# Structured Summary of Odoo 18.0 Server Framework 101 Tutorial for Beginner Odoo Developers

- Odoo 18.0 follows a three-tier architecture: presentation (HTML5, JS, CSS), business logic (Python), and data storage (PostgreSQL).
- Modules are the building blocks of Odoo applications, defined by a manifest file and containing models, views, data, and controllers.
- Key concepts include models and fields, security rules, views (tree, form, kanban), relationships between models, computed fields, actions, constraints, inheritance, and module interactions.
- Practical examples demonstrate how to define models, create views, manage security, implement business logic, and extend functionality through inheritance and module integration.
- Following coding guidelines and best practices ensures maintainable, secure, and user-friendly Odoo applications.

## Introduction

The Odoo 18.0 Server Framework 101 tutorial is designed to introduce beginner developers to the foundational concepts and practical aspects of Odoo development. Odoo is a powerful open-source ERP and CRM platform that relies on a modular architecture to provide extensible business applications. This tutorial guides developers through creating a real estate module, incrementally introducing core concepts such as architecture, models, views, security, and advanced features like computed fields and inheritance. The goal is to provide a clear, structured pathway for developers to gain confidence and competence in building custom Odoo modules.

## Chapter 1: Architecture Overview

**Core Concepts**:
Odoo's architecture is a three-tier system separating presentation, business logic, and data storage. The presentation tier uses HTML5, JavaScript, and CSS, while business logic is implemented in Python. Data storage relies on PostgreSQL. Odoo modules are self-contained packages that include models (business objects), views (UI definitions), data files (security and configurations), and web controllers. The **manifest file** (`__manifest__.py`) is essential as it declares the module's metadata and dependencies. Odoo is evolving its presentation tier with the OWL framework, which improves UI development.

**Key Takeaways**:
- Understand Odoo's three-tier architecture and the role of each tier.
- Recognize the importance of modules and their structure in Odoo development.
- Identify the purpose of the manifest file in module declaration.
- Be aware of Odoo's transition to OWL for modern UI development.

**Code/Configuration Highlights**:

```
# Typical module directory structure
module/
├── models/           # Contains Python model definitions
│   ├── *.py
│   └── __init__.py
├── data/             # Contains XML data and security files
│   └── *.xml
├── __init__.py       # Python package initialization
└── __manifest__.py   # Module manifest file
```

**Pitfalls/Tips**:
- Ensure you have basic HTML knowledge and intermediate Python skills before starting.
- Refer to the official Python tutorial if needed to strengthen Python fundamentals.

**Why It Matters**:
A solid grasp of Odoo's architecture and module structure is critical for developing custom applications and extending existing functionality. It forms the foundation for creating well-organized, maintainable, and scalable Odoo modules.

## Chapter 2: A New Application

**Core Concepts**:
This chapter walks through creating a new Odoo application from scratch. It emphasizes setting up the development environment correctly, creating a new module directory, and generating the necessary files. The setup guide provides essential steps to prepare the environment, including installing dependencies and configuring the Odoo server.

**Key Takeaways**:
- Follow the setup guide to properly configure your development environment.
- Create a new module directory with essential files (`__init__.py`, `__manifest__.py`).
- Understand the basic structure of an Odoo module.

**Code/Configuration Highlights**:

```
# Commands to create a new module directory and files
mkdir my_module
cd my_module
touch __init__.py __manifest__.py
```

**Pitfalls/Tips**:
- Verify your environment setup before proceeding to avoid issues.
- Adhere to the recommended directory structure for consistency.

**Why It Matters**:
Creating a new application is the first practical step in Odoo development. A well-structured module directory and properly configured environment are essential for successful development.

## Chapter 3: Models And Basic Fields

**Core Concepts**:
Models in Odoo represent business objects and are defined as Python classes inheriting from `models.Model`. Fields within these classes define the data structure and are mapped to database columns via Odoo's ORM. This chapter explains how to create models, define basic fields (e.g., Char, Text), and manage them within a module.

**Key Takeaways**:
- Define business objects as Python classes inheriting from `models.Model`.
- Understand how fields are mapped to database columns.
- Create and manage models within your module.

