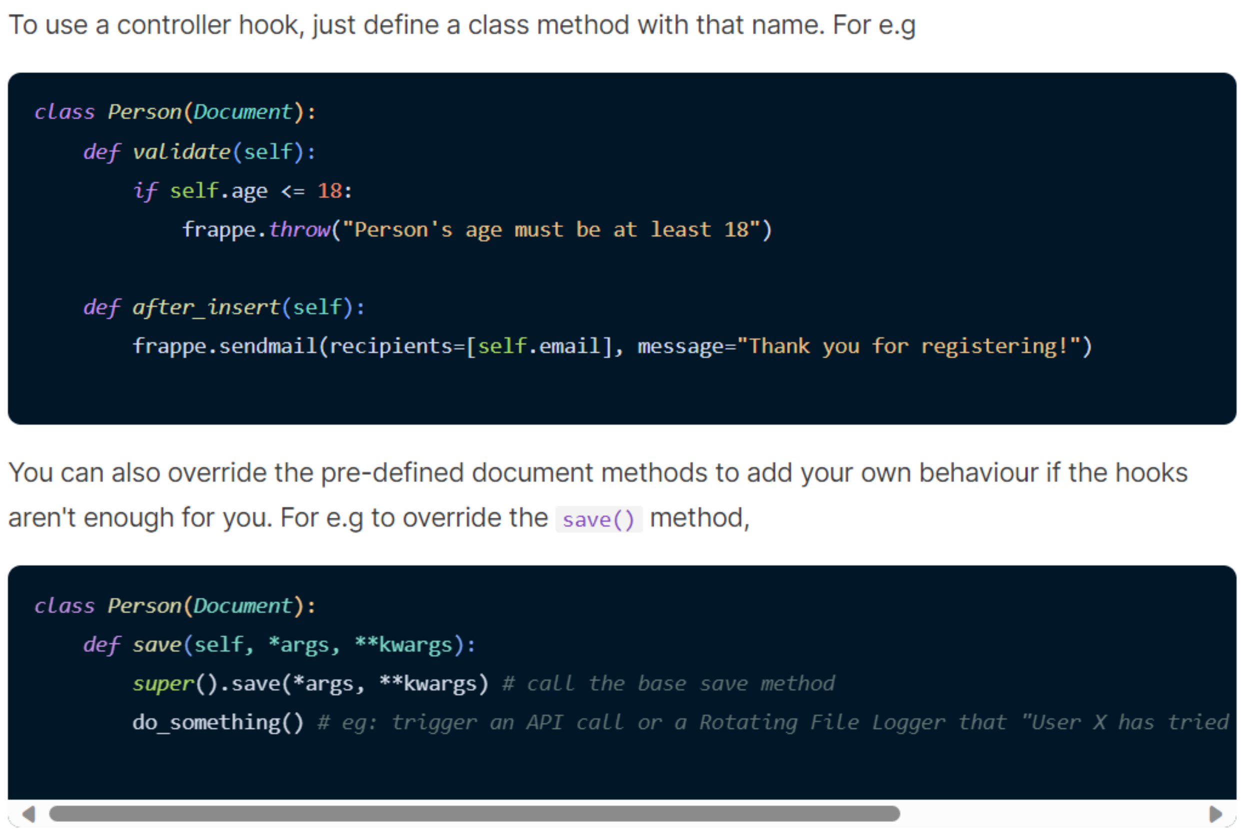
Controllers

In Frappe, a Controller is a Python class that extends from the `frappe.model.Document` base class.

Key Insights:

1. **Basics of Controllers**: When a DocType named "Person" is created, a file named `person.py` is generated. This file contains the basic structure of the Controller, which has access to all the fields of the DocType as attributes.
2. **Controller Methods**: Custom methods can be added to the Controller, and they can be invoked using the document object. For instance, a method `get_full_name` can be defined to return the full name of a person by combining their first and last names.
3. **Controller Hooks**: These are methods that get triggered at various stages in the lifecycle of a document. Some of the hooks include:
    - `before_insert`: Called before a document is prepared for insertion.
    - `validate`: Used to throw validation errors.
    - `before_save`: Triggered before the document is saved.
    - `on_submit`: Activated when a document is submitted.
    - `on_cancel`: Called when a submitted document is cancelled.
    - `on_update_after_submit`: Triggered when a submitted document's values are updated.
    - And many more...



To use a controller hook, just define a class method with that name. For e.g

```python
class Person(Document):
    def validate(self):
        if self.age <= 18:
            frappe.throw("Person's age must be at least 18")

    def after_insert(self):
        frappe.sendmail(recipients=[self.email], message="Thank you for registering!")
```

You can also override the pre-defined document methods to add your own behaviour if the hooks aren't enough for you. For e.g to override the `save()` method,

```python
class Person(Document):
    def save(self, *args, **kwargs):
        super().save(*args, **kwargs) # call the base save method
        do_something() # eg: trigger an API call or a Rotating File Logger that "User X has tried
```

4. **Special Rules**:
    - Child DocTypes don't follow naming rules.
    - Amended documents get a suffix added to their original name.
5. **Type Annotations**: From Version 15 onwards, Frappe supports auto-generating Python type annotations in controller files. These annotations aid in auto-completion, reference, and type-checking within the controller file.
6. **Document Creation & Loading**: The document can be created and saved to the database using the `frappe.get_doc` method. Similarly, existing documents can be fetched from the database using the same method.

