of available top level domains	REGISTRAR_GETTLDLIST	getTLDList()	**Managing Supported TLDs List (on page 41)**
Return domain expiration date	REGISTRAR_GETDOMAINEXPDA	getExpDate()	Not provided in this document.

Return domain availability	`REGISTRAR_CH ECKDOMAINAVA`	`checkDomainAvai lability()`	Not provided in this document.
Return domain transfer availability	`REGISTRAR_CH ECKDOMAINTRA`	`checkDomainTran sferAvailabilit`	Not provided in this document.
Return a reseller account balance	`REGISTRAR_CH ECKACCOUNTBA`	`checkAccountBal ance()`	Not provided in this document.
Return all domains of an account	`REGISTRAR_GE TDOMAINS`	`getDomains()`	Not provided in this document.

Getting Domain Information

For performing operations with domain names, you need information that the domain owner provided in the online store, say, top and second level domains or preferred DNS servers. This information is stored in the `$this->input` array. Particularly, this array contains the following data:

1. Domain attributes:
   - `domainTLD`. Top level domain - a part of the domain name after the last period ('.'), For example, in *example.com*, the top level domain it *com.*
   - `domainSLD`. Second level domain - a part of a domain name that is directly below a TLD. For example, in *example.com*, the second level domain is *example.*
   - `domainYears`. Domain registration period in years.
   - `domainNameserver1`. Primary name server URL.
   - `domainNameserver2`. Secondary name server URL.
   - `domainNameserver3`. Secondary name server URL.
   - `domainNameserver4`. Secondary name server URL.
   - `domainTransferKey`. Domain transfer key (or *transfer secret*).
   - `domainRegistrarLock`.. Domain transfer lock status: *1* if the lock is enabled, *0* - if disabled.
   - `domainRenewType`. Domain renewal type: *0* if the automatic renewal is enabled, *1* - if disabled.
   - `domainAccessPassword`. Domain access password.
2. Registrant data:

   Note: Hereinafter, we do not provide explanations of the contact information fields since their names are self-explanatory.

   - `registrantAddress1`
   - `registrantAddress2`
   - `registrantCity`
   - `registrantCountry`
   - `registrantEmailAddress`
   - `registrantFax`
   - `registrantFirstName`
   - `registrantJobTitle`
   - `registrantLastName`
   - `registrantOrganizationName`
   - `registrantPhone`
   - `registrantPhoneExt`

- `registrantPostalCode`
- `registrantStateProvince`
- `registrantStateProvinceChoice.`

3. Administrative contact data:
   - `adminAddress1`
   - `adminAddress2`
   - `adminCity`
   - `adminCountry`
   - `adminEmailAddress`
   - `adminFax`
   - `adminFirstName`
   - `adminJobTitle`
   - `adminLastName`
   - `adminOrganizationName`
   - `adminPhone`
   - `adminPhoneExt`
   - `adminPostalCode`
   - `adminStateProvince`
   - `adminStateProvinceChoice`
4. Billing contact data:
   - `billingAddress1`
   - `billingAddress2`
   - `billingCity`
   - `billingFullCountry`
   - `billingCountry`
   - `billingEmailAddress`
   - `billingFax`
   - `billingFirstName`
   - `billingJobTitle`
   - `billingLastName`
   - `billingOrganizationName`
   - `billingPhone`
   - `billingPhoneExt`
   - `billingPostalCode`
   - `billingStateProvince`
   - `billingStateProvinceChoice`