**Code/Configuration Highlights**:

```
from odoo import models, fields

class MyModel(models.Model):
    _name = 'my.model'          # Model name
    _description = 'My Model'   # Description

    name = fields.Char(string='Name')          # Character field
    description = fields.Text(string='Description')  # Text field
```

**Pitfalls/Tips**:
- Always inherit from `models.Model` when creating new models.
- Use descriptive names for models and fields to improve code clarity.

**Why It Matters**:
Models and fields are fundamental to Odoo applications as they define the data structure and business logic. Properly defined models ensure data integrity and facilitate business process automation.

## Chapter 4: Security - A Brief Introduction

**Core Concepts**:
Security in Odoo is managed through security rules defined in XML data files. These rules control access to models and data by defining groups, permissions, and access rights. This

chapter introduces basic security concepts and demonstrates how to set up security rules within a module.

**Key Takeaways**:
- Understand the role of security rules in Odoo.
- Define security rules in XML data files.
- Control access to models and data using groups and permissions.

**Code/Configuration Highlights**:

```
<odoo>
    <data noupdate="1">
        <record id="group_my_module_user" model="res.groups">
            <field name="name">My Module User</field>
            <field name="category_id" ref="base.module_category_hidden"/>
        </record>
    </data>
</odoo>
```

**Pitfalls/Tips**:
- Ensure security rules are correctly defined to prevent unauthorized access.
- Follow best practices for security rule setup.

**Why It Matters**:
Security is critical to protect sensitive data and ensure proper access control within Odoo applications. Implementing security rules correctly prevents data breaches and unauthorized actions.

## Chapter 5: Finally, Some UI To Play With

**Core Concepts**:
This chapter introduces the user interface (UI) development in Odoo, focusing on creating and managing UI elements using XML files. It covers defining views and other UI components within a module, which are essential for user interaction.

**Key Takeaways**:
- Create and manage UI elements using XML files.
- Define views and UI components within your module.
- Understand the basic structure of Odoo's UI components.

**Code/Configuration Highlights**:

```
<odoo>
    <data>
        <record id="view_my_model_tree" model="ir.ui.view">
            <field name="name">my.model.tree</field>
            <field name="model">my.model</field>
            <field name="arch" type="xml">
                <tree>
```

```
                    <field name="name"/>
                    <field name="description"/>
                </tree>
            </field>
        </record>
    </data>
</odoo>
```

**Pitfalls/Tips**:
- Ensure views are correctly defined to avoid UI rendering issues.
- Follow recommended practices for UI element creation.

**Why It Matters**:
The UI is crucial for user interaction and experience. Well-defined views and UI components ensure that the application is intuitive and user-friendly.

## Chapter 6: Basic Views

**Core Concepts**:
This chapter explores the different types of views in Odoo, including tree views, form views, and kanban views. It explains how to create and customize these views using XML files to fit specific application needs.

**Key Takeaways**:
- Understand the different types of views (tree, form, kanban).
- Create and customize views using XML files.
- Manage views to match application requirements.

**Code/Configuration Highlights**:

```
<odoo>
    <data>
        <record id="view_my_model_form" model="ir.ui.view">
            <field name="name">my.model.form</field>
            <field name="model">my.model</field>
            <field name="arch" type="xml">
                <form>
                    <sheet>
                        <group>
                            <field name="name"/>
                            <field name="description"/>
                        </group>
                    </sheet>
                </form>
            </field>
        </record>
    </data>
</odoo>
```

**Pitfalls/Tips**:
- Ensure views are correctly defined to avoid UI issues.
- Follow best practices for view customization.

**Why It Matters**:
Views determine how data is presented to users. Well-designed views enhance usability and ensure a smooth user experience.

## Chapter 7: Relations Between Models

**Core Concepts**:
This chapter covers relationships between models, such as one-to-many and many-to-many relationships. It explains how to define and manage these relationships using Python classes, which is essential for modeling complex business data structures.

**Key Takeaways**:
- Define and manage relationships between models (one-to-many, many-to-many).
- Use Python classes to create and manage relationships.
- Understand the importance of relationships in Odoo's data model.

