Phase 4: No-Code Form & Workflow Builders - TWIST ERP

Implementation Guide

Duration: 8–10 weeks
Version: 1.0
Date: October 2025
Project: TWIST ERP - Visual Drag-and-Drop Multi-Company ERP

1. Phase Overview

Phase 4 empowers non-technical users to create custom forms, modules, and automated workflows without writing code. This is a key differentiator for TWIST ERP, enabling SMEs to adapt the system to their unique business processes.

Key Objectives

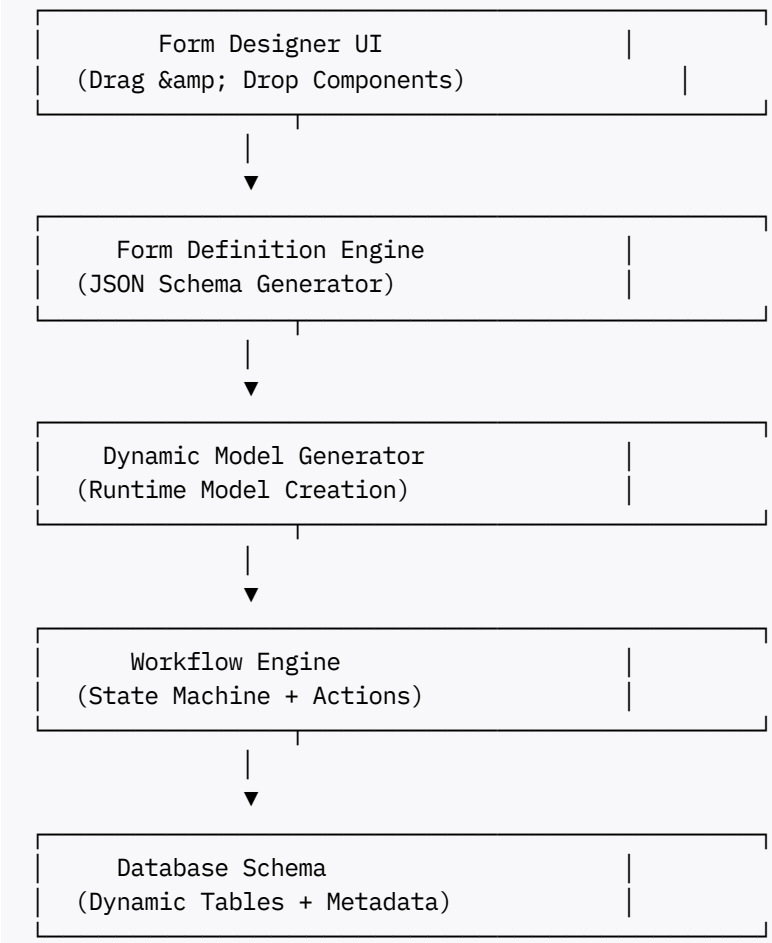
- Build visual drag-and-drop form designer
- Implement no-code module creator
- Create workflow automation studio with visual editor
- Enable dynamic field types and validations
- Support conditional logic and calculated fields
- Implement approval workflows with multi-level routing
- Create form template library
- Enable module versioning and publishing

Success Criteria

- Users can create functional forms in < 10 minutes
- Zero coding required for form creation
- Workflow automation reduces manual tasks by 60%
- Custom modules deployable without developer
- Form/workflow templates reusable across companies
- Real-time preview during form design

