
Tryton Payment Gateway Documentation

Release 3.0.1.0dev1

Openlabs

January 03, 2014

Contents

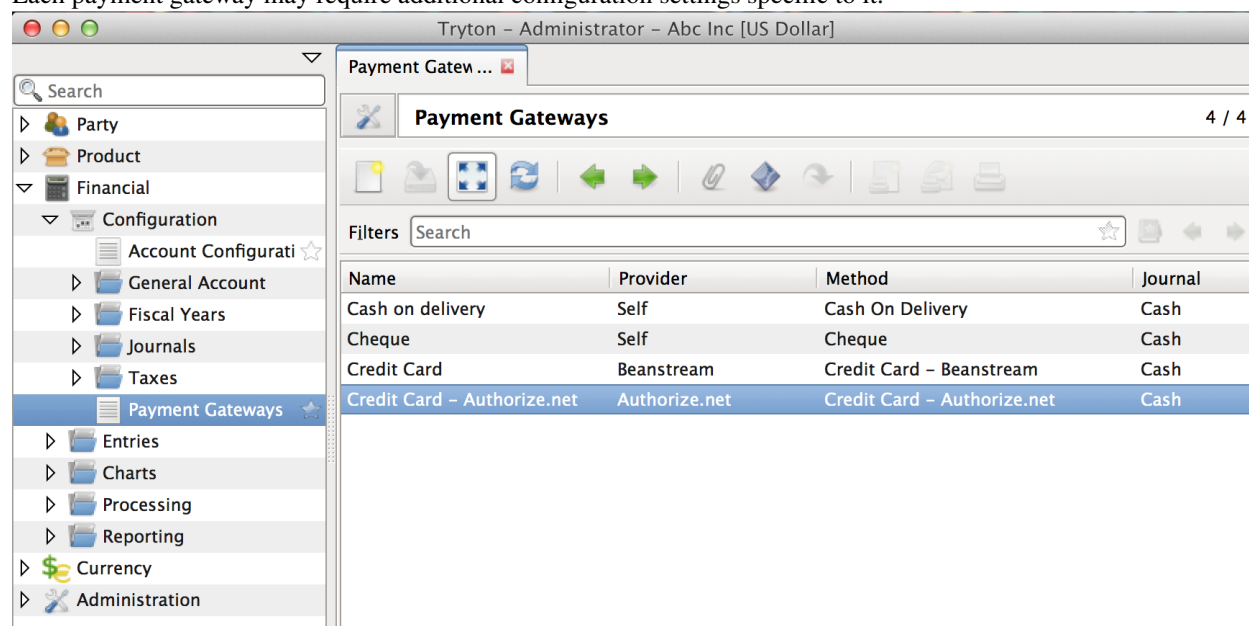
Contents:

Introduction

The payment-gateway module offers a flexible payments model which allows multiple payment gateways to co-exist in a single Tryton database. The logic for storing payment profile and transactions are decoupled from the gateway specific implementation itself making it easy to create custom payment gateways with their own processing logic and feature sets.

1.1 Payment Gateway

Payment gateway represents a specific method of payment by a specific provider (like Authorize.net, Paypal etc.). Each payment gateway may require additional configuration settings specific to it.



Tryton - Administrator - Abc Inc [US Dollar]