5. Auxiliary billing contact data:
   - `auxAddress1`
   - `auxAddress2`
   - `auxCity`
   - `auxCountry`
   - `auxEmailAddress`
   - `auxFax`
   - `auxFirstName`
   - `auxJobTitle`
   - `auxLastName`
   - `auxOrganizationName`
   - `auxPhone`
   - `auxPhoneExt`
   - `auxPostalCode`
   - `auxStateProvince`
   - `auxStateProvinceChoice`
6. Technical contact data:
   - `techAddress1`
   - `techAddress2`
   - `techCity`
   - `techCountry`
   - `techEmailAddress`
   - `techFax`
   - `techFirstName`
   - `techJobTitle`
   - `techLastName`
   - `techOrganizationName`
   - `techPhone`
   - `techPhoneExt`
   - `techPostalCode`
   - `techStateProvince`
   - `techStateProvinceChoice`

Output Contact Information Fields

When you retrieve contact information of a domain, store it in the following elements of the `arrayData` array:

- `$this->arrayData['RegistrantContactData']: registrantAddress1, registrantAddress2, registrantCity, registrantCountry, registrantEmailAddress, registrantFax, registrantFirstName, registrantJobTitle, registrantLastName, registrantOrganizationName, registrantPhone, registrantPhoneExt, registrantPostalCode, registrantStateProvince, registrantStateProvinceChoice.`
- `$this->arrayData['AuxBillingContactData']: auxAddress1, auxAddress2, auxCity, auxCountry, auxEmailAddress, "auxFax, auxFirstName, auxJobTitle, auxLastName, auxOrganizationName, auxPhone, auxPhoneExt, auxPostalCode, auxStateProvince, auxStateProvinceChoice.`
- `$this->arrayData['TechContactData']: techAddress1, techAddress2, techCity, techCountry, techEmailAddress, techFax, techFirstName, techJobTitle, techLastName, techOrganizationName, techPhone, techPhoneExt, techPostalCode", techStateProvince, techStateProvinceChoice.`
- `$this->arrayData['AdminContactData']: adminAddress1, adminAddress2, "adminCity, adminCountry, adminEmailAddress, adminFax, adminFirstName, adminJobTitle, adminLastName, adminOrganizationName, adminPhone, adminPhoneExt, adminPostalCode", adminStateProvince, adminStateProvinceChoice.`
- `$this->arrayData['BillingContactData']: billingAddress1, billingAddress2, billingCity, billingFullCountry, billingCountry, billingEmailAddress, billingFax, billingFirstName, billingJobTitle, billingLastName, billingOrganizationName, billingPhone, billingPhoneExt, billingPostalCode, billingStateProvince, billingStateProvinceChoice.`

CHAPTER 6

SSL Certificate Plugins: Reference

This section contains method and field reference of the Business Manager API for writing SSL certificate plugins.

In this chapter:

Plugin Capabilities

You can add capabilities to the plugin constructor into the array *$this->capabilities*. Depending on the array values the system defines how to work with a plugin and what plugin functions to call. To check if a plugin realizes a capability, use the isCapable method defined in the base class and accepting a capability ID, one of these constants:

- *SSLModule::CAPABILITY_USER_PARAMS*
  The capability defines if a plugin requests additional input from customers. If the capability is on, the respective forms will be displayed in Parallels Panel and Client Panel, the legacy panel of Business Manager.

- *SSLModule::CAPABILITY_PRODUCT_PARAMS*
  Is not supported since 10.3. It defines if a plugin renders a form with SSL certificate plan-specific parameters (the provider name, certificate name, validity period, that an administrator can fill in when editing a plan).

Public Methods

This section lists the public methods of the *ExampleSSL* abstract class your plugin can use or redefine.

Name: *__construct*

Description: The plugin constructor that initializes a plugin. During the initialization, Business Manager retrieves plugin instance settings and associated SSL certificate plan properties. redefine the method if you want to turn on some plugin capabilities.

Name: *getModuleInfo*

Description: Returns an associative array that contains information about a plugin. You should redefine it; for details, refer to **1. Providing Information About a Plugin** (on page 45).

Name: *getModuleInfoData*

Description: Not used.

Name: *getCapabilities*

Description: Returns the *$this->capabilities* array. We recommend that you do not redefine this method.