More Explanation

In Frappe, Controllers are Python classes that define the behavior and properties of DocTypes. They contain methods that are executed at various points in the lifecycle of a document, such as when it is inserted, updated, submitted, cancelled, etc.

```
your_app/
```

```
├── your_app/
│   ├── doctype/
│   │   ├── your_doctype/
│   │   │   ├── your_doctype.py  # Server Controller
│   │   │   ├── your_doctype.js  # Form Controller
│   ├── public/
│   │   ├── js/
│   │   │   ├── custom_script.js  # Client Script (if not in Desk UI)
```

Server Controller

Perform on the server-side, ensuring data integrity, and performing operations that are too risky to be done on the client-side.

Some use cases:

- **Validate Data**: Ensure that the data adheres to the business logic before it gets stored.
- **Manipulate Data**: Modify data before it gets saved into the database.
- **Automate Processes**: Perform automatic actions like sending emails or creating new records.

Example:

```
class MonsterCard(Document):
    def on_submit(self):
        if self.is_legendary:
            frappe.get_doc({
                "doctype": "Legendary Collection",
                "monster_name": self.monster_name
            }).insert()
```

The code above will create & insert a new record automatically in the “Legendary Collection” DocType if the “Monster Card” new record is legendary.

Form Controller

Form Controllers can manipulate and validate the user interface in the Desk. **(DocType specific)**

Some use cases:

- **Validate Data**: Ensure that the data entered by the user adheres to specific rules.
- **Manipulate UI**: Hide, show, or modify fields and sections dynamically based on user input.
- **Fetch Data**: Retrieve data from other DocTypes and display it in the current form.

Example:

Imagine we have two DocTypes: **Spell Card** and **Duelist**. Each **Duelist** has a favorite **Spell Card**. When we select a **Duelist** in a form, we want to automatically display their favorite **Spell Card** in a read-only field.

```
// duel_record.js
frappe.ui.form.on('Duel Record', {
    // Trigger when the 'duelist' field value changes
    duelist: function(frm) {
        if (frm.doc.duelist) {
            // Fetch the 'favorite_spell' from the 'Duelist' DocType
            frappe.call({
                method: "frappe.client.get_value",
                args: {
                    doctype: "Duelist",
                    filters: {
                        name: frm.doc.duelist
                    },
                    fieldname: "favorite_spell"
                },
                callback: function(r) {
                    if (r.message) {
                        // Set the fetched 'favorite_spell' value to the 'favorite_spell' field
                        frm.set_value('favorite_spell', r.message.favorite_spell);
                    }
                }
            });
        }
    }
});
```

With this script, when you select a **Duelist** in the **Duel Record** form, their favorite **Spell Card** will be automatically fetched and displayed in the form.

Client Script

Client Scripts work on the client-side without the need to interact with the server. **(Can be applied to multiple DocTypes, more universal)**

Some use cases:

- **Manipulate UI**: Change visibility and values of fields dynamically.
- **Client-side Validations**: Ensure data integrity before it reaches the server.
- **Interact with Server**: Call whitelisted functions to fetch or manipulate data on the server.

Example:

```
frappe.ui.form.on('Duel Log', {
    monster_name: function(frm) {
        if (frm.doc.monster_name === "Dark Magician") {
            frm.set_df_property('magic_attack', 'hidden', false);
        } else {
            frm.set_df_property('magic_attack', 'hidden', true);
        }
    }
});
```

If "Dark Magician" is selected as the monster name, a field named "magic_attack" becomes visible.

Whitelisted Function

Whitelisted Functions are server-side Python functions that are explicitly marked to be accessible via API calls or client-side scripts.

Imagine you have a treasure trove (database) and you've employed various sorcerers (server-side functions) to manage it. You wouldn't want just anyone to command your sorcerers, would you? Whitelisting acts as a protective barrier, only allowing specific spells (functions) to be cast (called) from the outside world (client-side/browser).

Once a function is whitelisted, it can be invoked from the client-side using `frappe.call()`.

Example:

```
frappe.call({
    method: "myapp.mymodule.my_whitelisted_function",  // path to server method
    args: {
        arg1: "Dark Magician",
        arg2: "Blue-Eyes White Dragon"
    },
    callback: function(r) {
        if(r.message) {
            console.log(r.message);  // Output: "The Pharaoh approves!"
        }
    }
});
```

A Few Notes to Ponder:

- **Security**: Whitelisting ensures that not all server-side functions can be accessed from the client-side, providing a layer of security.
- **Arguments**: You can pass arguments to your whitelisted function using the `args` property in `frappe.call()`.
- **Callback**: The `callback` function is used to handle the data returned from the server-side function.

