# Caspio Payment System Technical Requirements

#### Draft specification

**UPDATE/CHANGE LOG**

|  |  |  |
| --- | --- | --- |
| Date | Version | Comments |
| 11/30/12 | v.1.0 | Initial version |
| 12/6/12 | V 1.1 | Added UI Screens Explanation section Fixed incorrect CPSCheckPromotionalCode method signature |

## About this document

This document describes system requirements, overall architecture, and some additional technical assumptions needed to build UI part of Caspio Payment System (“CPS” from here). Also, it provides information about BridgePay web service and its methods required to integrate CPS UI with system backend.

This document is not a functional specification for CPS UI.

This document is not a developer’s reference of BridgePay service.

This document does not describe CPS backend.

## CPS System Requirements

CPS application must meet the following requirements:

1. Written using .Net Framework 4.0 (C# or VB.Net)
2. ASP.Net Web Application, target platform – x86, utilize VisualStudio 2010 solution and project format
3. Do not include references to any 3rd party frameworks or components, only standard Microsoft assemblies are allowed
4. Be able to run on the following run-time configuration:
   1. Windows Server 2008 R2 (64-bit)
   2. IIS 7.5
   3. 4GB RAM
5. Must be able to work in session-less web farm/clustered web servers environment and cannot rely on any kind of session, including database-based. It is recommended (although not mandatory) to use client-side cookies/webStorage if there is any need to maintain sessions. If cookies/webStorage are used, they should be session ones and not be preserved between browser sessions. Also, they should allow different customer accounts (different “sessionKey”s, see below) to use CPS UI one after another without undesirable side effects, i.e. these “sessions” should be different for different customer account, even though they are created during the same browser session.
6. Support the following client browsers
   1. Internet Explorer 8 or higher
   2. Mozilla Firefox for Windows, Linux and Android – latest available version (v17 for Windows at the moment)
   3. Google Chrome – latest available version (v23 at the moment)
   4. Safari 5.1 (WebKit2) or higher - Windows and Mac
   5. Mobile Safari on iOS version 5 or higher. CPS UI must be fully functional on iPad 2 or higher
   6. Default Android browser (with possible limitations, see 7 below)
7. Must be overall touch-screens friendly, i.e. support sequential taps instead of double-clicks, allow scroll and zoom gestures, etc. CPS UI may have some limitation in user experience on smaller iOS5+ devices (iPad Mini, iPhone, iPod touch) and/or Android-based systems because of small screen size, but not because of limited browser capabilities (if any) on these platforms.

***Note***. Some other technical requirements are mentioned below in the text, highlighted with yellow background

## Overall architecture

The main purpose of CPS UI application is to allow Caspio customers (“accounts”) to switch between different “account packages” that determines their recurring service payment. Each account package has its own base fee, set of possibly assigned add-ons, discounts and promotions. In addition to that, each account can have different billing cycle that determines the period when recurring fees occur.

CPS UI is a separate front-end project/application that communicates with CPS backend by means of separate BridgePay web service. CPS UI should consider BridgePay as a “black box” that performs all actions and returns all information needed by CPS UI. URL to BridgePay web service should not be hardcoded and stored in CPS UI web.config file as configurable value.

All calls to BridgePay service should be made using HTTPS. Application should use https:// as protocol prefix even if the configured BridgePay URL value starts with http://, unless this is debug build of application (#if DEBUG)

BridgePay is SOAP-based web service that is used by different, other than CPS UI, components of the existing product. Most of the methods described in this document have CPS prefix in their name to highlight the fact that are used exclusively by CPS UI application. CPS UI cannot communicate to any other components or parts of the system in any way but via BridgePay web service.

## Authentication, Security and Login

Since CPS UI and BridgePay web service are session-less, they have to validate user credentials during each call. This is achieved through 2 special parameters that passed to all of CPSxxxx methods. These parameters are called “key” and “sessionKey”. The “key” parameter is a string used as a secure token across the system’s components to verify the fact that caller can be trusted. It must be configurable via CPS UI web.config file

The “sessionKey” parameter is a long random string stored in browser cookie “SESSIONKEY”. The cookie is set if CPS UI user was previously logged in his/her account via main UI console and can be used by backend to determine account ID and profile of current user. From CPS UI application perspective, if this cookie is present in client request, all CPS UI needs to do is to pass its value into BridgePay web service. If, for any reason, CPS UI application needs to check validity of sessionKey, it can use CPSValidateSession method.

