ChatGPT

PayFast Integration POC in ASP.NET Core Web API

Understanding the PayFast Payment Flow

PayFast offers an **off-site payment integration** (sometimes called "redirect" integration). This means your application will direct the user to PayFast's secure payment portal for entering payment details, rather than collecting sensitive card information on your site. The typical flow is:

- **Initiate Payment:** Your ASP.NET Core API provides an endpoint (e.g. `/api/pay`) that prepares a payment request. When a user hits this endpoint (via a browser or frontend call), it will generate a **checkout form** with the required fields and redirect the user to PayFast's payment page [1] [2]. The user will see PayFast's interface to fill in payment details (credit card, EFT, etc.).

- **User Pays on PayFast:** The customer completes the payment on PayFast's site. PayFast then processes the transaction securely on their end.

- **Instant Notification (ITN) Callback:** Once the payment is processed, PayFast's server sends an **Instant Transaction Notification (ITN)** HTTP POST to your specified `notify_url` endpoint on your API **before** redirecting the user back [3]. This server-to-server callback includes the transaction result and details.

- **Return to Your Site:** After sending the ITN, PayFast redirects the customer back to the `return_url` you provided. This is typically a page on your site (or frontend) that can show a success/failure message. The redirection is not a form post, just a GET redirect, so it usually doesn't contain detailed payment info. Instead, your application should already know the outcome from the ITN by the time the user returns [3] [4].

- **Cancellation Flow:** If the user cancels the payment on PayFast, they will be redirected to the `cancel_url` you provided. In a cancellation scenario, PayFast does **not** send an ITN (since no payment was completed) – your application should handle cancellations separately (e.g. mark the transaction as canceled when the `cancel_url` is hit) [5].

This flow ensures sensitive data is handled by PayFast, and your system is notified of the result through a secure back-channel (ITN). Next, we'll set up the necessary configuration and implementation steps for this flow.

Setting Up PayFast Credentials and Environment

Before coding the integration, ensure you have the following from PayFast:

- **Merchant ID and Key:** These identify your PayFast account. For example, PayFast provides sandbox testing credentials (Merchant ID `10000100` and Key `46f0cd694581a` are the default

sandbox credentials) [6], or you can use the merchant ID/key from your sandbox account. In production, use your live ID and key from your PayFast account.

- **Passphrase:** If your PayFast account has a passphrase set (used for added security), you will need it for generating and verifying signatures. The passphrase is configured in your PayFast account settings (for **sandbox, make sure to set a passphrase in the sandbox account settings as well**). Using a passphrase is recommended, and it is **required for certain features like subscriptions**, but even for one-time payments it helps ensure security [7].

- **PayFast URLs:** Use the correct PayFast endpoint depending on environment. For sandbox (testing), use `https://sandbox.payfast.co.za/eng/process`. For live transactions, use `https://www.payfast.co.za/eng/process` [8]. Your integration should easily switch between these (e.g. a config flag for sandbox vs production) [9].

When developing locally, note that PayFast's ITN **will only send callbacks to certain ports**. According to PayFast, the notify URL must use port 80, 443, 8080, or 8081 (they do not send ITN to arbitrary ports) [10] [11]. If your ASP.NET Core app is running on `localhost:5000` or another disallowed port, PayFast's server might not reach it. For local testing, consider using a tunnel (like **ngrok**) to expose your local endpoint on a public URL with an allowed port, or host your test on a server using one of those ports.

Implementing the Payment Request Endpoint

For a quick POC, you can implement an endpoint (e.g. an ASP.NET Core Controller action) that, when hit, responds with an HTML form that immediately posts to PayFast. This "redirect with form" approach will send the user from your app to PayFast with all required parameters. Key steps:

1. **Collect or Define Transaction Info:** Determine what the user is paying for and how much. In a POC, you might hardcode an item name and amount or accept query parameters. Also generate a unique identifier for the transaction on your side (like an order ID or GUID) – this can be passed to PayFast as `m_payment_id` to track the payment on return.