2. Architecture Overview

2.1 Component Structure



3. Form Builder Models

3.1 Form Definition

```
# backend/apps/form_builder/models/form.py
from django.db import models
from shared.models import CompanyAwareModel
import json

class FormCategory(models.TextChoices):
    MASTER_DATA = 'MASTER', 'Master Data'
    TRANSACTION = 'TRANSACTION', 'Transaction'
    REPORT = 'REPORT', 'Report'
    WORKFLOW = 'WORKFLOW', 'Workflow Form'

class Form(CompanyAwareModel):
    """
    Custom form definition
```

```
    """
    name = models.CharField(max_length=255)
    code = models.CharField(max_length=50)
    description = models.TextField(blank=True)

    category = models.CharField(
        max_length=20,
        choices=FormCategory.choices,
        default=FormCategory.TRANSACTION
    )

    # Form schema (JSON)
    schema = models.JSONField(
        default=dict,
        help_text="Form field definitions"
    )

    # UI layout
    layout = models.JSONField(
        default=dict,
        help_text="Visual layout configuration"
    )

    # Validation rules
    validation_rules = models.JSONField(default=list)

    # Calculations
    calculated_fields = models.JSONField(default=list)

    # Permissions
    can_create_roles = models.ManyToManyField(
        'permissions.Role',
        related_name='creatable_forms',
        blank=True
    )
    can_view_roles = models.ManyToManyField(
        'permissions.Role',
        related_name='viewable_forms',
        blank=True
    )
    can_edit_roles = models.ManyToManyField(
        'permissions.Role',
        related_name='editable_forms',
        blank=True
    )

    # Settings
    is_active = models.BooleanField(default=True)
    is_template = models.BooleanField(default=False)
    version = models.IntegerField(default=1)

    # Linked module
    custom_module = models.ForeignKey(
        'CustomModule',
        on_delete=models.SET_NULL,
        null=True,
```

```
        blank=True
    )

    class Meta:
        unique_together = [['company', 'code']]
        ordering = ['name']
        indexes = [
            models.Index(fields=['company', 'category']),
            models.Index(fields=['company', 'is_active']),
        ]

    def __str__(self):
        return f"{self.name} (v{self.version})"

    def get_field_names(self):
        """Extract field names from schema"""
        return [field['name'] for field in self.schema.get('fields', [])]

    def validate_schema(self):
        """Validate form schema structure"""
        required_keys = ['fields', 'sections']
        for key in required_keys:
            if key not in self.schema:
                raise ValueError(f"Schema missing required key: {key}")

        return True

class FormField(models.Model):
    """
    Individual form field definition
    Helper model for better querying
    """
    form = models.ForeignKey(
        Form,
        on_delete=models.CASCADE,
        related_name='fields'
    )

    field_name = models.CharField(max_length=100)
    field_type = models.CharField(
        max_length=50,
        choices=[
            ('text', 'Text'),
            ('number', 'Number'),
            ('date', 'Date'),
            ('datetime', 'Date Time'),
            ('select', 'Dropdown'),
            ('multiselect', 'Multi Select'),
            ('checkbox', 'Checkbox'),
            ('radio', 'Radio Button'),
            ('textarea', 'Text Area'),
            ('email', 'Email'),
            ('phone', 'Phone'),
            ('currency', 'Currency'),
            ('percentage', 'Percentage'),
            ('file', 'File Upload'),
```

```
            ('image', 'Image Upload'),
            ('lookup', 'Lookup/Reference'),
            ('calculated', 'Calculated Field'),
        ]
    )

    label = models.CharField(max_length=255)
    placeholder = models.CharField(max_length=255, blank=True)
    help_text = models.TextField(blank=True)

    # Validation
    is_required = models.BooleanField(default=False)
    is_unique = models.BooleanField(default=False)
    min_length = models.IntegerField(null=True, blank=True)
    max_length = models.IntegerField(null=True, blank=True)
    min_value = models.DecimalField(
        max_digits=20,
        decimal_places=2,
        null=True,
        blank=True
    )
    max_value = models.DecimalField(
        max_digits=20,
        decimal_places=2,
        null=True,
        blank=True
    )
    regex_pattern = models.CharField(max_length=500, blank=True)

    # Options (for select/radio)
    options = models.JSONField(
        default=list,
        help_text="List of options for select/radio fields"
    )

    # Lookup configuration
    lookup_model = models.CharField(max_length=100, blank=True)
    lookup_field = models.CharField(max_length=100, blank=True)
    lookup_filters = models.JSONField(default=dict)

    # Calculation formula
    formula = models.TextField(
        blank=True,
        help_text="Formula for calculated fields"
    )

    # Conditional visibility
    visibility_condition = models.JSONField(
        default=dict,
        help_text="Condition for showing this field"
    )

    # UI properties
    order = models.IntegerField(default=0)
    width = models.CharField(
        max_length=20,
```

```
        default='full',
        choices=[
            ('full', 'Full Width'),
            ('half', 'Half Width'),
            ('third', 'Third Width'),
            ('quarter', 'Quarter Width'),
        ]
    )

    class Meta:
        ordering = ['form', 'order']
        unique_together = [['form', 'field_name']]
```