In a case when sessionKey cookie is not present in client request, CPS UI should not allow user to proceed but perform user login instead. This is achieved via the call of CPSValidateAccountPassword BridgePay method. If login was successful, the method returns newly generated sessionKey that CPS UI must store in “sessionKey” response cookie. Also, the .Domain property of the cookie must be set to 2nd level domain of the site CPS UI is running on. For example, if CPS UI is deployed on cps.company.com, the cookie.Domain property must be set to “company.com”

CPS UI should not allow user to access any of application pages (URLs) without checking for valid sessionKey. All HTTP requests that lack sessionKey cookie must be redirected to CPS UI login form.

If, for any reasons, CPS UI needs to log out the current user, the recommended way to do it is by resetting “sessionKey” cookie value to empty string. No call to BridgePay service is needed.

Some of so called “lookup” BridgePay methods return the information that is not specific to the particular account. These methods (e.g. ListCountries and ListIndustries) do not require sessionKey, but do require key parameter as a proof that call was made by one of the registered system components.

## BridgePay web service

As mentioned above, CPS UI must utilize BridgePay web service to communicate with backend. This section provides the information about BridgePay methods needed to achieve the above goal. See also BridgePay.wsdl, attached as separate document

#### Common parameters

|  |  |
| --- | --- |
| **Name** | **Description** |
| key | System security token, see its description in section IV above |
| sessionKey | User session security token, see its description in section IV above |

#### Methods

* ***void CPSValidateSession(string key, string sessionKey)***  
  Validates particular sessionKey. Throws exception BPWS-002 if session is invalid or expired or account wasn’t logged in correctly, returns nothing otherwise.
* ***string CPSValidateAccountPassword(string key, string AccountName, string UserName, string Password)***

Performs account login. Returns account security token that must be set as sessionKey cookie value (see section IV above) and to be used as session key. If login was unsuccessful, throws BPWS-002 exception

* ***string CPSGetCurrentAccountPackage(string key, string sessionKey)***

Returns XML string with information about current account’s package, in the following format:  
<xml>

<package>

<packageName>**PackageName**</packageName>

<packageCategory>**PackageCategory**</packageCategory>

<baseFee>**49.99**</baseFee>

<baseFeeAbsoluteFormatted>**49.99**</baseFeeAbsoluteFormatted>

<billingCycle>**1**</billingCycle>

<recurringDiscounts>

<discount>

<title>**SomeDiscountName**</title>

<description>**SomeDiscountDescription**</description>

<amount>**12.00**</amount>

<amountAbsoluteFormatted>**12.00**</amountAbsoluteFormatted>

</discount>

…. <!- extra discounts -->

</recurringDiscounts>

<recurringFees>

<fee isAdditionalResource="{**True|False**}" addOnId="**101**" [addOnGroup=”**SomeGroup**”]>

<title>**SomeAddOnTitle**</title>

<description>**SomeAddOnDescription**</description>

<amount>**9.99**</amount>

<amountAbsoluteFormatted>**9.99**</amountAbsoluteFormatted>

</fee>

…. <!- extra fees -->

</recurringFees>

<recurringTotal>**47.98**</recurringTotal>

<recurringTotalAbsoluteFormatted>**47.98**</recurringTotalAbsoluteFormatted>

<nonRecurringTotal>**0.00**</nonRecurringTotal>

<nonRecurringTotalAbsoluteFormatted>**0.00**</nonRecurringTotalAbsoluteFormatted>

<creditsTotal>**0.00**</creditsTotal>

<creditsTotalAbsoluteFormatted>**0.00**</creditsTotalAbsoluteFormatted>

<isNewBillingSystem>**True**</isNewBillingSystem>

<isAutoBill>**True**</isAutoBill>

<invoicesBalanceTotal>**29.99**</invoicesBalanceTotal>

<invoicesBalanceTotalAbsoluteFormatted>**29.99**</invoicesBalanceTotalAbsoluteFormatted>