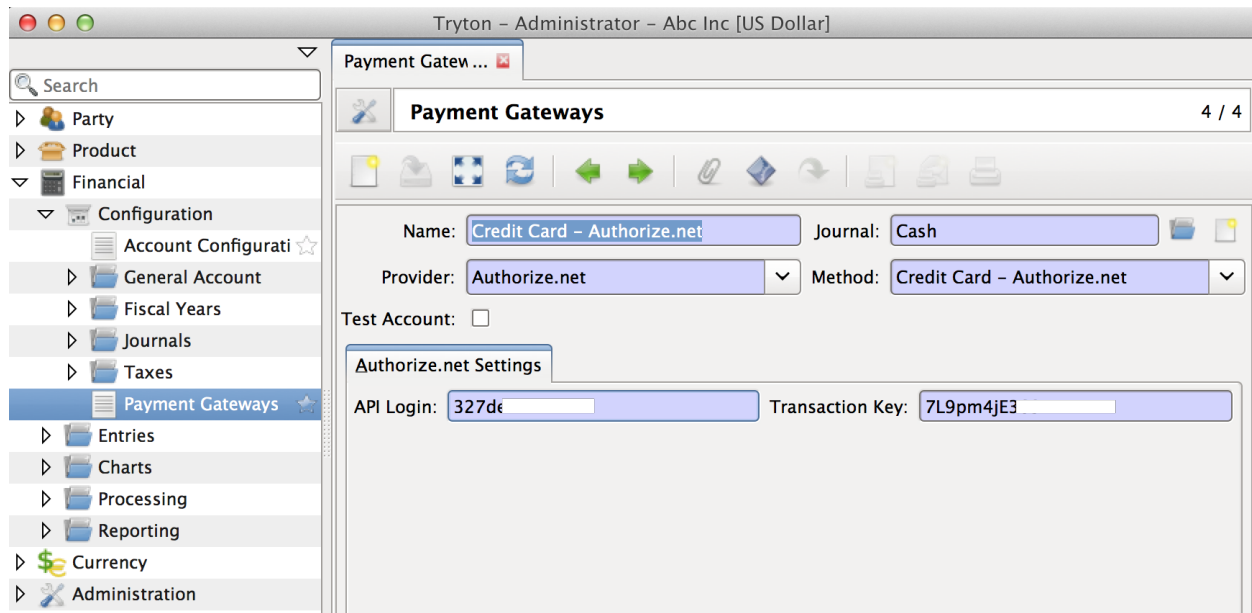
Payment Gatew ...

Payment Gateways 4 / 4

Filters Search

Name	Provider	Method	Journal
Cash on delivery	Self	Cash On Delivery	Cash
Cheque	Self	Cheque	Cash
Credit Card	Beanstream	Credit Card - Beanstream	Cash
Credit Card - Authorize.net	Authorize.net	Credit Card - Authorize.net	Cash