2. **Prepare the Payment Data:** Create a data structure with all the fields PayFast expects. Required fields include:

3. `merchant_id` – Your PayFast Merchant ID (use sandbox ID for testing) [12]

4. `merchant_key` – Your PayFast Merchant Key [12]
5. `amount` – The payment amount (format as a decimal string with two decimals, e.g. `"123.45"`).
6. `item_name` – A description of the item or order (e.g. `"Test Product"`).
7. `return_url` – URL on your site where the user is sent after payment (success page).
8. `cancel_url` – URL on your site for if they cancel the payment.
9. `notify_url` – URL on your server to receive the ITN callback (this should be an API endpoint that we'll implement). [13]

Additionally, you can include **buyer details** to pre-fill on PayFast:

- `name_first` and `name_last` – Customer's first and last name.
- `email_address` – Customer's email.

It's also good to include `m_payment_id` – a unique ID for the payment on your side (e.g. your order or transaction ID). PayFast will send this back in the ITN data, which helps you identify which order was paid [14]. For example, you might use a GUID or a database ID.

Example: The data might look like:

```
merchant_id = "10000100"
merchant_key = "46f0cd694581a"
return_url  = "https://your-app.test/payfast-return"
cancel_url  = "https://your-app.test/payfast-cancel"
notify_url  = "https://your-app.test/api/payfast/notify"
name_first  = "John"
name_last   = "Doe"
email_address = "john.doe@example.com"
m_payment_id = "TXN12345"         (your internal transaction ID)
amount      = "100.00"
item_name   = "Test Order #12345"
```

1. **Generate the Security Signature:** PayFast requires an MD5 signature of the above data (especially if you have a passphrase). The signature ensures the data isn't tampered with in transit. The signature is computed by concatenating all the fields and their values in a specific way:

2. Sort the data by field name **alphabetically** (as per PayFast's specification for signature generation).

3. Create a string in the format `key1=value1&key2=value2&...` for all the fields (excluding the `signature` itself).
4. If a passphrase is set, append `&passphrase=YOUR_PASSPHRASE` to that string [15] (note: the key `passphrase` is lowercase in the concatenated string) and include the exact passphrase value.
5. Compute the MD5 hash of this final string. The result (32-character hex string) is the signature.

In code, you should mimic what PayFast expects. For example, a pseudo-code approach in C#:

```
var data = new SortedDictionary<string,
string>(StringComparer.OrdinalIgnoreCase) {
    {"merchant_id", MERCHANT_ID},
    {"merchant_key", MERCHANT_KEY},
    {"return_url", RETURN_URL},
    {"cancel_url", CANCEL_URL},
    {"notify_url", NOTIFY_URL},
    {"name_first", firstName},
    {"name_last", lastName},
    {"email_address", email},
    {"m_payment_id", paymentId},
    {"amount", amountString},
    {"item_name", itemName}
    // (add item_description or other optional fields if needed)
```

```
};

// Build query string
string query = string.Join("&", data
                 .Where(kv => !string.IsNullOrEmpty(kv.Value))
                 .Select(kv => $"{kv.Key}
={Uri.EscapeDataString(kv.Value)}"));

if (!string.IsNullOrEmpty(PASSPHRASE)) {
    query += "&passphrase=" + Uri.EscapeDataString(PASSPHRASE);
}
string signature = CalculateMd5(query); // your MD5 hash function
data["signature"] = signature;
```

This is essentially what the PayFast documentation suggests. In the PHP example from PayFast's docs, they do the same: concatenate fields and append passphrase before MD5 hashing [16] [15]. Ensure the MD5 is calculated on the **exact string format** PayFast expects (URL-encoded values, correct case for keys, etc.). A correctly generated signature must be included as a hidden field `signature` in the form [2] [17].

1. **Return an HTML Form (Auto-submit):** Once you have the data and signature, your endpoint can return an HTML form that posts to the PayFast URL. The form should include each data field as a hidden input. You can then use a bit of JavaScript to auto-submit the form immediately (so the user doesn't have to click a second "Pay Now" button). This way, when the user hits your endpoint, they are seamlessly redirected to PayFast.