</package>

</xml>

-billingCycle contains number of month in account’s billing cycle – 1 (charged monthly), 3 (quarterly) or 12 (annualy)

- all amounts are decimal numbers formatted using “0.00” pattern, “absoluteFormatted” amounts are decimals formatted using “###,##0.00” pattern

- base recurring price of account package is returned as **baseFee**, selected add-ons are returned as **recurringFees** and account total recurring price is returned as **recurringTotal.** Other fields are not relevant for CPS UI

Note. There is a special treatment of accounts with “Business” package categories. In general, they have so called “additional resources” (DataPages, users, storage quota, etc.) while other account categories use unlimited resources. Attribute value isAddtionalResource=”True” is used to tell additional resource for Business packages from regular add-ons.

* ***DataTable CPSGetAccountPackages(string key, string sessionKey)***

Returns DataTable with the following fields:

* + acct\_type\_id – int32, PackageId to be used as parameters in other BridgePay methods
  + acct\_type\_name – string, package name
  + bandwidth – int32, allowed data transfer in MBytes
  + category\_name – string, Package category name
  + standard\_fee – decimal, account monthly price
  + data\_transfer\_unit – int32, size of excess usage unit in MBytes
  + data\_transfer\_fee – decimal, price for one excess usage unit
  + data\_pages\_unit – int32, number of extra DataPages
  + data\_pages\_fee – decimal, price for one extra DataPage
  + emails\_unit – int32, number of extra emails available to be sent
  + emails\_fee – decimal, price for one extra email
  + storage\_unit-– int32, size of extra storage in MBytes
  + stogare\_fee – decimal, price for one extra storage unit
* ***string CPSGetAvailableAddOns(string key, string sessionKey, string newPackageName)***

Returns XML string with information about add-ons available for particular package, in the following package

<AddOns>

<Group Name=”**Support**”>

<AddOn>

<FeeId>**1234**</FeeId>

<AddOnId>**1111**</AddOnId>

<Title>**SomeTitle**</Title>

<Description>**SomeDescription**</Description>

<Checked>**True**</Checked>

<CanBeChanged>**False**</CanBeChanged>

<SetUpFee>**0.00**</SetUpFee>

<SetUpFeeAbsoluteFormatted>**0.00**</SetUpFeeAbsoluteFormatted>

<MonthlyFee>**79.99**</MonthlyFee>

<MonthlyFeeAbsoluteFormatted>**79.99**</MonthlyFeeAbsoluteFormatted>

<Rewritten>**False**</Rewritten>

<Automated>**True**</Automated>

</AddOn>

</Group>

<Group Name=”Other”>

…<!—other mutually exclusive add-on group -->

</Group>

<!—other, non-grouped add-ons that can be selected independently -->

</AddOns>

* ***DataTable ListIndustries(string key)***

Returns DataTable with the following fields:

* + industry\_id – int32, ID
  + industry\_code – string, code
  + industry\_name – string, description
* ***DataTable ListCountries(string key)***

Returns DataTable with the following fields:

* + CountryId – int32, ID
  + CountryCode – string, code
  + Name – string, country name
* **DataTable CPSGetContactInfo(string key, string sessionKey)**

Returns DataTable with the following fields:

* + contact\_first\_name – string, first name
  + contact\_last\_name – string, last name
  + title – string, contact person occupation title
  + company - string, company name
  + company\_size - string, number of company employees, e.g. “51-100”
  + industry - string, name of the company industry
  + phone - string, contact phone
  + secondary\_phone - string, secondary contact phone
  + email - string, email address
  + url- - string, account site URL
  + address1 - string, contact street address line 1
  + address2 - string, contact street address line 2
  + city- string, contact address city
  + state - string, contact state
  + country - string, contact country
  + postalcode - string, contact ZIP
* ***void CPSSetContactInfo(string key, string sessionKey, string firstName, string lastName, string title, string company, string employees, string industry, string phone, string secondaryPhone, string email, string url, string address1, string address2, string city, string state, string country, string zip)***