Name: *isValid*

Description: Returns true if a plugin instance was correctly initialized by Business Manager. We recommend that you do not redefine this method.

Name: *addConfigParams*

Description: Renders a plugin settings form in Business Manager. We recommend that you do not redefine this method; redefine the *drawConfigParams* method instead.

Name: *collectConfigParams*

Description: Saves and retrieves plugin settings to database . We recommend that you do not redefine this method; redefine the *getCollectedConfigParams* method instead.

Name: *addProductParams*

Description: Not used since 10.3.

Name: *collectProductParams*

Description: Not used since 10.3.

Name: *addUserParams*

Description: Renders a from which requests customers to provide additional input. We recommend that you do not redefine this method; redefine the *drawUserParams* instead.

Name: *getUserParams*

Description: Returns the description of form for inputting additional details in Parallels Panel. We recommend that you do not redefine this method; redefine the *getUserParamsDescriptor* instead.

Name: *collectUserParams*

Description: Stores and retrieves parameters filled in the *addUserParams* method. We recommend that you do not redefine this method; redefine the *getCollectedUserParamsFromUIForm* instead.

Name: *collectUserParamsFromArray*

Description: Stores and retrieves parameters filled in the *collectUserParams* method. We recommend that you do not redefine this method; redefine the *getCollectedUserParamsFromArray* instead.

Name: *cryptedConfigParams*

Description: Returns the array of form field IDs that should be crypted. Use this method in conjunction with *drawConfigParams*.

Name: *cryptedProductParams*

Description: Not used since 10.3.

Name: *cryptedUserParams*

Description: Returns the array of form field names; these fields (customers fill as additional information to issue a certificate) will be encrypted when saved to database.

Name: *getReturnXML*

Description: Not used.

Name: *isDone*

Description: Returns *true* if one of the following methods returned a status other than SSLModule::ACTION_STATUS_PENDING: *pluginAdd, pluginRenew, pluginDelete, pluginGetIssuedCert,* or *pluginGetCSR.* For details about the status, see **Status Descriptions** (on page 78).

Name: *isCapable*

Description: Returns *true* if a given capability is supported by a plugin.

Name: *getParamGroupsMap*

Description: Returns an associative array defining the correspondency between form fields for additional input and localization constants.

Name: *getDomainOpts*

Description: Returns an associative array containing the information about a domain; such domains are associated with a subscription to an SSL certificate plan linked with this plugin. Not used since 10.2. Do not redefine this method.

Name: *getDomainName*

Description: Returns an associative array containing the information about a domain; such domains are associated with a subscription to an SSL certificate plan linked with this plugin. Not used since 10.2. Do not redefine this method.

Other public methods are used for internal purposes of the system, neither redefine them nor use in your code.

Protected Methods

This section lists the protected methods of the *ExampleSSL* abstract class your plugin can use or redefine.

Name: *pluginCheckAttributes*

Description: Performs validation of user input in forms of certificates order and additional input forms in Client Panel and Parallels Panel.

Name: *getCountries*

Description: Returns a list of countries customers select from in online stores.

Name: *getUSStates*

Description: Returns a list the USA states.

Name: *getConfig*

Description: Returns a list of plugin configuration settings defined in *drawConfigParams.*

Name: *isStandAloneCert*

Description: Returns *true* if a certificate was ordered as a separate plan. If the certificate plan was an add-on to a hosting plan, the method returns *false*.

Name: *generateCSR*

Description: Generates a CSR using contact information and Common Name provided during the ordering procedure. The method returns an associative array with the keys *csr* and *pvt*.

Name: *pushErrorToAdminsToDo*

Description: Recieves a text and adds it as a to-do item for an administrator.

Abstract Methods

This section lists the abstract methods of the *ExampleSSL* abstract class your plugin must define.

Name: *getCertificates*

Description: Returns an associative array which keys are certificate types received from a provider of SSL certificates. This method is used to list available certificate types on the plugin settings page.