1.1.1 Adding payment gateway

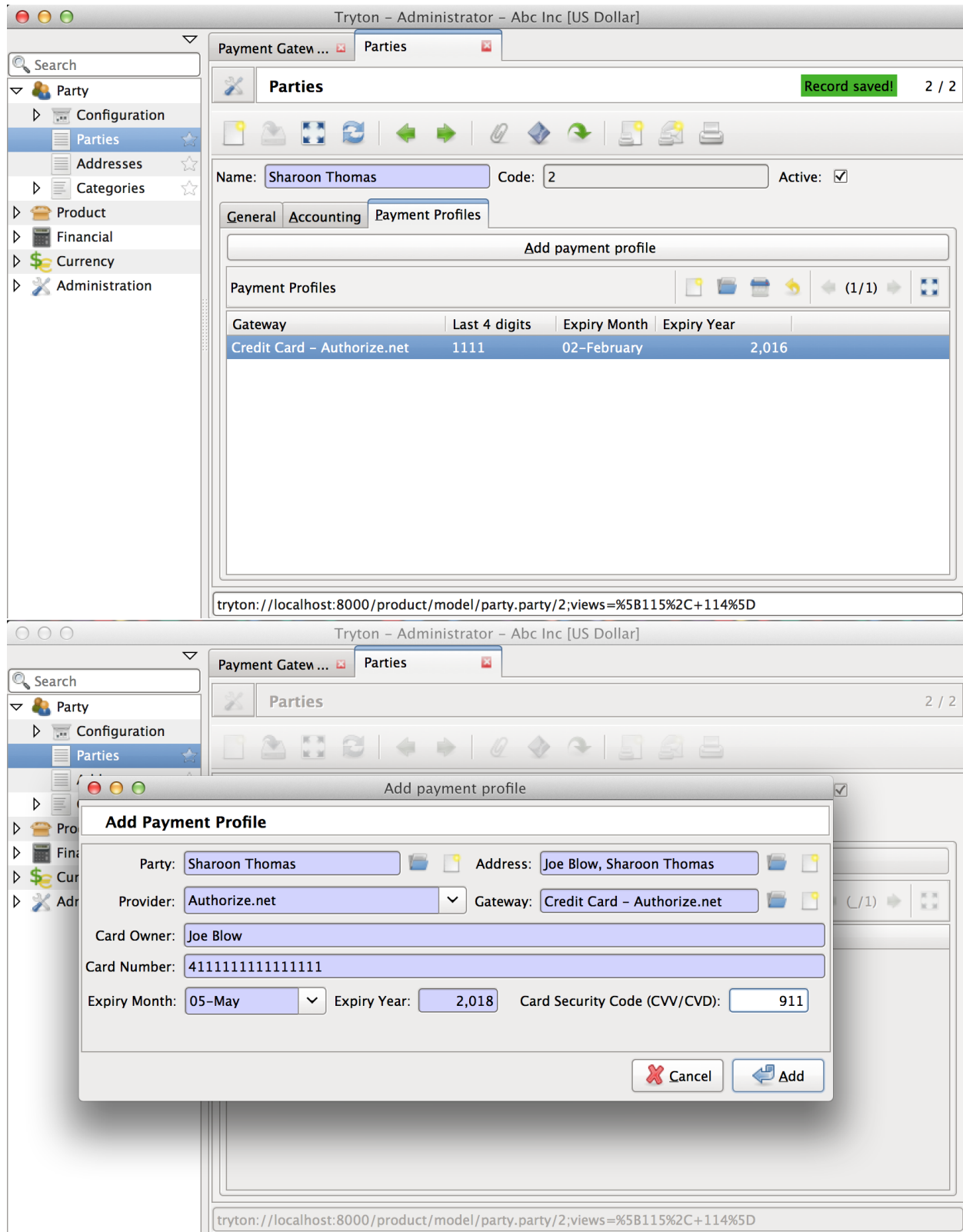


See `PaymentGateway`.

1.2 Payment Profile - Store Credit Card data

Several payment gateway service providers offer a secure way to store confidential customer credit card information on their server. Transactions can then be processed against these profiles without the need to recollect payment information from the customer, and without the need to store confidential credit card information in Tryton.

This model represents a profile thus stored with any of the third party providers. The module only stores the last 4 digits and expiration date in the database. Remaining confidential information is stored on the payment service providers server and a reference to the same is stored in the `provider_reference` field.



See PaymentProfile.

1.3 Payment Transaction

The transaction model stores and tracks payments that are made using the payment gateways.

When a transaction is created, it is assigned a unique `uuid.uuid4()`. This is used as transaction reference when transactions are sent to payment gateways. Without this identifier, some providers mistakenly report duplicate payments.

See `PaymentTransaction`.

1.3.1 States of a Transaction

State	Description
Draft	The transaction is just being filled by the user. This is the default state where every transaction begins
In Progress	Some gateways do not immediately return a success or failure of a transaction. Such transactions could be moved to the in-progress state and the status of the transaction is queried later to see if the transaction succeeded or failed.
Failed	The transaction failed. The reasons can be seen from the logs.
Authorized	The transaction has been authorized, but not settled.
Completed	The transaction has been captured, but the account moves itself, has not been created within Tryton.
Posted	The transaction is complete and the necessary account moves have also been created.
Cancelled	The transaction was cancelled.