Updates the account’s contact information, generates exceptions if some parameters are incorrect

* ***DataTable CPSGetBillingInfo(string key, string sessionKey)***

Returns DataTable with the following fields:

* + billing\_first\_name – string, billing first name
  + billing\_last\_name – string, billing last name
  + billing\_address1 - string, billing street address line 1
  + billing\_address2 - string, billing street address line 2
  + billing\_city- string, billing address city
  + billing\_state - string, billing state
  + billing\_postalcode - string, billing ZIP
  + billing\_country - string, billing country
  + billing\_phone - string, billing phone
  + billing\_email - string, billing email address
  + cc\_number – string, credit card number
  + cc\_type – string, credit card type
  + cc\_exp\_month – string, credit card expiration month
  + cc\_exp\_year – string, credit card expiration year
  + send\_receipt\_email - bool, flag that show the account receives email receipts
  + receipt\_emails – string, semicolon-separated list of email addresses the receipts are sent to
* ***void CPSSetBillingInfo(string firstName, string lastName, string address1, string address2, string city, string state, string zip, string country, string phone, string email, string ccNumber, string ccType, string ccExpMonth, string ccExpYear, bool emailReceipt, string receiptEmails)***

Updates the account’s billing information, generates exceptions if some parameters are incorrect

* ***string CPSCheckPromotionCode(string key, string sessionKey, string promotionCodeValue, int PackageId)***

Checks if promotional code is valid for particular package. If the code is invalid, throws BPWS-009 exception, otherwise returns XML string in the following format:

<xml>

<codeSettings>

<codeId>**1234**</codeId>

<codeValue>**SomeValue**</codeValue>

<promotionTitle>**SomeTitle**</promotionTitle>

<promotionDescription>**SomeDescription**</promotionDescription>

<promotionDiscountPercent>**5.00**</promotionDiscountPercent>

</codeSettings>

</xml>

**codeId** is an integer that can be passed into ***CPSDoUpgradeDowngradeAccount*** in order to apply the particular promotion during upgrade

* ***string CPSCalcNewPackage(string key, string sessionKey, string newPackageName, int newBillingCycle, int promoCodeId, string addOnsXml)***

Calculates new account price. ***addOnsXml*** must be in the following format

<AddOns>

<AddOn>

<FeeId>**12324**</FeeId>

<AddOnId>**1234**</AddOnId>

<SetUpFee>**0.00**</SetUpFee>

<MonthlyFee>**1.99**</MonthlyFee>

<Checked>**True**</Checked>

<Rewritten>**False**</Rewritten>

<Group>**Support**</Group>

</AddOn>

…<!—other add-ons -->

</AddOns>

For account from Business packages category, the following optional node may be added to specify additional resources

<xml>

<AddOns>

…..

</AddOns>

<AdditionalResources>

<DataPages>**0**</DataPages>

<DeployedDataPages>**0**</DeployedDataPages>

<Storage>**0**</Storage>

<DataTransfer>**0**</DataTransfer>

<NumberRecords>**0**</NumberRecords>

<FileStorage>**0**</FileStorage>

<NumberEmails>**0**</NumberEmails>

<NumberWSCalls>**0**</NumberWSCalls>

</AdditionalResources>

</xml>

- **newPackageName** is used to specify new package

- **newBillingCycle** is used to reset billing cycle. 1 means “Monthly”, 3 – “Quarterly”, 12 – “Yearly”

Note. Discounts are applied automatically for quarterly and annual billing cycles. There is no way to get pre-defined discounts from backend. Quarterly and annual discount amount shown in UI must be configured via web.config

- **promocodeId** is used to apply promotion codes

The method returns calculated account fees as XML string in the following format:

<CalculatedPackage>

<BaseFeeMonthly>**39.99**</BaseFeeMonthly>

<BaseFeeMonthlyAbsoluteFormatted>**39.99**</BaseFeeMonthlyAbsoluteFormatted>

<CustomResourcesMonthly>**0.00**</CustomResourcesMonthly>

<CustomResourcesMonthlyAbsoluteFormatted>

0.00

</CustomResourcesMonthlyAbsoluteFormatted>

<AddOnsMonthly>**1.99**</AddOnsMonthly>

<AddOnsMonthlyAbsoluteFormatted>1.99</AddOnsMonthlyAbsoluteFormatted>

<recurringFees>

<fee isAdditionalResource="{True|False}" addOnId="101" [addOnGroup=”SomeGroup”]>

<title>SomeAddOnTitle</title>

<description>SomeAddOnDescription</description>

<amount>9.99</amount>

<amountAbsoluteFormatted>9.99</amountAbsoluteFormatted>

</fee>

</recurringFees>

<NewDiscount>

<discount type="**{Current|Frequency|Promotional}**" categoryId="**1234**">

<title>**SomeTitle**</title>

<description>**Some Multiline description**</description>

<amount>**-1.00**</amount>

<amountAbsoluteFormatted>**-1.00**</amountAbsoluteFormatted>

</discount>

</NewDiscount>

…<!—other discounts -->

<PendingNonRecurringFees>

<fee>

<title>**SomeTitle**</title>

<description>**Some Multiline description**</description>

<amount>**0.00**</amount>

<amountAbsoluteFormatted>**0.00**</amountAbsoluteFormatted>

</fee>

… <!—other non-recurring fees -->

</PendingNonRecurringFees>

<TotalRecurringMontly>**40.98**</TotalRecurringMontly>

<TotalRecurringMontlyAbsoluteFormatted>**40.98**</TotalRecurringMontlyAbsoluteFormatted>

<TotalRecurringCycle>**40.98**</TotalRecurringCycle>

<TotalRecurringCycleAbsoluteFormatted>**40.98**</TotalRecurringCycleAbsoluteFormatted>

<TotalNonRecurring>0.00</TotalNonRecurring>

<TotalNonRecurringAbsoluteFormatted>0.00</TotalNonRecurringAbsoluteFormatted>

<TotalCredits>-0.00</TotalCredits>

<TotalCreditsAbsoluteFormatted>-0.00</TotalCreditsAbsoluteFormatted>

<TotalCycleDue>**40.98**</TotalCycleDue>

<TotalCycleDueAbsoluteFormatted>**40.98**</TotalCycleDueAbsoluteFormatted>

<IsDowngrade>**False**</IsDowngrade>

</CalculatedPackage>

* ***string CPSDoUpgradeDowngradeAccount(string key, string sessionKey, string newPackageName, int newBillingCycle, int promoCodeId, string addOnsXml)***

This method can be used to change account package to **newPackageName**, to reset billing cycle to **newBillingCycle**, to apply promotion code set via **promoCodeId**, or to add/remove add-ons. addOnsXml uses the same format as **addOnsXml** parameter of ***CPSCalcNewPackage*** method. If upgrade fails, the method generates an exception; otherwise it returns XML string in the following format:

<xml>

<warningCode>BPWS-111 or BPWS-112 or empty</warningCode>

<description>Result Description</description>

<creditCard>{TRUE|FALSE}</creditCard>

<transactionResponse>

<transactionResult>APPROVED|DECLINED|ERROR</transactionResult>

<transactionId>Transaction ID</transactionId>

<authorizationNumber>Authorization Number</authorizationNumber>

<reasonCode>Gateway Response Code</reasonCode>

<reasonDescription> Gateway Response Description</reasonDescription>

</transactionResponse>

<invoice>

...

</invoice>

</xml>

Most of the information returned is irrelevant for CPS UI.

#### Exceptions

During the call of CPSxxxx methods BridgePay service can generate exceptions with the following messages:

|  |  |
| --- | --- |
| **Message** | **Meaning** |
| BPWS-001 | Internal error, unexpected error |
| BPWS-002 | Incorrect sessionKey or account is not logged in |
| BPWS-003 | Incorrect package name |
| BPWS-006 | Incorrect billing cycle argument |
| BPWS-007 | Incorrect email argument |
| BPWS-008 | Incorrect credit card number/other CC info |
| BPWS-009 | Incorrect promo code |
| BPWS-010 | Generic “Invalid Argument” error |
| BPWS-025 | Account cannot be upgraded because new package has insufficient DataTransfer allocated, less than current accounts already used |
| BPWS-100 | Failed to adjust account resources |
| BPWS-101 | Error processing credit card |
| BPWS-111 | Billing customer does not exists or not included in billing, warning only |
| BPWS-112 | Account cannot be upgraded, because it was ignored by Billing System as a legacy account, warning only |
| BPWS-113 | Account is not under Direct Billing provider and can't be charged directly |
| BPWS-114 | Account does not use CC billing method and can't be charged |
| BPWS-115 | Account is not operable or is not marked as Auto Bill |