3.2 Form Data Storage

```
# backend/apps/form_builder/models/data.py
from django.db import models
from django.contrib.postgres.fields import JSONField

class FormSubmission(models.Model):
    """
    Stores submitted form data
    Uses flexible JSON storage for custom fields
    """
    form = models.ForeignKey(
        'Form',
        on_delete=models.PROTECT,
        related_name='submissions'
    )
    company = models.ForeignKey(
        'companies.Company',
        on_delete=models.PROTECT
    )

    # Flexible data storage
    data = models.JSONField(
        default=dict,
        help_text="Form field values"
    )

    # Metadata
    submission_number = models.CharField(max_length=50)
    status = models.CharField(
        max_length=20,
        choices=[
            ('DRAFT', 'Draft'),
            ('SUBMITTED', 'Submitted'),
            ('APPROVED', 'Approved'),
            ('REJECTED', 'Rejected'),
            ('CANCELLED', 'Cancelled'),
        ],
        default='DRAFT'
    )

    # Workflow
```

```
    workflow_state = models.CharField(max_length=50, blank=True)
    current_approver = models.ForeignKey(
        'users.User',
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name='pending_approvals'
    )

    # Audit
    submitted_by = models.ForeignKey(
        'users.User',
        on_delete=models.PROTECT,
        related_name='submitted_forms'
    )
    submitted_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-submitted_at']
        indexes = [
            models.Index(fields=['company', 'form', 'status']),
            models.Index(fields=['company', 'current_approver']),
        ]

    def __str__(self):
        return f"{self.form.name} - {self.submission_number}"

    def get_field_value(self, field_name):
        """Get value of specific field"""
        return self.data.get(field_name)

    def set_field_value(self, field_name, value):
        """Set value of specific field"""
        self.data[field_name] = value
        self.save()
```