1.3.2 Transaction using card

Tryton - Administrator - Abc Inc [US Dollar]

Payment Gateway Transactions 1 / 1

UUID: 57a0c355-cd53-428c-a04b-63ff4d2dedea

General Other Information

Use Card

Enter Card Information

Card Owner:

Card Number:

Expiry Month: Expiry Year: Card Security Code (CVV/CVD):

Draft

tryton://localhost:8000/product/model/payment_gateway.transaction/9;views=%5B231%2C+232%5D

Tryton - Administrator - Abc Inc [US Dollar]

Payment Gatew ...

Payment Gateway Transactions 1 / 1

UUID: 57a0c355-cd53-428c-a04b-63ff4d2dedea

General **Other Information**

Party: Address:

Gateway: Payment Profile:

Amount: Currency:

Draft

tryton://localhost:8000/product/model/payment_gateway.transaction/9;views=%5B231%2C+232%5D

1.3.3 Transaction using payment profile

Tryton - Administrator - Abc Inc [US Dollar]

Payment Gatew ...

Payment Gateway Transactions 1 / 1

UUID: 57a0c355-cd53-428c-a04b-63ff4d2dedea

General **Other Information**

Party: Address:

Gateway: Payment Profile:

Amount: Currency:

Draft

tryton://localhost:8000/product/model/payment_gateway.transaction/9;views=%5B231%2C+232%5D

1.3.4 Safe Posting

when the journal is created) and a fiscal period to exist on the date of the transaction. Hence, if the system was to make the account move along with the transaction capture or authorization, it could lead to inconsistencies since the capture/authorize could have already been completed on the payment gateway but the creation of account move might result in the failure of the entire transaction change.

To solve the problem, the design introduces a *completed* stage during which no account moves are created. This state makes a transition with minimal scope for error (a single state field is update), to be available. This is important since a transaction rollback due to any error could lead to Tryton having an inconsistent state of the transaction compared to the gateway.

In addition to this the transaction model offers a `safe_post()` method which tries to post the transaction, but leaves the transaction in the current state on failure. The user could later look into the completed transaction and post them manually.

1.4 Payment Transaction Log

The transaction log model stores responses from the payment service provider. When a response is received from a payment service provider, it could be passed onto `TransactionLog.serialize_and_create()`, which would then serialize the response object as **YAML** and store it. The responses can be useful in identifying the reason why a transaction may have failed.

Writing a payment gateway module

This is a developer guide for programmers wanting to write a `payment_gateway` for a payment provider. This guide assumes a beginner level of expertise in writing modules for Tryton.

The examples in the case use `Authorize.net` as an example. The completely built module can be seen on github ([payment-gateway-authorize-net](#)).

2.1 Step 0: Identify a qualified name for the provider

To keep the code simple, the `payment-gateway` module appends the name of the provider to method names and expects them to exist in the models. This requires that you use a consistent provider name which can also be a valid identifier in python.

In this example the provider name chosen is `authorize_net` for `Authorize.net`. Though it is not a requirement, the identifier is all in small case as python identifiers are case sensitive and method names by coding convention use small case.

2.2 Step 1: Setup the payment gateway fields for configuration

Every payment gateway has a different way of authentication and requirements. Hence, the only common component the base module offers you is a `test` boolean field if the gateway is working in a test mode or production.

2.2.1 Add provider name to providers selection list

As you can see in the code above, the fields' properties are based on the value of the provider field which is a `trytond.model.fields.Selection` in which the options are returned by the `get_providers()` method. So the code also needs to inject `authorize_net` as an option.:

```
@classmethod
def get_providers(cls, values=None):
    """
    Add authorize_net as a provider option.
    """
    rv = super(PaymentGatewayAuthorize, cls).get_providers()
    authorize_record = ('authorize_net', 'Authorize.net')
```

```
if authorize_record not in rv:
    rv.append(authorize_record)
return rv
```

The model also includes a method selection field in which the values are added dynamically based on the chosen provider. This is achieved using the `selection_change_with` functionality of selection fields.:

```
def get_methods(self):
    if self.provider == 'authorize_net':
        return [
            ('credit_card', 'Credit Card - Authorize.net'),
        ]
    return super(PaymentGatewayAuthorize, self).get_methods()
```

The currently recognised types and the special features attached to them are:

Method name	Description
<i>credit_card</i>	When credit card is the method chosen, the payment transaction form shows the <i>Enter Credit Card</i> button. Other methods are considered as off-line payment methods, with no special functionality attached to it.