## UI Screens Explanation

#### Configure Package step

* This screen performs two functions: it shows information about current account and allows to select new package (“plan”) and/or add-ons
* The screen consists of Current Plan section, New Plans menu, Add-ons section and top-right-aligned Your Package section

#### Current Plan

* Current Plan shows information about current account package (“plan”). The information is obtained from ***CPSGetCurrentAccountPackage*** method call return value, from the following XML node values (see Figures 1.a and 1.b in RFP.docx to see sample values mentioned below)
  + **packageName** (“Professional ||”)
  + **baseFee** (“$149 per month”)
  + If **packageCategory**=”Trial”, we do not show a price (it’s zero) and Data Transfer (it’s unlimited)
* DataTransfer (“250 MB”) cannot be obtained from ***CPSGetCurrentAccountPackage*** method call directly. Instead it should be extracted from ***CPSGetAccountPackages*** method call (see section **New Plans** below)

#### New Plans

* This section shows all packages that are available for existing account to upgrade or downgrade. The list can be obtained from ***CPSGetAccountPackages*** method call and contains the following information
* acct\_type\_name (Small/Medium/Large/etc, as a title)
* bandwidth (“250MB”, “2,000 MB” etc. as “Data Transfer”)
* standard\_fee ($50, $300, etc.)
* ***CPSGetAccountPackages*** method always returns current account package among the results. Current account package should not be shown in New Plans section