For example, your endpoint could return something like:

```
<html>
<body onload="document.forms[0].submit();">
  <form action="https://sandbox.payfast.co.za/eng/process" method="POST">
    <input type="hidden" name="merchant_id" value="10000100" />
    <input type="hidden" name="merchant_key" value="46f0cd694581a" />
    <!-- ...other fields... -->
    <input type="hidden" name="signature" value="computed_md5_hash" />
    <noscript><input type="submit" value="Pay Now"/></noscript>
  </form>
  <p>Redirecting to PayFast...</p>
</body>
</html>
```

This is similar to the approach shown in PayFast integration examples [18] [19] – the form is constructed with all the required fields and then auto-submitted. In ASP.NET Core, you can return this HTML from a controller (for a quick POC, you might hardcode the HTML string or use a Razor page/view to render it). Once this form posts, the user lands on PayFast's payment interface.

Note: Ensure you use the sandbox URL (`sandbox.payfast.co.za`) for testing. Once you move to production, switch the form action to the live URL (`www.payfast.co.za`) [8] and use your live credentials.

At this point, the user is on PayFast's site, completes the payment, and PayFast will handle the rest of the flow (calling your notify URL and redirecting back). Next, we handle the server-side notification.

Handling the ITN Callback (Payment Notification)

You need to implement an endpoint in your ASP.NET Core API to receive PayFast's Instant Transaction Notification. This is a crucial part of the integration – it's how you confirm that the payment was successful and update your system (e.g. mark an order as paid). Key steps for the `notify_url` handler:

1. **Create a Notify Endpoint:** In your controller, set up a route for PayFast to POST to, e.g. `[HttpPost] /api/payfast/notify`. This should accept form URL-encoded data (the fields PayFast sends). In .NET you can capture these via `[FromForm]` parameters or by reading `Request.Form`. **Important:** The first thing this endpoint should do is return a simple **HTTP 200 OK** response back to PayFast. PayFast requires an acknowledgment. You should send this immediately (before lengthy processing). In practice, you can send the 200 OK by just ensuring you don't return an error; usually writing to the response or ending the method normally is fine (in PHP they do `header('HTTP/1.0 200 OK'); flush();` to explicitly flush the OK status).

2. **Verify the Signature:** PayFast will send back all the fields you originally sent (like `m_payment_id`, `amount`, etc.), along with additional fields about the transaction (e.g. `pf_payment_id` which is PayFast's transaction ID, `payment_status` or similar, and a `signature` among others). You must recompute the MD5 signature on the data PayFast sent and compare it to the `signature` they included, to ensure the data wasn't tampered with in transit [20]. The process is similar to how you generated the signature initially:

3. Take all the POST variables PayFast sent, except the `signature` field.

4. Concatenate them as `key=value&...` sorted in the same order (usually alphabetical by keys). Use the *exact values* PayFast sent (after URL-decoding them, and be mindful of character encoding).
5. Append your passphrase (`&passphrase=...`) if you have one (same one used before).
6. MD5 hash the string and compare to PayFast's provided signature.

If the signatures do not match, you **must not treat the payment as successful** (it could be an attempt to spoof a callback). In a POC environment, log this scenario for debugging if it occurs. (Common causes of signature mismatch are using wrong passphrase or sorting/encoding incorrectly [21].)

1. **Verify Source (Security):** It's good practice to ensure the IP address or domain of the incoming request is actually PayFast, to avoid malicious posts. PayFast recommends checking that the callback's origin IP matches their servers. One approach is to resolve PayFast's domains (`www.payfast.co.za`, `sandbox.payfast.co.za`, and a couple of others they use for ITN) to get the list of valid IP addresses, then verify the request's IP is in that list [22] [23]. Another simpler check is to check the `HTTP_REFERER` or the Hostname of the incoming request against PayFast's domains [24]. In production, implement these checks to improve security (in a quick local POC, you might skip or just log the sender IP).

2. **Verify Payment Data:** As an additional sanity check, confirm that the details of the payment match what you expected: for example, the amount PayFast says was paid (`amount_gross` in ITN data) matches the amount you intended to charge [25] [26], and the `merchant_id` in the

POST matches your ID, etc. This ensures no one manipulated the amount on the client side. PayFast suggests ensuring the difference is no more than a few cents (to account for rounding) [25].