Note: Future versions of the module may support additional methods like *Electronic Bill payments (EBP)* and *Automated Clearing House (ACH)* which works like electronic versions of cheques.

2.2.2 Add gateway specific fields to the model

Authorize.net requires a *login* and *transaction_key* to interact with it's web service API. So the two fields can be created into the `payment_gateway.gateway` module:

```
class PaymentGatewayAuthorize:
    __name__ = 'payment_gateway.gateway'

    authorize_net_login = fields.Char(
        'API Login', states={
            'required': Eval('provider') == 'authorize_net',
            'invisible': Eval('provider') != 'authorize_net',
        }, depends=['provider']
    )
    authorize_net_transaction_key = fields.Char(
        'Transaction Key', states={
            'required': Eval('provider') == 'authorize_net',
            'invisible': Eval('provider') != 'authorize_net',
        }, depends=['provider']
    )
```

Tip: The states make the field appear only when the chosen provider is Authorize.net. The fields are also required only when Authorize.net is the gateway.

2.2.3 Add the fields to the view

The fields above will not be available on the view of the gateway unless explicitly added using XML. The base module provides an empty notebook into which pages can be added which are displayed based on the value of the provider selection field.


```
<!-- XML record for the view which inherits gateway form view -->
<record model="ir.ui.view" id="gateway_view_form">
    <field name="model">payment_gateway.gateway</field>
    <field name="inherit" ref="payment_gateway.gateway_view_form"/>
    <field name="name">gateway_form</field>
</record>
```

And the view code could be something like:

```
<?xml version="1.0"?>
<data>
    <xpath expr="/form/notebook" position="inside">
        <page string="Authorize.net Settings" id="authorize_net"
            states="{ 'invisible': Eval('provider') != 'authorize_net' }">
            <label name="authorize_net_login"/>
            <field name="authorize_net_login"/>
            <label name="authorize_net_transaction_key"/>
            <field name="authorize_net_transaction_key"/>
        </page>
    </xpath>
</data>
```

Note: The empty notebook in the original view (*payment_gateway.gateway_view_form*) in the *xpath /form/notebook* offers a simple way to add payment gateway specific configuration fields on a separate notebook page which is visible only when the gateway which defines them is chosen.

2.3 Step 2: Add Methods for transactions

Payment gateway transaction usually involve the following operations. The method names used for the same are also highlighted in the table.

Op-eration	Description	Pre-fix	Example
Autho-riza-tion	Authorization hold (also card authorization, preauthorization, or preauth) is the practice within the banking industry of authorizing electronic transactions done with a debit card or credit card and holding this balance as unavailable either until the merchant clears the transaction (also called settlement), or the hold “falls off.”	au-tho-rize_	autho-rize_authorize_net
Settle	Credit card settlement is the process by which authorized transactions are submitted to card issuers for payment.	set-tle_	set-tle_authorize_net
Capture	Capture is the process of performing an authorization and settlement at once without having separate steps.	cap-ture_	cap-ture_authorize_net
Retry	When a transaction fails some gateways offer the option to retry the transaction which failed.	retry_	retry_authorize_net
Up-date	Update the transaction status.	up-date_	up-date_authorize_net
Can-cel	Cancel an authorization	can-cel_	can-cel_authorize_net

Not all of the above methods need to be implemented for a gateway to be useful. The *capture* method is a minimum requirement for a functional gateway.

Note: This example uses a third party python module called *authorize_sause* to interact with authorize.net.

2.3.1 Authorization

authorize_authorize_net ([*card_info*])

Authorize the current transaction with the card (if provided) or the *payment_profile*.

