Inter-App Data Communication

How does data communication in Frappe work?

In Frappe, all apps within a site share the same database. This means that DocTypes and data from one app can be accessed by another app if the necessary permissions are granted. The data communication between apps is seamless because they all operate within the same database context.

Here's how you can test the data communication between apps:

1. Using Code:

a. Accessing DocTypes from another app:

Suppose you have a DocType called `Sample` in the `logandemail` app. You can access this DocType from the `erpnext` app like this:

```
from frappe.model.document import get_document

doc = get_document('Sample', 'Sample Name')
print(doc.field_name)
```

b. Querying data from another app:

You can use the `frappe.db.sql()` method to run SQL queries. This allows you to fetch data from any table (DocType) regardless of which app it belongs to.

```
result = frappe.db.sql("""SELECT field_name FROM `tabSample` WHERE some_condition""")
print(result)
```

2. Using Desk UI:

a. Custom Report:

You can create a custom report in the `erpnext` app that fetches data from a DocType in the `logandemail` app:

1. Go to the "Report List" view.
2. Click on "New" to create a new report.
3. Select the DocType from the `logandemail` app.
4. Define your filters and columns.
5. Save and run the report.

b. Custom Script:

You can write a custom script for a DocType in the `erpnext` app that fetches data from a DocType in the `logandemail` app:

1. Go to "Custom Script" under the "Developer" module.
2. Create a new Custom Script.
3. Select the target DocType from the `erpnext` app.
4. In the script section, write a script to fetch data from the `logandemail` app using `frappe.call()`.

Example:

```
frappe.call({
    method: "frappe.client.get_list",
    args: {
        doctype: "Sample", // DocType from logandemail app
        filters: {
            field_name: ["=", "some_value"]
        }
```

```
    },
    callback: function(response) {
        console.log(response.message);
    }
});
```

c. Linking DocTypes:

You can also create a Link field in a DocType from the `erpnext` app that links to a DocType in the `logandemail` app. This will allow you to select records from the `logandemail` app's DocType while creating/editing records in the `erpnext` app's DocType.

In conclusion, apps in Frappe can freely access data from other apps within the same site, provided the necessary permissions are in place. This design promotes modularity and extensibility, allowing developers to build upon existing apps and create integrations with ease.

▼ Does it need any special mechanisms like API?

So, each app can freely access data from other apps, without having to use any kind of mechanism like using an API?

Yes, that's correct. Within the context of a single Frappe site, all apps share the same database. This means that any app can directly access the data (DocTypes, records, etc.) of another app without needing to go through an API or any other intermediary mechanism. The data communication is direct and seamless.

However, there are a few things to keep in mind:

1. **Permissions**: Even though apps can access data from each other, the actual access is governed by the permissions set up in the system. If a user or role does not have permission to read/write a particular DocType, they won't be able to access it, regardless of which app is trying to access the data.
2. **Best Practices**: While direct data access is possible and sometimes convenient, it's essential to be cautious. Directly accessing and modifying data from another app can lead to unintended side effects, especially if the other app's internal logic or data structures change. For maintainability and to reduce tight coupling, it's often a good idea to expose specific functionalities via API-like methods or hooks, even if they're just Python functions, rather than directly querying or modifying another app's data.
3. **Updates & Dependencies**: If one app relies heavily on the data structures or logic of another app, updates to one app might break the other. It's crucial to manage dependencies and be aware of how apps interact with each other.

In summary, while direct data access is possible and provides a lot of flexibility, it's essential to use this capability judiciously and follow best practices to ensure the long-term maintainability and stability of the system.

▼ Code Implementation Examples

Provide real implementation examples using encapsulation

In the context of Frappe and inter-app communication, encapsulation can be crucial for maintaining a clean and maintainable codebase. Here are some circumstances and code examples specific to Frappe:

1. **Data Validation and Transformation**:
   - Circumstance: Before fetching data from another app, you might want to ensure the data meets certain criteria or transform it into a specific format.
   - Example:

```
# In App B (e.g., erpnext)
def get_valid_invoices():
```

```
    invoices = frappe.get_all('Invoice', filters={'status': 'Paid'})
    valid_invoices = [invoice for invoice in invoices if some_validation_function(invoice)]
    return valid_invoices
```

2. **Hide Complex Queries**:
   - Circumstance: The process to fetch data from another app involves complex ORM queries.
   - Example:

```
# In App B (e.g., erpnext)
def get_customer_invoices(customer_name):
    return frappe.get_all('Invoice', filters={'customer_name': customer_name, 'status': 'Unpaid
'})
```

3. **Controlled Access to Sensitive Data**:
   - Circumstance: You want to provide access to certain data from another app but hide sensitive information.
   - Example:

```
# In App B (e.g., erpnext)
def get_employee_basic_info(employee_id):
    employee = frappe.get_doc('Employee', employee_id)
    # Return only non-sensitive data
    return {
        'name': employee.name,
        'designation': employee.designation
    }
```