3. **Server Confirmation with PayFast (optional but recommended):** PayFast provides an endpoint to double-check the ITN data. You can take the *raw POST string* you received and send it back to PayFast at `https://www.payfast.co.za/eng/query/validate` (or sandbox equivalent) as an HTTP POST [27] [28]. If the response is `"VALID"`, it confirms that PayFast recognizes this transaction and the data is legitimate [29]. This step is analogous to PayPal's IPN validation. It's an extra layer to ensure the notification wasn't intercepted and replayed by someone else. In .NET, you could use `HttpClient` to POST the data back to PayFast's validate URL and check the response. This step should be done after sending the 200 OK to PayFast (or asynchronously) to not delay the acknowledgment.

4. **Update Your Database/State:** Once all the above checks pass, consider the payment **successful**. You can then mark the order as paid or perform whatever post-payment actions are needed in your POC (e.g. log a message, update an in-memory list, etc.). In a real app, you'd update your database record for that `m_payment_id` to indicate payment received. For example, the reference PHP integration sets status to "completed" if all checks pass [30]. If any check fails, you should flag the payment as failed or suspicious [31] – do not give access to goods/services in that case.

5. **Return No Content:** After processing, your notify endpoint doesn't need to return any content to PayFast (they only care that a 200 status was received). You might simply end the method or return an empty OK. PayFast will typically stop retrying the notification once they get a 200 OK from your server.

By following these steps, your API will reliably know whether the payment was successful before the user even gets back to your site. The Stack Overflow discussion confirms *"PayFast will send the ITN to your notify_url before the user gets redirected to your return_url."* [3]. This means your database can be updated by the time the customer is redirected back.

Handling Return and Cancel Redirects

Finally, you should handle what happens when the user is redirected to your `return_url` or `cancel_url` after PayFast:

- **Return URL (Success Page):** This is typically a page in your front-end or a simple webpage that says "Thank you for your payment" (or "Payment Failed" if something went wrong). Since the return redirect itself doesn't provide detailed info (to avoid exposing data via GET), your page should use the information you have on the server. A common technique is to include the `m_payment_id` or some reference in the return URL (or store it in the user's session before redirect). Then, when the user lands on the return page, you query your backend (or database) to get the status of that transaction. For example, in the PHP example, they stored the payment ID in session and looked up the transaction status in the database on the return page [32] [33]. In ASP.NET Core, you could have the return page call an API to fetch the status or embed logic to check a record. If the status is "completed" (success), show a success message (and maybe order details); if it's "failed", show an error/retry message. Since your notify handler already updated the status, the return page just displays the result to the user.

- **Cancel URL:** If the user cancels out of the PayFast payment flow, they will be sent to your `cancel_url`. PayFast does not notify your server for canceled transactions [5], so your application should treat hitting the cancel page as an indication to mark the transaction as canceled/aborted. In a POC, you might simply show “Payment was canceled” and perhaps invalidate that pending transaction ID in your system. If you were storing a temporary order, you could set its status to "canceled" here. (In the PHP example, they updated the status to "canceled" in the cancel page code itself [34].)

For a **POC running purely locally**, handling return and cancel could be as simple as printing out the status. You might not have a full front-end, so even a basic HTML page or a console log is fine. The main goal is to see that the flow completes and you get the success notification.

Testing the Integration Locally

Testing a payment gateway integration locally has some challenges, but here are some tips:

- Use the **PayFast Sandbox**: Log in to the PayFast Sandbox with your account to get sandbox credentials or use the defaults. Ensure you’ve set up your sandbox merchant settings (e.g. add a passphrase if you plan to use one). The sandbox will actually process dummy payments (no real money). You can use the Merchant ID `10000100` and Key `46f0cd694581a` with the passphrase you set (or the default one if provided) for quick tests [6].

- **Run on an Allowed Port**: As mentioned, PayFast’s ITN will only reach certain ports. If your dev server is on port 5000/5001 (Kestrel default for https in .NET Core), PayFast might not send the ITN. You have a few options:

- Use a tunneling service (like **ngrok**) to forward a public URL (on port 443 or 80) to your local dev server. This way PayFast can reach your notify URL. For example, `https://yourapp.ngrok-free.app/notify` could map to `http://localhost:5000/api/payfast/notify`.
- Alternatively, host your application in IIS/IIS Express on port 443 locally with a self-signed certificate, so that your notify URL is something like `https://localhost/notify` (port 443). Ensure to configure PayFast to use `https` and port 443 in the URL.