Parameters *card_info* – An instance of *CreditCardView*

Raises *UserError* If card and profile are missing.

This instance method receives the transaction to be authorized as its instance (*self*) and optionally *card_info* if a card was entered for the transaction to be processed. The *card_info* is available only when the transaction processed using a card. Alternatively, a previously stored *payment profile* could have been specified in the *payment_profile* field:

```
def authorize_authorize_net(self, card_info=None):
    """
    Authorize using authorize.net for the specific transaction.

    :param credit_card: An instance of CreditCardView
    :raises UserError: If card and profile are missing.
    """
    TransactionLog = Pool().get('payment_gateway.transaction.log')

    client = self.gateway.get_authorize_client()

    # A hack to inject the currency paramater into base_params of the
    # authorize sause transaction API since the implementation itself
    # does not offer a better way of handling currency
    client._transaction.base_params['x_currency_code'] = self.currency.code

    if card_info:
        # Card information is specified, so create a Credit Card
        cc = CreditCard(
            card_info.number,
            card_info.expiry_year,
            card_info.expiry_month,
            card_info.csc,
            card_info.owner,
        )
        credit_card = client.card(cc)
    elif self.payment_profile:
        # A stored payment profile is used to process the transaction.
        # Use the saved card instead
        credit_card = client.saved_card(
            self.payment_profile.provider_reference
        )
    else:
        self.raise_user_error('no_card_or_profile')

    try:
        # try to authorize the card for the amount in the transaction
        result = credit_card.auth(self.amount)
    except AuthorizeResponseError, exc:
        # This error is raised when Authorize.net returns an error
        # response
        self.state = 'failed'
        self.save()
```

```
    # The full response of the error is part of the exception
    # raised, store that in the logs for easy debugging.
    TransactionLog.serialize_and_create(self, exc.full_response)
else:
    # the authorization was succesful, so set the state and save
    self.state = 'authorized'
    self.provider_reference = str(result.uid)
    self.save()

    # Save the full response either way into the logs
    TransactionLog.serialize_and_create(self, result.full_response)
```

2.3.2 Settle

settle_authorize_net()

Settle the current transaction for the full amount.

This instance method receives the transaction to be authorized as its instance (*self*). On being called it attempts to settle the complete amount of the transaction with the service provider. Future versions may support the ability to have partial settlements.:

```
def settle_authorize_net(self):
    """
    Settles this transaction if it is a previous authorization.
    """
    TransactionLog = Pool().get('payment_gateway.transaction.log')

    client = self.gateway.get_authorize_client()

    # A hack to inject the currency paramater into base_params of the
    # authorize sause transaction API since the implementation itself
    # does not offer a better way of handling currency
    client._transaction.base_params['x_currency_code'] = self.currency.code

    auth_net_transaction = client.transaction(self.provider_reference)
    try:
        # Try to settle the transaction
        result = auth_net_transaction.settle()
    except AuthorizeResponseError, exc:
        # This error is raised whn Authorize.net returns an error
        # response
        self.state = 'failed'
        self.save()
        TransactionLog.serialize_and_create(self, exc.full_response)
    else:
        # Mark the transaction as completed.
        self.state = 'completed'
        self.provider_reference = str(result.uid)
        self.save()
        TransactionLog.serialize_and_create(self, result.full_response)

    # Try to post the transaction
    self.safe_post()
```

Tip: The `safe_post()` method is a helper which tries to post the transaction, but on failure, it ignores the attempt without an error. This is important as an error at this stage would mean the transaction state being changed on the

service provider while tryton may not have the right status because the error caused a rollback.

2.3.3 Capture

capture_authorize_net ([*card_info*])

Capture and complete the current transaction with the card (if provided) or the *payment_profile*.

Parameters *card_info* – An instance of *CreditCardView*

Raises *UserError* If card and profile are missing.

