

Hi,

This is a very important technical distinction to share with the community. In Ultimate++, while the built-in `HttpRequest::Part` exists, it often fails with binary files (like PDFs) because it treats the payload as a standard `String` within a concatenation flow, which can lead to memory overhead or encoding issues depending on how the data is passed.

Here is a detailed explanation in English that you can use for the community post:

Technical Note: Why `HttpRequest::Part` May Fail with Binary PDFs

When using the native `HttpRequest::Part` implementation in Ultimate++, developers often encounter issues when uploading binary files such as PDFs.

The Original Implementation

The built-in code looks like this:

```
HttpRequest& HttpRequest::Part(const char *id, const String& data,
                             const char *content_type, const
char *filename)
{
    if(IsNull(multipart)) {
        POST();
        multipart = AsString(Uuid::Create());
        ContentType("multipart/form-data; boundary=" + multipart);
    }
    postdata << "--" << multipart << "\r\n"
        << "Content-Disposition: form-data; name=\"" << id << "\"";
    if(filename && *filename)
        postdata << "; filename=\"" << filename << "\"";
    postdata << "\r\n";
    if(content_type && *content_type)
        postdata << "Content-Type: " << content_type << "\r\n";
    postdata << "\r\n" << data << "\r\n"; // <--- Potential failure point
    return *this;
}
```

The Limitations

String Buffer Management: The native `Part` method appends the binary data directly to an internal `postdata` string. When dealing with large PDF files, this double-buffering (storing the file in a `String`, then appending it to another large `String`) can cause significant memory fragmentation or hit limits in the internal `Stream` used by `HttpRequest`.

Binary Integrity: While U++ `String` is binary-safe, the way `postdata << data` operates in some

environments or older versions can sometimes interfere with null terminators or specific byte sequences if the underlying stream expects a different handling for large BLOBs.

Lack of Stream Support: Native Part requires the entire file to be loaded into a String variable first. This is inefficient for server-side applications where memory economy is vital.

The Solution: Custom Multipart Class

To solve this, we implemented a custom class SimpleMultipartRequest. The key improvements that made it work where the native version failed are:

Explicit Control over the Request Body: By building the StringBuffer manually and then using PostData(String(body)), we ensure the boundary and content-type are perfectly aligned before the Execute() engine takes over.

Decoupled Logic: Using Upp::Array to store field and file metadata avoids the is_upp_guest (Moveable) static assertions that often plague Upp::Vector with custom structs.

Reliable Binary Handling: Manually appending the file data to a StringBuffer and then setting the ContentType explicitly to application/octet-stream ensures the Skylark server receives the binary data as a raw stream, preventing the "Null Content" or "Corrupted String" errors at the destination.

Conclusion for Skylark Users

If you are building a Skylark-based server and need to receive files into a database (MariaDB/MySQL), do not rely solely on the default HttpRequest::Part for binary uploads. Creating a specialized wrapper ensures that the multipart/form-data protocol is followed strictly, allowing the http["field"] and http["field.filename"] variables to be populated correctly on the server side.