More Examples

▼ **Data Validation**: Ensure that the data entered by users adheres to specific rules and business logic. (Desk: Yes)

```
class Employee(Document):
    def validate(self):
        if self.date_of_birth >= getdate():
            frappe.throw(_("Date of Birth must be before today"))
```

In this example, when an Employee document is saved or submitted, it checks whether the date of birth is before today's date and throws an error if it's not.

Through Desk UI: Yes, you can validate data using Client Scripts by checking field values and ensuring they adhere to specific rules, providing immediate feedback to the user.

▼ **Automate Processes**: Automatically create or update related documents when a particular document is created or updated. (Desk: Partial)

```
class SalesInvoice(Document):
    def on_submit(self):
        # Create a new Ledger Entry when a Sales Invoice is submitted
        ledger_entry = frappe.new_doc("Ledger Entry")
        ledger_entry.invoice = self.name
        ledger_entry.amount = self.total
        ledger_entry.posting_date = self.posting_date
        ledger_entry.insert()
```

Here, when a Sales Invoice is submitted, a new Ledger Entry document is automatically created and linked to the invoice.

Through Desk UI: Limited. While you can manipulate fields and perform some automated tasks using Client Scripts, creating or updating related documents is restricted due to security reasons.

▼ **Manage Side Effects**: Perform side effects like sending emails or notifications when certain events occur. (Desk: Partial)

```
class LeaveApplication(Document):
    def on_submit(self):
        # Send an email when a Leave Application is submitted
        frappe.sendmail(
            recipients=[self.employee_email],
            subject="Leave Application Submitted",
            message=f"Your leave application for {self.from_date} to {self.to_date} has been submitted."
        )
```

This example sends an email to the employee when their Leave Application is submitted.

Through Desk UI: Partial. You can perform some UI side effects like hiding/showing fields or changing field properties. However, sending emails or notifications typically requires server-side logic.

▼ **Interact with Other Systems**: Communicate with other systems or services, for example, by making HTTP requests to external APIs. (Desk: Limited)

```
import requests

class PaymentEntry(Document):
    def on_submit(self):
        # Send payment info to an external payment gateway
        response = requests.post(
            "https://api.paymentgateway.com/pay",
            json={
                "amount": self.amount,
                "payee": self.payee,
                # other payment details
            }
        )
        if response.status_code != 200:
            frappe.throw(_("Payment failed"))
```

Upon submitting a Payment Entry, it sends payment details to an external payment gateway API.

Through Desk UI: Limited. While it's possible to make AJAX calls to external APIs using JavaScript in Client Scripts, it's not recommended due to security concerns (e.g., exposing API keys). Server-side interactions are preferable.

▼ **Define Custom Endpoints**: Create custom API endpoints to expose specific functionality of your application. (Desk: No)

```
@frappe.whitelist()
def get_employee_birthdays(month):
    # Fetch employees whose birthdays are in the specified month
    return frappe.get_all("Employee", filters={"month(date_of_birth)": month}, fields=["name", "date_of_bir
```

This function, when added to an API script in Frappe, provides an endpoint that returns employees with birthdays in a specified month.

Through Desk UI: No. Defining custom API endpoints is a server-side operation and cannot be performed through the Desk UI.

▼ **Data Manipulation**: Manipulate data before it is saved or after it is retrieved from the database. (Desk: Yes, to an extent)

```
class SalesOrder(Document):
    def validate(self):
        # Ensure the total is always updated before saving
        self.total = sum([item.amount for item in self.items])
```

This ensures that the total amount in a Sales Order is always the sum of the amounts of its items whenever it's saved.

Through Desk UI: Yes, to an extent. You can manipulate data on the client side (like changing field values based on user input) but manipulating data before saving or after fetching from the database is best done server-side.

▼ **User Feedback**: Provide feedback to users by raising validation messages or creating log entries. (Desk: Yes)

```
class Meeting(Document):
    def validate(self):
        # Check if the meeting time is available
        overlapping_meetings = frappe.get_list("Meeting", filters={
            "start_time": ("<", self.end_time),
            "end_time": (">", self.start_time),
            "name": ("!=", self.name)
        })
        if overlapping_meetings:
            frappe.msgprint(_("There are overlapping meetings"))
```

This example checks for overlapping meetings and shows a message to the user if any are found.

Through Desk UI: Yes. You can provide instant feedback using alert messages or validation messages through Client Scripts.

▼ **Security**: Implement custom security checks to ensure that users can only perform actions they are authorized to perform. (Desk: Limited)

```
class SensitiveData(Document):
    def validate(self):
        # Ensure only authorized users can create or modify
        if frappe.session.user not in ["Authorized User 1", "Authorized User 2"]:
            frappe.throw(_("You are not authorized to create or modify this document"))
```

This example ensures that only specified users can create or modify SensitiveData documents.

Through Desk UI: Limited. While you can hide/show fields based on user roles or input, true security checks (like permission verification) must be implemented server-side to ensure they cannot be bypassed.