This instance method receives the transaction to be authorized as its instance (*self*) and optionally *card_info* if a card was entered for the transaction to be processed. The *card_info* is available only when the transaction processed using a card. Alternatively, a previously stored *payment profile* could have been specified in the *payment_profile* field:

```
def capture_authorize_net(self, card_info=None):
    """
    Capture using authorize.net for the specific transaction.

    :param card_info: An instance of CreditCardView
    """
    TransactionLog = Pool().get('payment_gateway.transaction.log')

    client = self.gateway.get_authorize_client()

    # A hack to inject the currency paramater into base_params of the
    # authorize sause transaction API since the implementation itself
    # does not offer a better way of handling currency
    client._transaction.base_params['x_currency_code'] = self.currency.code

    if card_info:
        cc = CreditCard(
            card_info.number,
            card_info.expiry_year,
            card_info.expiry_month,
            card_info.csc,
            card_info.owner,
        )
        credit_card = client.card(cc)
    elif self.payment_profile:
        # A stored payment profile is used to process the transaction.
        # Use the saved card instead
        credit_card = client.saved_card(
            self.payment_profile.provider_reference
        )
    else:
        self.raise_user_error('no_card_or_profile')

    try:
        result = credit_card.capture(self.amount)
    except AuthorizeResponseError, exc:
        self.state = 'failed'
        self.save()
        TransactionLog.serialize_and_create(self, exc.full_response)
    else:
        self.state = 'completed'
        self.provider_reference = str(result.uid)
```

```
self.save()
TransactionLog.serialize_and_create(self, result.full_response)
self.safe_post()
```

2.3.4 Cancel

`cancel_authorize_net()`

Cancel the current transaction authorization.

With `authorize.net` cancellation *VOIDs* a previous authorization that has not yet been settled:

```
def cancel_authorize_net(self):
    """
    Cancel this authorization or request
    """
    TransactionLog = Pool().get('payment_gateway.transaction.log')

    if self.state != 'authorized':
        self.raise_user_error('cancel_only_authorized')

    client = self.gateway.get_authorize_client()
    client._transaction.base_params['x_currency_code'] = self.currency.code

    auth_net_transaction = client.transaction(self.provider_reference)

    # Try to void the transaction
    result = auth_net_transaction.void()

    # Mark the state as cancelled
    self.state = 'cancel'
    self.save()

    TransactionLog.serialize_and_create(self, result.full_response)
```

2.4 Step 3: Add support for payment profiles (Optional)

If the gateway you are writing supports storing confidential credit card information for later use, the provider could be added to the supported providers for maintaining payment profiles of parties.

The addition of a payment profile is expected to add the card to the payment provider's vault and return a unique reference to it which is stored in `provider_reference` field.

2.4.1 Add provider to selection field

Extend the `party.payment_profile.add_view` model to add the provider identifier as an option in the providers selection field:

```
class AddPaymentProfileView:
    __name__ = 'party.payment_profile.add_view'

    @classmethod
    def get_providers(cls):
        """
        Return the list of providers who support credit card profiles.
```

```
"""
res = super(AddPaymentProfileView, cls).get_providers()
res.append(('authorize_net', 'Authorize.net'))
return res
```

2.4.2 Implement transition_add method

The AddPaymentProfile wizard offers a form to the user to fill up confidential information which is then sent to the server.

The API requires that a *transition_add_<provider_identifier>* method be available which should create the card on the payment provider's server and save the reference to the *provider_reference*.