Name: *getPeriod*

Description: Returns a maximum period for a given certificate type.

Name: *getCertificateAttributeName*

Description: Returns a plan property that keeps the certificate type.

Name: *getPeriodAttributeName*

Description: Returns a plan property that keeps the period assigned to a certain certificate.

Name: *getParamGroupsMap*

Description: Returns an associative array defining the correspondency between form fields for additional input and localization constants.

Name: *pluginAdd*

Description: Prepares data and sends requests for new certificates to the API of a certificate provider. This method must return an associative array which contains the *status* key that goes with one of these values: either *self::ACTION_STATUS_ERROR* or *self::ACTION_STATUS_COMPLETED*. Moreover, the successful response should contain the key data which must be an associative array with required fields *CSR* and *SSLCertificate*. For details about the statuses, see **Status Descriptions** (on page 78).

Name: *pluginRenew*

Description: Prepares data and sends requests to renew existing certificates by means of a certificate provider's API. This method must return an associative array which contains the *status* key that goes with one of these values: either *self::ACTION_STATUS_ERROR* or *self::ACTION_STATUS_COMPLETED*. Moreover, the successful response should contain the key data which must be an associative array with required fields *CSR* and *SSLCertificate*. For details about the statuses, see **Status Descriptions** (on page 78).

Name: *pluginDelete*

Description: Prepares data and sends requests to cancel a certificates by means of a certificate provider's API. This method must return an associative array which contains the *status* key that goes with one of these values: either *self::ACTION_STATUS_ERROR* or *self::ACTION_STATUS_COMPLETED*. For details about the statuses, see **Status Descriptions** (on page 78).

Name: *pluginGetIssuedCert*

Description: Not used, leave the method body empty.

Name: *pluginGetCSR*

Description: Not used, leave the method body empty.

Name: *drawConfigParams*

Description: Renders a form for adding or editing a provider using Simple CBM UI Framework.

Name: *getCollectedConfigParams*

Description: Processes user input after saving the plugin settings form and returns an associated array which contains configuration parameters and values. Business Manager saves these parameters to database.

Name: *drawProductParams*

Description: Not used since 10.3, leave the method body empty.

Name: *getCollectedProductParams*

Description: Not used since 10.3, leave the method body empty.

Name: *drawUserParams*

Description: Renders a form that gets additional information from customers. This method runs only if the *self::CAPABILITY_USER_PARAMS* capability is on.

Name: *getCollectedUserParamsFromUIForm*

Description: Validates and saves input of the parameters received by *drawUserParams* to database. These parameters are saved as subscription properties. This method runs only if the *self::CAPABILITY_USER_PARAMS* capability is on.

Name: *getUserParamsDescriptor*

Description: Returns a two-level associative array that describes a form for submitting additional input. The first level sets field groups (Business Manager uses *getParamGroupsMap* to set field group labels). The second level sets fields within the groups. Each such group is also an associated array. For samples, see 4. **Requesting Extra Details from Customers** (on page 50). This method runs only if the *self::CAPABILITY_USER_PARAMS* capability is on.

Name: *getCollectedUserParamsFromArray*

Description: Validates data submitted into a form *getCollectedUserParamsFromArray* and saves the results to database. These parameters are saved as subscription properties. This method runs only if the *self::CAPABILITY_USER_PARAMS* capability is on.

Name: *webServerTypes*

Description: Returns the list of web servers which provide certificates. The return value is an array. The first array key is designates the ID of a default server. The second element is an associative array that contains server IDs and values. The default value should match one of the server IDs.

Status Descriptions

The methods that plugin uses to interact with Business Manager (*pluginAdd*, *pluginRenew*, *pluginDelete*, *pluginGetIssuedCert* and *pluginGetCSR*) return associative arrays containing the required key *status*. Below we provide details on how Business Manager interprets each of the statuses.