#### Add-Ons

* This section is shown and handled differently depending on the current account package category. The package category can be obtained from **packageCategory** XML node value returned from ***CPSGetCurrentAccountPackage*** method call. There are 3 possible values of this field that make sense in the context of this document – “Trial”, “Business”, and everything else. Specific of “Trial” accounts handling is mentioned above, in Current Plan section. Only the “Business” category is treated differently in the context of Add-Ons section.
* If current account package category is not “Business” or account was started as one from “Business” category but user already has selected a new package, the Add-Ons section must consist of 3 subsections – Technology, Data Transfer and Support
* If current account package category is “Business” and user has not yet selected any new account package to upgrade to, the Add-Ons section must consist of 6 subsections - Technology, Data Transfer, DataPages, Email, Storage and Support
* Subsections are filled with the information obtained from ***CPSGetAvailableAddOns*** (all categories)plus ***CPSGetAccountPackages*** (“Business” category only) method calls. For Business category packages, the Data Transfer, DataPages, Email and Storage subsections’ content is based on ***CPSGetAvailableAddOns*** method return value and shown as follows:
  + Some static hardcoded text, one per subsection
  + Picker with corresponding “unit size”, obtained from ***CPSGetAccountPackages*** method’s **data\_transfer\_unit, data\_pages\_unit, emails\_unit, storage\_unit** result fields respectively. Data Transfer and Storage must have the word “MB” added as the end
  + Prices per unit, obtained from **data\_transfer\_fee, data\_pages\_fee, emails\_fee, storage\_fee** result fields, respectively.
* Technology and Support subsections’ content is based on ***CPSGetAvailableAddOns*** method’s return value and shown as follows (for all package categories):
  + Add-ons that do not have associated Group and listed outside any parent “Group” node must be shown in Technology subsection
  + Add-ons that listed under <Group Name=”Support”> node must be shown in Support subsection
  + Add-ons that listed under each <Group> node are mutually exclusive and must be represented in UI as radio buttons. Add-ons that are listed outside all groups must be shown in UI as checkboxes
  + Add-ons that have **CanBeChanged** attribute set to False, must be shown in UI with corresponding controls disabled
  + Add-ons that have **Checked** attribute set to true, must be shown in UI as checked by default
* The following <AddOn> child nodes are used inside subsection to represent each add-on:
  + **Title** to show add-on title (green text on Fig 1.a and 1.b, e.g. “White Label”)
  + **Description** to show add-on description (2nd line of text, e.g. “The ability to show unbranded apps and ….”)
  + **MonthlyFee** to show price with the words “per month” at the end (“$20 per month”)
  + **SetUpFee** to show one-time price (e.g. “$100”), without the words “per month”
* Data Transfer section for non-Business category packages is based on ***CPSGetAvailableAddOns*** method’s return value and shown in the same way as “Support” subsection, with the only difference that Add-on group in this case is “DataTransfer”. In other words, unlike in Business category packages, where DataTransfer is “additional resource”, for all other package categories the data transfer is an add-on.
* When Configure Package UI screen is loaded, Add-ons section must reflect the information about add-ons currently enabled in the account. To do so, use ***CPSGetAvailableAddOns*** method and pass current account package as **newPackageName** there. The method will return full add-on information for current account. Now, if user checks or unchecks some add-ons in UI, these changes in Add-on **Checked** status have to be reflected in XML returned from ***CPSGetAvailableAddOns*** and be passed (mostly unchanged) as **addOnsXml** argument into ***CPSCalcNewPackage***  method. The only additional XML change that has to be made in this scenario is that add-on group name has to be transferred from attribute (***CPSGetAvailableAddOns*** return) to node value (***CPSCalcNewPackage*** argument’s format).
* If user changes existing package to a new one, the Add-Ons section has to be updated. To do so, use ***CPSGetAvailableAddOns*** method and pass new account package name as **newPackageName** argument. The method will return new add-ons layout which must be used to refresh Add\_ons section. Since some add-ons can change their behaviour and/or price depending on which package they are assigned to, some add-ons may be returned with **Rewritten** attribute set to True. This flag highlights the fact that system has changed the price of add-on because of upgrade and this new price should be reflected in UI. UI code should not try and alter status of this add-on attribute and should pass it into ***CPSCalcNewPackage*** and/or upgrade method as is.
* UI code can always find corresponding add-ons using **addOnId** node value – it is unique across all packages

#### Recalculation and Your Package section

* When Configure Package UI screen is loaded, Your Package section shows the information obtained from ***CPSGetCurrentAccountPackage*** method results, as follows:
  + Name of the package (“Medium” + “platform plan” string literal) from **packageName** node followed by the price from **baseFee** node
  + For “Business” category only, the additional resources obtained from ./recurringFees/fee[@isAdditionalResource=’True’]/**title** and …/**amount** nodes
  + For all categories, the content of all other recurringFees/**fee** nodes
  + Discounts (if any) obtained from recurringDiscounts/discount, **title** and **amount** nodes, amounts must be shown as negative numbers
  + Credits (if any) from **creditsTotal** node, preceded by word “Credits”, shown only if non-zero
  + The number from **recurringTotal** node as “Monthly Total”
* The following changes may trigger recalculation and, eventually, make upgrade or downgrade possible
  + Changes of account plan on **Configure Package** step
  + Changes of additional resources on **Configure Package** step (for accounts from Business plan category)
  + Any changes in **Data Transfer** and **Support** sections or other selected add-ons on **Configure Package** step (for accounts of any plans and categories)
  + Changes of **Payment Frequency** on **Payment Information** step
  + Click on Apply (the promo code) button on **Payment Information** step
* Use ***CPSCalcNewPackage*** method call to obtain newly calculated invoice and show it in Your Package section.
* In order to recalculate any non-Business category package, the **addOnsXml** parameter must be prepared as described above in Add-Ons section. For “Business” category packages, in case of nonzero number of units from DataTransfer, DataPages, Email and Storage subsections, UI must calculate the actual value (NumberOfUnits\*SizeOfOneUnit) and pass the results of the calculation as respective **DataTransfer**, **DataPages**, **NumberEmails** and **Storage** child node of **AdditionalResources** node
* When account package is changed to one of the plans mentioned in New Plans section, Your Package section must show the information obtained from ***CPSCalcNewPackage*** method call. This method returns basically the same information as ***CPSGetCurrentAccountPackage*** but in slightly different format. Your Package section must be filled with the following information:
  + Name of new package from **newPackageName** call argument followed by the price from **BaseFeeMonthly** node
  + The same fee node(s) for additional resources and add-ons
  + Discount (always one) obtained from NewDiscount/discount, **title** and **amount** nodes, amounts must be shown as negative numbers (only “Current” discount can be returned on step 1, but “Promotional” or “Frequency” discounts can be returned on step 3 as well).
  + Credits (if any) from **TotalCredits** node, preceded by word “Credits”, shown only if non-zero
  + The content of **TotalRecurringMonthly** node as “Monthly Total”
  + Non-recurring fees from PendingNonRecurringFees/fee/**title** and **amount** nodes, under One-Time Fees subsection
  + The number from **TotalNonRecurring** node as “One-time total”