A convenience method *PaymentProfile.create_profile()* creates a new profile and returns the active record of the created profile, when called with the payment provider's reference as an argument:

```
class AddPaymentProfile:
    """
    Add a payment profile
    """
    __name__ = 'party.party.payment_profile.add'

    def transition_add_authorize_net(self):
        """
        Handle the case if the profile should be added for authorize.net
        """
        card_info = self.card_info

        client = card_info.gateway.get_authorize_client()
        cc = CreditCard(
            card_info.number,
            card_info.expiry_year,
            card_info.expiry_month,
            card_info.csc,
            card_info.owner,
        )
        address = Address(
            card_info.address.street,
            card_info.address.city,
            card_info.address.zip,
            card_info.address.country.code,
        )
        saved_card = AuthorizeCreditCard(
            client,
            credit_card=cc,
            address=address,
            email=card_info.party.email
        )
        saved_card = saved_card.save()
        self.create_profile(saved_card.uid)

        return 'end'
```

API Reference

Payment Gateway Transaction

copyright

3. 2013-2014 by Openlabs Technologies & Consulting (P) Ltd.

license BSD, see LICENSE for more details

3.1 *payment_gateway.gateway*

`transaction.PaymentGateway`

alias of `payment_gateway.gateway`

3.1.1 Fields

`PaymentGateway.name`

Define a char field (unicode).

`PaymentGateway.journal`

Define many2one field (int).

`PaymentGateway.provider`

Define a selection field (str).

`PaymentGateway.method`

Define a selection field (str).

`PaymentGateway.test`

Define a boolean field (True or False).

3.1.2 Methods

classmethod `PaymentGateway.get_providers()`

Downstream modules can add to the list

3.2 *payment_gateway.transaction*

`transaction.PaymentTransaction`
alias of `payment_gateway.transaction`

3.2.1 Fields

`PaymentTransaction.uuid`
Define a char field (unicode).

`PaymentTransaction.provider_reference`
Define a char field (unicode).

`PaymentTransaction.date`
Define a date field (date).

`PaymentTransaction.company`
Define many2one field (int).

`PaymentTransaction.party`
Define many2one field (int).

`PaymentTransaction.payment_profile`
Define many2one field (int).

`PaymentTransaction.address`
Define many2one field (int).

`PaymentTransaction.amount`
Define a numeric field (decimal).

`PaymentTransaction.currency`
Define many2one field (int).

`PaymentTransaction.gateway`
Define many2one field (int).

`PaymentTransaction.provider`
Define function field (any).

`PaymentTransaction.method`
Define function field (any).

`PaymentTransaction.move`
Define many2one field (int).

`PaymentTransaction.logs`
Define one2many field (list).

`PaymentTransaction.state`
Define a selection field (str).

3.2.2 Methods

`PaymentTransaction.safe_post()`
If the initial configuration including defining a period and journal is not completed, marking as done could fail. In such cases, just mark as in-progress and let the user to manually mark as done.
Failing would otherwise rollback transaction but its not possible to rollback the payment

3.3 *payment_gateway.transaction.log*

`transaction.TransactionLog`
alias of `payment_gateway.transaction.log`

3.3.1 Methods

classmethod `TransactionLog.serialize_and_create` (*transaction*, *data*)
Serialise a given object and then save it as a log

Parameters

- **transaction** – The transaction against which the log needs to be saved
- **data** – The data object that needs to be saved

3.4 *party.payment_profile*

`transaction.PaymentProfile`
alias of `party.payment_profile`

3.4.1 Fields

`PaymentProfile.party`
Define many2one field (int).

`PaymentProfile.address`
Define many2one field (int).

`PaymentProfile.gateway`
Define many2one field (int).

`PaymentProfile.provider_reference`
Define a char field (unicode).

`PaymentProfile.last_4_digits`
Define a char field (unicode).

`PaymentProfile.expiry_month`
Define a selection field (str).

`PaymentProfile.expiry_year`
Define an integer field (int).

3.5 Wizard: *party.party.payment_profile.add*

`transaction.AddPaymentProfile`
alias of `party.party.payment_profile.add`

3.5.1 Methods

`AddPaymentProfile.create_profile(provider_reference)`

A helper function that creates a profile from the card information that was entered into the View of the wizard. This helper could be called by the method which implement the API and wants to create the profile with `provider_reference`.

Parameters `provider_reference` – Value for the `provider_reference` field.

Returns Active record of the created profile

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

t

transaction,??