4. Custom Module Builder

4.1 Module Definition

```
# backend/apps/form_builder/models/module.py
from django.db import models
from shared.models import CompanyAwareModel

class CustomModule(CompanyAwareModel):
    """
    User-created custom module
    """
    name = models.CharField(max_length=255)
    code = models.CharField(max_length=50)
    description = models.TextField(blank=True)
    icon = models.CharField(max_length=50, default='cube')
```

```
    # Module type
    module_type = models.CharField(
        max_length=20,
        choices=[
            ('MASTER', 'Master Data'),
            ('TRANSACTION', 'Transactional'),
            ('REPORT', 'Reporting'),
            ('UTILITY', 'Utility'),
        ],
        default='TRANSACTION'
    )

    # Entity definition
    entity_name = models.CharField(max_length=100)
    entity_plural = models.CharField(max_length=100)

    # Fields (similar to form schema)
    fields_schema = models.JSONField(default=list)

    # UI Configuration
    list_view_config = models.JSONField(
        default=dict,
        help_text="Configuration for list/grid view"
    )
    detail_view_config = models.JSONField(
        default=dict,
        help_text="Configuration for detail view"
    )

    # Relationships
    has_many = models.JSONField(
        default=list,
        help_text="One-to-many relationships"
    )
    belongs_to = models.JSONField(
        default=list,
        help_text="Many-to-one relationships"
    )
    many_to_many = models.JSONField(
        default=list,
        help_text="Many-to-many relationships"
    )

    # Permissions
    permissions = models.JSONField(
        default=dict,
        help_text="Module-level permissions"
    )

    # Settings
    is_active = models.BooleanField(default=True)
    is_published = models.BooleanField(default=False)
    version = models.IntegerField(default=1)

    # Menu
```

```
    show_in_menu = models.BooleanField(default=True)
    menu_group = models.CharField(max_length=100, blank=True)
    menu_order = models.IntegerField(default=0)

    class Meta:
        unique_together = [['company', 'code']]
        ordering = ['menu_order', 'name']

    def __str__(self):
        return self.name

    def get_database_table_name(self):
        """Generate database table name"""
        return f"custom_{self.company.code.lower()}_{self.code.lower()}"

    def get_api_endpoint(self):
        """Get API endpoint for this module"""
        return f"/api/v1/custom/{self.code.lower()}/"

class CustomModuleField(models.Model):
    """
    Field definition for custom module
    """
    module = models.ForeignKey(
        CustomModule,
        on_delete=models.CASCADE,
        related_name='fields'
    )

    field_name = models.CharField(max_length=100)
    field_type = models.CharField(max_length=50)
    label = models.CharField(max_length=255)

    # Same field types as FormField
    # Validation, options, etc.

    is_required = models.BooleanField(default=False)
    is_unique = models.BooleanField(default=False)
    is_indexed = models.BooleanField(default=False)

    # Display in lists
    show_in_list = models.BooleanField(default=False)
    list_order = models.IntegerField(default=0)

    order = models.IntegerField(default=0)

    class Meta:
        ordering = ['module', 'order']
        unique_together = [['module', 'field_name']]
```