#### Contact Information step

* Use **CPSGetContactInfo** method to obtain the contact information of existing customer account and prefill the form.
* **Employees** dropdown is not database-driven, selected value is submitted as a string using **employees** parameter
* **Industry** and **Country** dropdowns are database-driven (use BridgePay **ListIndustries** and **ListCountries** methods, respectively). The same value (**industry\_name** and **name**) is used for both dropdown’s display text and value.
* **State/Province** dropdown is not database-driven and pre-filled with a list of the U.S. states. Full state name must be used as dropdown value and submitted to BridgePay (i.e. the correct examples are “California” and “Alaska”, but not “CA” or “AK”)
* Email must be validated to ensure it is a correct email address
* Use **CPSSetContactInfo** method to store changes (if any) the user might have made during the current session. Changes must be stored only after successful account upgrade/downgrade, and **not** at the moment when user clicks Next on Contact information screen

#### Payment Information step

* Use **CPSGetBillingInfo** method to obtain the billing information of existing customer account and prefill the form.
* See Contact Information screen above – the same notes apply to **Country** and **State/Province** dropdowns on this screen
* **Billing Email** and **Receipt Email** must be validated to ensure they are correct email addresses
* **Card Number** must be validated using Luhn algorithm
* Payment systems logos on top of the **Payment Information** section are radio buttons to select Credit Card Type **(ccType)** They should be selected automatically as user types his/her credit card number (see, for example, <http://stackoverflow.com/questions/72768/how-do-you-detect-credit-card-type-based-on-number>) but should allow changes from user.
* Use **CPSSetBillingInfo** method to store changes (if any) the customer might have made during the current session. Changes must be stored only after successful account upgrade/downgrade.
* **Payment frequency** period names (Monthly, Quarterly or Yearly) can be hard-coded, but actual discount values (5% and 10%, as shown on mockups) must be obtained from app’s web.config file. Use **newBillingCycle** argument in order to recalculate price or upgrade an account.
* Use ***CPSCheckPromotionCode*** method when user clicks Apply button. **newPackageName** parameter must be the current account package name (if it was not changed on step 1) or new account package name selected on step 1. In case of successful validation the method returns an XML with numeric **codeId**. This value must later be passed as **promoCodeId** argument of ***CPSCalcNewPackage*** and ***CPSDoUpgradeDowngradeAccount*** methods.
* Only one (the biggest one) discount can be applied. I.e. if user enters promo code with 15% discount and select annual billing cycle (10% discount), 15% discount will be applied. From the other hand, if in above situation promo code gives only 5% discount, the higher (annual 10%) discount will be applied.

#### Upgrade Account (last step)

* Happens when user clicks Finish on Payment Information screen
* Use ***CPSDoUpgradeDowngradeAccount*** method to perform the actual change of account plan, call arguments are the same as for ***CPSCalcNewPackage*** method. See Recalculation section above for details
* If ***CPSDoUpgradeDowngradeAccount*** call did not generate exception and customer made changes in Contact Information and/or Payment Information forms, the corresponding **CPSSetContactInfo** and/or **CPSSetBillingInfo** methods must be called at this moment.