using namespace Upp;

```
// 1. Structs fora da classe para evitar problemas de escopo no compilador
```

```
struct MultipartField {  
    String id;  
    String value;  
};
```

```
struct MultipartFile {  
    String id;  
    String filepath;  
    String content_type;  
    String filename;  
};
```

```
// 2. Classe herdando de HttpRequest
```

```
class SimpleMultipartRequest : public HttpRequest {  
public:  
    SimpleMultipartRequest(const String& url = "") : HttpRequest(url) {  
        boundary = AsString(Uuid::Create());  
    }  
};
```

```
SimpleMultipartRequest& AddField(const String& id, const String& value) {
    MultipartField& f = fields.Add();
    f.id = id;
    f.value = value;
    return *this;
}
```

```
SimpleMultipartRequest& AddFile(const String& id, const String& filepath,
                                const String& content_type =
"application/octet-stream",
                                const String& filename = "") {
    MultipartFile& f = files.Add();
    f.id = id;
    f.filepath = filepath;
    f.content_type = content_type;
    f.filename = filename.IsEmpty() ? GetFileName(filepath) : filename;
    return *this;
}
```

```
String ExecuteMultipart();
```

```
private:
```

```
// Usando Array em vez de Vector para evitar o erro de MOVEABLE
Array<MultipartField> fields;
Array<MultipartFile> files;
String boundary;
};
```

```
// 3. Implementação do método de execução
String SimpleMultipartRequest::ExecuteMultipart()
{
    StringBuffer body;
```

```
// Adiciona campos de texto
for(int i = 0; i < fields.GetCount(); i++) {
    const MultipartField& f = fields[i];
    body << "--" << boundary << "\r\n";
    body << "Content-Disposition: form-data; name=\"" << f.id << "\"\r\n\r\n";
    body << f.value << "\r\n";
}
```

```
// Adiciona arquivos
for(int i = 0; i < files.GetCount(); i++) {
    const MultipartFile& f = files[i];
    String data = LoadFile(f.filepath);

    if(!data.IsVoid()) {
```

```

body << "--" << boundary << "\r\n";
body << "Content-Disposition: form-data; name=\"" << f.id
    << "\"; filename=\"" << f.filename << "\"\r\n";
body << "Content-Type: " << f.content_type << "\r\n\r\n";
body << data << "\r\n";
}
}

// Fecha o corpo do multipart
body << "--" << boundary << "--\r\n";

// Configura o HttpRequest
this->POST();
this->ContentType("multipart/form-data; boundary=" + boundary);
this->PostData(String(body));

return this->Execute();
}
void EnviarPdfParaServidor()
{
    SimpleMultipartRequest r;

    r.Host("localhost").Port(8001).Path("/webAppTest/uploaddemo");

    r.AddField("id_usuario", "12345");
    r.AddField("data_pagamento", "2026-05-14");

    r.AddFile("arquivo_pdf", "C:\\Users\\myPath\\Downloads\\boleto.pdf",
        "application/pdf", "boleto.pdf");

    String resp = r.ExecuteMultipart();

    Cout() << "Resposta do Servidor: " + resp << EOL;
}

/** in server **
SKYLARK(x, "uploaddemo:POST")
{
    String pdf = http["arquivo_pdf"].ToString();

    // 1. Captura os metadados textuais
    String idUsuario = http["id_usuario"].ToString();

```

```

LOG("ID DO USUARIO: " + idUsuario);

String dataPagamento = http["data_pagamento"];

// 2. Captura o binário puro e o nome original do arquivo
Value dadosBinarios = http["arquivo_pdf"];
String nomeArquivo = http["arquivo_pdf.filename"];
LOG("arquivo_pdf.filename: " + nomeArquivo);

MySqlSession session;
if(session.Connect("root", "10", "test")) {

    if(!dadosBinarios.IsNull()) {
        try {
            String bufferPdf = dadosBinarios.ToString(); // Binário pronto para o banco

            Sql sql(session);
            bool success = sql.Execute("INSERT INTO test (ID, TEXT, BF) "
                                     "VALUES (?, ?, ?)",
                                     idUsuario, nomeArquivo, bufferPdf);
            http.ContentType("text/plain");
            http << "OK: Arquivo recebido e gravado com sucesso!";
        }
        catch(SqlExc& ex) {
            Cerr() << "ERROR: " << ex << "\n";
            http.ContentType("text/plain");
            http << "erro de atualização no bd: " + ex;
        }
    }
    else {

        http.ContentType("text/plain");
        http << "conteudo do arquivo nulo!";
    }
    else {
        http.ContentType("text/plain");
        http << "não conectou com o mariaDB!";
    }
}
}

```

This worked in simple tests.

Thanks!