**Code/Configuration Highlights**:

```
from odoo import models, fields

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')
    related_ids = fields.One2many('related.model', 'my_model_id', string='Related Models')
```

**Pitfalls/Tips**:
- Ensure relationships are correctly defined to avoid data integrity issues.
- Follow recommended practices for relationship management.

**Why It Matters**:
Relationships between models are fundamental to modeling complex business processes and ensuring data consistency.

## Chapter 8: Computed Fields And Onchanges

**Core Concepts**:
This chapter introduces computed fields and onchanges. Computed fields depend on other fields and are automatically calculated, while onchanges trigger actions when field values change. These features are crucial for implementing business logic and ensuring data integrity.

**Key Takeaways**:
- Define computed fields that depend on other fields.
- Use onchanges to trigger actions on field value changes.
- Understand the role of computed fields and onchanges in Odoo's data model.

**Code/Configuration Highlights**:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')
    computed_field = fields.Char(string='Computed Field', compute='_compute_computed_field')

    @api.depends('name', 'description')
    def _compute_computed_field(self):
        for record in self:
            record.computed_field = f"{record.name} - {record.description}"
```

**Pitfalls/Tips**:
- Ensure computed fields and onchanges are correctly defined to avoid data integrity issues.
- Follow best practices for creating and managing these fields.

**Why It Matters**:
Computed fields and onchanges are essential for implementing business logic and maintaining data consistency.

## Chapter 9: Ready For Some Action?

**Core Concepts**:
This chapter covers actions in Odoo, which are methods that trigger specific behaviors, such as opening a view or executing a method. Actions enhance the functionality of the application by enabling user interactions and automations.

**Key Takeaways**:
- Define and manage actions within your module.
- Create actions that trigger specific behaviors.
- Use actions to enhance application functionality.

**Code/Configuration Highlights**:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'
```

```
    name = fields.Char(string='Name')
    description = fields.Text(string='Description')

    def my_action(self):
        # Action logic here
        pass
```

**Pitfalls/Tips**:
- Ensure actions are correctly defined to avoid functionality issues.
- Follow recommended practices for action creation.

**Why It Matters**:
Actions are crucial for defining behavior and functionality, enabling user interactions and automations.

## Chapter 10: Constraints

**Core Concepts**:
Constraints in Odoo ensure data integrity by enforcing rules on field values. This chapter covers defining constraints using Python classes, including unique constraints and check constraints.

**Key Takeaways**:
- Define and manage constraints within your module.
- Understand different types of constraints (unique, check).
- Use constraints to ensure data integrity.

**Code/Configuration Highlights**:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')

    @api.constrains('name')
    def _check_name(self):
        for record in self:
            if len(record.name) < 3:
                raise models.ValidationError("Name must be at least 3 characters long.")
```

**Pitfalls/Tips**:
- Ensure constraints are correctly defined to avoid data integrity issues.
- Follow best practices for constraint management.

**Why It Matters**:
Constraints are essential for maintaining data integrity and robustness in Odoo applications.

## Chapter 11: Add The Sprinkles

**Core Concepts**:
This chapter covers additional features and enhancements that can be added to an Odoo module, such as custom logic, integration with external systems, and UI improvements.

**Key Takeaways**:
- Add custom logic to your module.
- Integrate with external systems.
- Enhance the user interface for better user experience.

**Code/Configuration Highlights**:

```
from odoo import models, fields, api

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')

    def my_custom_method(self):
        # Custom logic here
        pass
```

**Pitfalls/Tips**:
- Ensure custom logic and integrations are correctly implemented to avoid functionality issues.
- Follow best practices for adding features.

**Why It Matters**:
Adding custom logic and integrating with external systems enhances functionality and user experience.

## Chapter 12: Inheritance

**Core Concepts**:
Inheritance allows models to inherit from existing models, extending or overriding their functionality. This chapter explains how to define inherited models and manage them within a module.

**Key Takeaways**:
- Define and manage inherited models within your module.
- Create models that inherit from existing models.
- Override and extend the functionality of inherited models.

**Code/Configuration Highlights**:

```
from odoo import models, fields

class MyInheritedModel(models.Model):
    _name = 'my.inherited.model'
    _description = 'My Inherited Model'
    _inherit = 'my.model'

    additional_field = fields.Char(string='Additional Field')
```

**Pitfalls/Tips**:
- Ensure inherited models are correctly defined to avoid functionality issues.
- Follow best practices for inheritance.

**Why It Matters**:
Inheritance is a powerful feature for extending and overriding functionality, promoting code reuse and maintainability.

## Chapter 13: Interact With Other Modules

**Core Concepts**:
This chapter covers interacting with other modules in Odoo, including using functionality from other modules and integrating with external modules.

**Key Takeaways**:
- Use functionality from other modules within your module.
- Integrate with external modules to enhance functionality.
- Understand the importance of module interactions.

**Code/Configuration Highlights**:

```
from odoo import models, fields

class MyModel(models.Model):
    _name = 'my.model'
    _description = 'My Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')

    def interact_with_other_module(self):
        # Logic to interact with another module
        pass
```

**Pitfalls/Tips**:
- Ensure interactions with other modules are correctly implemented to avoid functionality issues.
- Follow best practices for module integration.

**Why It Matters**:
Interacting with other modules is crucial for extending functionality and integrating with external systems.

## Chapter 14: A Brief History Of QWeb

**Core Concepts**:
QWeb is Odoo's templating engine used to create and manage templates for views and reports. This chapter provides a brief history of QWeb and its role in Odoo's presentation tier.

**Key Takeaways**:
- Understand QWeb's evolution and role in Odoo's presentation tier.
- Use QWeb to create and manage templates for views and reports.
- Appreciate QWeb's importance in UI development.

**Code/Configuration Highlights**:

```
<odoo>
    <data>
        <record id="view_my_model_qweb" model="ir.ui.view">
            <field name="name">my.model.qweb</field>
            <field name="model">my.model</field>
            <field name="arch" type="xml">
                <t t-name="my.model.qweb">
                    <div>
                        <h1>My Model</h1>
                        <p>Name: <span t-field="name"/></p>
                        <p>Description: <span t-field="description"/></p>
                    </div>
                </t>
            </field>
        </record>
    </data>
</odoo>
```

**Pitfalls/Tips**:
- Ensure QWeb templates are correctly used to avoid UI issues.
- Follow best practices for template creation.

**Why It Matters**:
QWeb is essential for creating robust and maintainable UI templates in Odoo.

## Chapter 15: The Final Word

**Core Concepts**:
This chapter summarizes the tutorial, emphasizing the importance of following recommended practices and guidelines for Odoo development. It encourages further exploration and learning.

**Key Takeaways**:
- Follow recommended practices and guidelines for Odoo development.
- Continue exploring and learning to enhance Odoo development skills.
- Apply the knowledge gained to real-world Odoo development tasks.

**Code/Configuration Highlights**:

```
from odoo import models, fields

class MyRealWorldModel(models.Model):
    _name = 'my.real.world.model'
    _description = 'My Real World Model'

    name = fields.Char(string='Name')
    description = fields.Text(string='Description')

    def my_real_world_method(self):
        # Real-world logic here
        pass
```

**Pitfalls/Tips**:
- Ensure knowledge is correctly applied to avoid functionality issues.
- Follow best practices for real-world development.

**Why It Matters**:
Applying tutorial knowledge to real-world tasks is crucial for becoming a proficient Odoo developer.

## Conclusion

The Odoo 18.0 Server Framework 101 tutorial provides a comprehensive, structured introduction to Odoo development for beginners. It covers essential concepts including architecture, module structure, models, views, security, relationships, computed fields, actions, constraints, inheritance, module interactions, and QWeb templating. Each chapter builds upon the previous, guiding developers through creating a real estate module incrementally.

By following this tutorial, beginner developers can gain a solid foundation in Odoo development, understanding both the theoretical and practical aspects of building custom modules. The tutorial emphasizes following coding guidelines and best practices to ensure the creation of robust, maintainable, and user-friendly Odoo applications. This structured summary serves as a valuable reference for developers starting their journey with Odoo 18.0.