4. **Data Aggregation**:
   - Circumstance: You want to fetch aggregated data from another app, which involves multiple steps or queries.
   - Example:

```
# In App B (e.g., erpnext)
def get_monthly_sales_summary(month, year):
    sales_invoices = frappe.get_all('Sales Invoice', filters={'month': month, 'year': year})
    total_sales = sum(invoice.total for invoice in sales_invoices)
    return total_sales
```

5. **Encapsulating Behavior**:
   - Circumstance: You want to execute certain behaviors in another app based on data or triggers in your app.
   - Example:

```
# In App A (e.g., logandemail)
def send_invoice_reminder(invoice_id):
    invoice = frappe.get_doc('Invoice', invoice_id)
    if invoice.due_date < frappe.utils.nowdate():
        # Use a function from App B to send a reminder
        erpnext.send_reminder_to_customer(invoice.customer_email)
```

In these examples, the idea is to encapsulate the logic and data access within the app that owns the data (e.g., App B). This way, if there are any changes in the data structure or logic of App B, you only need to update the functions within App B, and App A remains unaffected. This encapsulation ensures that the apps remain modular and maintainable.

▼ What about whitelisted methods?

Is there no need for special mechanisms like using whitelisted methods to implement all of these examples?

In Frappe, the term "whitelisted" is used to denote methods that can be called from the client side (like from a web browser). If you're interacting between server-side Python code of two apps within the same Frappe instance, you don't necessarily need to use whitelisted methods. However, if you're trying to access a method via an AJAX call from the frontend (like from a custom script in Frappe's Desk UI), then you'd need to whitelist the method.

For the examples I provided, if they're all server-side interactions, you don't need to whitelist anything. But if you want to call any of these methods directly from the frontend, then you'd need to whitelist the specific method.

Here's how you can whitelist a method in Frappe:

```
@frappe.whitelist()
def your_method_name(args):
    # your code here
```

By adding `@frappe.whitelist()` above your method, you're allowing it to be called from the client side.

When to use whitelisted methods:

1. **AJAX Calls from Frontend**: If you're making AJAX requests from a custom script or a web page in Frappe, you'll need the method to be whitelisted.
2. **Custom API Endpoints**: If you're creating custom API endpoints in Frappe that external systems might call, you'd whitelist those methods.
3. **Custom Button Actions**: If you've added custom buttons in the Desk UI and want them to trigger server-side methods when clicked, those methods should be whitelisted.

However, remember that whitelisting a method exposes it to the client side, so ensure that you validate any data that's passed to it and handle potential security concerns.

What I Tested

This is supposed to be just a simple implementation to see if data from other app can be directly accessed and used.

For this example, let's assume we want to fetch a list of all customers from `erpnext` and log their names in `pentaho_processes`.

Steps:

1. **Fetch Data from ERPNext**:

   In your `pentaho_processes` app, we can use Frappe's ORM to fetch data from the `Customer` DocType in `erpnext`.

   ```
   # pentaho_processes/pentaho_processes/logger/interapp_logger.py

   import frappe

   def fetch_and_log_customers():
       # Fetch all customer names from ERPNext
       customer_list = frappe.get_all('Customer', fields=['name'])

       # Log the names in the Inter-app Connection DocType
       for customer in customer_list:
           doc = frappe.new_doc('Inter-app Connection')
           doc.erpnext_customer_name = customer.name
           doc.insert()
   ```

2. **Create a Button in Desk UI**:

   To trigger this function from the Desk UI, we can add a custom button.

   - Create a new DocType in `pentaho_processes` or use an existing one.
   - In the `Custom Script` section for that DocType, add a script to create a button and call the method:

```
// pentaho_processes/public/js/interapp_log.js

frappe.ui.form.on('Your DocType Name', {
    refresh: function(frm) {
        frm.add_custom_button('Fetch and Log Customers', function() {
            // Call the server-side function
            frappe.call({
                method: 'logandemail.logandemail_module_name.some_file.fetch_and_log_customers',
                callback: function(r) {
                    if(!r.exc) {
                        frappe.msgprint("Customers fetched and logged successfully!");
                    }
                }
            });
        });
    }
});
```

3. **Whitelist the Method**:

   Ensure that the server-side method is accessible by whitelisting it. In the `pentaho_processes` app, in the `.py`, above the `fetch_and_log_customers` function, add:

```
@frappe.whitelist()
```

   This decorator ensures that the method can be called from the frontend.

4. **Apply it on hooks.py:**

```
# include js in page
# page_js = {"page" : "public/js/file.js"}

# include js in doctype views
# doctype_js = {"doctype" : "public/js/doctype.js"}
doctype_js = {"Inter-app Connection" : "public/js/interapp_log.js"}
# doctype_list_js = {"doctype" : "public/js/doctype_list.js"}
# doctype_tree_js = {"doctype" : "public/js/doctype_tree.js"}
# doctype_calendar_js = {"doctype" : "public/js/doctype_calendar.js"}
```

5. **Test**:
   - Navigate to the DocType in the `pentaho_processes` app where you added the button.
   - Click on the "Fetch and Log Customers" button.
   - You should see a message indicating successful execution, and the customer names should be printed in the server logs.

This is a basic example to demonstrate the concept. In a real-world scenario, you'd probably want to do more than just print the names, but this should give you a starting point to understand how data can be accessed across apps in Frappe.