- *SSLModule::ACTION_STATUS_PENDING*

  The request is sent to an SSL certificate provider. Business Manager creates an event to repeat the method every 5 minutes until receiving one of the next statuses.
- *SSLModule::ACTION_STATUS_COMPLETED*

  The certificate is successfully issued. Business Manager assigns a certificate to a customer and will not repeat the method anymore.
- *SSLModule::ACTION_STATUS_ERROR*

  The request was completed with an error. The system will not repeat any further requests.

Plugin Settings

Plugin settings are defined on plugin creation, they are stored in database and are available in code through the protected method *getConfig.* This method returns settings as an associative array. There are no limitations on the number of settings a plugin declares - these could be any parameters that are required to interact with a provider's API.

The settings are rendered by the *drawConfigParams* method. To make Business Manager start handling these fields, like saving and loading them into the form, use the *getCollectedConfigParams* method. For samples, refer to **2. Declaring the Plugin Settings** (on page 46).

C H A P T E R 7

Localization

When you develop plugins and modules, consider using localization keys instead of actual values. For example, if you need a group label *Configuration*, insert a localization key instead of the actual label value. Normally localization keys are started with *TRANS* and they are in uppercase. A sample key can look like this: *TRANS_SSL_PLUGIN_CONFIG*.

The localization mechanism is straightforward:

1. Specify a key instead of a value in your code.
2. Create a PHP file called `en.php` in the `opt/plesk-billing/lib/lib-mbapi/include/modules/`***<module_type>***`/translations/`***<module name>***`/` directory, where the ***<module name>*** is the name of your module and the ***<module_type>*** is the following:
   - `gateway` for payment gateways.
   - `registrar` for domain name registrar plugins.
   - `sslmodule` for SSL certificate providers plugins
3. Define the key-value mapping in the `en.php` file. The file should look like the following:

```
<?php
$translation_keys = array(
      "<key1_name>" => "<value1>",
      "<key2_name>" => "<value2>",
      ...
 );
```

For example, use the line provided below to show the word *Configuration* instead of the key *TRANS_SSL_PLUGIN_CONFIG* that you write in your code:

```
"TRANS_SSL_PLUGIN_CONFIG" => "Configuration",
```

If you use localization keys, you are able to easily translate your plugins to other languages. To do this, create a copy of the `en.php` in the same directory, rename the copy in correspondence with the table below, and finally translate the values in the key-value pairs.

Locale	File name
German	`de.php`
English	`en.php`
Spanish	`es.php`
French	`fr.php`
Italian	`it.php`
Japanese	`ja.php`

Dutch	`nl.php`
Polish	`pl.php`
Russian	`ru.php`
Simplified Chinese	`zh_Hans.php`
Traditional Chinese	`zh_Hant.php`

CHAPTER 8

Debugging

To output the debugging information, you can use two ways:

- The standard PHP function `error_log()`. This function outputs the error information to the `/var/log/sw-cp-server/error_log` file.
- Business Manager function `dbg()`. You can view the output information of this function on the page **Business Setup** > **All Settings** > **Debug Log** of the Business Manager administrator interface.

The system also saves the debug log in the Business Manager database, table `dbg_entries`.

To find a debug entry by a keyword in the database, run the following commands in the Business Manager server shell:

- On Linux:

```
/usr/share/plesk-billing/billing-db
select * from dbg_entries where dbg_entry_message like '<keyword>';
```

- On Windows:

```
%plesk-dir%\admin\bin\billing-db
select * from dbg_entries where dbg_entry_message like '<keyword>';
```

➢ *To enable the `dbg()` function:*

1. Set the `TRUE` value for the `ALLOW_DEBUG_LOGGING` constant in the file `opt/plesk-billing/lib-billing/include/config/config.php.`
2. Enable the debug logging on the page **Business Setup** > **All Settings** > **Debug Log** of the Business Manager administrator interface.