- As a simpler workaround for a POC, you can simulate the ITN: after payment, PayFast will redirect you to the return page. You can manually copy the transaction ID and check the PayFast dashboard for payment status. You could also create a test endpoint to mimic PayFast by posting the expected fields to your notify URL (using a tool like Postman) using the data from PayFast’s return (not as secure, but for POC validation).

- **Check Sandbox Responses**: The PayFast sandbox will show you the payment interface and allow you to simulate a payment (for example, you can choose a payment method and use test card numbers or login to a test bank account for EFT). After completing the payment, observe:

- Did your application log or record the ITN callback? (Set breakpoints or logging in your notify handler to verify it ran.)
- Was the signature verification successful? If not, double-check your passphrase and signature code.

- Did the user get redirected to your return URL? You should see the return page with the result.

- **Debugging**: If something doesn't work, PayFast's error messages can be cryptic (e.g. "Invalid merchant details" or "signature mismatch"). Common issues include incorrect merchant ID/Key (make sure to use sandbox values in sandbox mode) [35], forgetting to include the passphrase in the signature string [36], or using a notify URL that PayFast couldn't reach. The PayFast documentation and support forums have FAQs for these errors [21]. For instance, a "signature mismatch" means the MD5 you generated didn't match what PayFast expected – ensure the sorting and concatenation rules are exactly followed [21]. If the ITN isn't coming through, ensure your URL is accessible and on the right port [10] [11].

Once you have the process working locally with the sandbox, switching to production is straightforward: update the URLs to PayFast's live URL and plug in your live merchant ID/key/passphrase. The flow and code remain the same, just make sure *not* to use sandbox credentials in production or vice-versa (PayFast will reject mismatched IDs in the wrong environment) [37].

Conclusion: For a "quick and dirty" PayFast integration in ASP.NET Core, the simplest route is to have your API generate an HTML form that redirects the user to PayFast, and implement a receiver endpoint for the ITN callback to handle the result. The above steps outline the best-practice flow: redirect user to PayFast with required fields, verify the payment via ITN on your backend, then update your system and inform the user. By following PayFast's guidelines for signature generation and ITN validation, you'll ensure the integration is secure and reliable [20] [30]. Good luck with your POC, and happy coding!

Sources:

- PayFast official documentation – *Custom integration guide and ITN details* [1] [20]
- PayFast PHP integration example (Artisans Web) – *Illustrates form fields, signature generation, and ITN validation in code* [38] [30]
- Stack Overflow – *PayFast ITN sequence and notify/return behavior* [3] [10]
- PayFast Support FAQ – *Common integration pitfalls (ports and signatures)* [10] [21]

---

[1] [2] [5] [9] [14] [15] [16] [17] [18] [19] [20] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [38] PayFast Payment Gateway Integration in PHP - Artisans Web
https://artisansweb.net/payfast-payment-gateway-integration/

[3] [4] [10] [11] [12] [13] payment gateway - payfast not redirecting to notify_url - Stack Overflow
https://stackoverflow.com/questions/51889825/payfast-not-redirecting-to-notify-url

[6] [7] PayFast Developer Documentation>
https://developers.payfast.co.za/

[8] 404 Bad request: Unable to process transaction using sandbox testing payment integration in payfast - PHP - SitePoint Forums | Web Development & Design Community
https://www.sitepoint.com/community/t/404-bad-request-unable-to-process-transaction-using-sandbox-testing-payment-integration-in-payfast/418424

[21] Why do I get "Merchant Authorization Failed"? - Payfast by Network
https://support.payfast.help/portal/en/kb/articles/why-do-i-get-merchant-authorization-failed-20-9-2022

[35] What causes the error "The supplied variables are not according to ...
https://support.payfast.help/portal/en/kb/articles/what-causes-the-error-the-supplied-variables-are-not-according-to-specification-20-9-2022

36 What causes the ITN security check errors? - Payfast by Network
https://support.payfast.help/portal/en/kb/articles/what-causes-the-itn-security-check-errors-20-9-2022

37 Why am I getting a merchant_id / merchant_key error?
https://support.payfast.help/portal/en/kb/articles/why-am-i-getting-a-merchant-id-merchant-key-error-20-9-2022