5. Workflow Engine

5.1 Workflow Definition

```
# backend/apps/workflows/models/workflow.py
from django.db import models
from shared.models import CompanyAwareModel

class Workflow(CompanyAwareModel):
    """
    Workflow definition with state machine
    """
    name = models.CharField(max_length=255)
    code = models.CharField(max_length=50)
    description = models.TextField(blank=True)

    # Trigger
    trigger_type = models.CharField(
        max_length=20,
        choices=[
            ('MANUAL', 'Manual Trigger'),
            ('ON_CREATE', 'On Record Create'),
            ('ON_UPDATE', 'On Record Update'),
            ('ON_DELETE', 'On Record Delete'),
            ('SCHEDULED', 'Scheduled'),
            ('WEBHOOK', 'Webhook Trigger'),
        ]
    )

    # Target
    target_module = models.CharField(max_length=100)
    target_model = models.CharField(max_length=100)

    # Workflow definition (graph)
    states = models.JSONField(
        default=list,
        help_text="List of workflow states"
    )
    transitions = models.JSONField(
        default=list,
        help_text="State transitions"
    )
    actions = models.JSONField(
        default=list,
        help_text="Actions to execute at each state"
    )

    # Conditions
    start_condition = models.JSONField(
        default=dict,
        help_text="Condition to start workflow"
    )

    # Settings
    is_active = models.BooleanField(default=True)
```

```
    version = models.IntegerField(default=1)

    class Meta:
        unique_together = [['company', 'code']]
        ordering = ['name']

    def __str__(self):
        return self.name

class WorkflowState(models.Model):
    """
    Individual state in workflow
    """
    workflow = models.ForeignKey(
        Workflow,
        on_delete=models.CASCADE,
        related_name='state_definitions'
    )

    state_code = models.CharField(max_length=50)
    state_name = models.CharField(max_length=255)
    description = models.TextField(blank=True)

    # State type
    state_type = models.CharField(
        max_length=20,
        choices=[
            ('START', 'Start State'),
            ('INTERMEDIATE', 'Intermediate'),
            ('APPROVAL', 'Approval Required'),
            ('END', 'End State'),
            ('ERROR', 'Error State'),
        ]
    )

    # Actions to execute when entering this state
    on_enter_actions = models.JSONField(default=list)
    on_exit_actions = models.JSONField(default=list)

    # Timeout
    timeout_minutes = models.IntegerField(
        null=True,
        blank=True,
        help_text="Auto-transition after timeout"
    )
    timeout_transition = models.CharField(
        max_length=50,
        blank=True,
        help_text="Transition to execute on timeout"
    )

    # Approval
    requires_approval = models.BooleanField(default=False)
    approval_role = models.ForeignKey(
        'permissions.Role',
        on_delete=models.SET_NULL,
```

```
        null=True,
        blank=True
    )

    order = models.IntegerField(default=0)

    class Meta:
        ordering = ['workflow', 'order']
        unique_together = [['workflow', 'state_code']]

class WorkflowTransition(models.Model):
    """
    Transition between states
    """
    workflow = models.ForeignKey(
        Workflow,
        on_delete=models.CASCADE,
        related_name='transition_definitions'
    )

    from_state = models.ForeignKey(
        WorkflowState,
        on_delete=models.CASCADE,
        related_name='outgoing_transitions'
    )
    to_state = models.ForeignKey(
        WorkflowState,
        on_delete=models.CASCADE,
        related_name='incoming_transitions'
    )

    transition_code = models.CharField(max_length=50)
    transition_name = models.CharField(max_length=255)

    # Condition
    condition = models.JSONField(
        default=dict,
        help_text="Condition to allow transition"
    )

    # Actions
    on_transition_actions = models.JSONField(default=list)

    # UI
    button_label = models.CharField(max_length=100)
    button_color = models.CharField(max_length=20, default='primary')
    confirmation_message = models.TextField(blank=True)

    class Meta:
        unique_together = [['workflow', 'transition_code']]

class WorkflowAction(models.Model):
    """
    Action definition
    """
    ACTION_TYPES = [
```

```
        ('SEND_EMAIL', 'Send Email'),
        ('SEND_NOTIFICATION', 'Send Notification'),
        ('UPDATE_FIELD', 'Update Field'),
        ('CREATE_RECORD', 'Create Record'),
        ('CALL_API', 'Call External API'),
        ('RUN_SCRIPT', 'Run Custom Script'),
        ('ASSIGN_USER', 'Assign to User'),
        ('SEND_SMS', 'Send SMS'),
    ]

    workflow = models.ForeignKey(
        Workflow,
        on_delete=models.CASCADE,
        related_name='action_definitions'
    )

    action_code = models.CharField(max_length=50)
    action_name = models.CharField(max_length=255)
    action_type = models.CharField(max_length=50, choices=ACTION_TYPES)

    # Configuration
    config = models.JSONField(
        default=dict,
        help_text="Action-specific configuration"
    )

    # Error handling
    on_error = models.CharField(
        max_length=20,
        choices=[
            ('CONTINUE', 'Continue Workflow'),
            ('STOP', 'Stop Workflow'),
            ('RETRY', 'Retry Action'),
        ],
        default='STOP'
    )
    retry_count = models.IntegerField(default=0)

    order = models.IntegerField(default=0)

    class Meta:
        ordering = ['workflow', 'order']
        unique_together = [['workflow', 'action_code']]
```

5.2 Workflow Execution

```
# backend/apps/workflows/models/execution.py
from django.db import models

class WorkflowInstance(models.Model):
    """
    Running instance of a workflow
    """
    workflow = models.ForeignKey(
        'Workflow',
```

```
        on_delete=models.PROTECT
    )
    company = models.ForeignKey(
        'companies.Company',
        on_delete=models.PROTECT
    )

    # Target record
    target_model = models.CharField(max_length=100)
    target_id = models.IntegerField()

    # Current state
    current_state = models.ForeignKey(
        'WorkflowState',
        on_delete=models.PROTECT,
        related_name='active_instances'
    )

    # Status
    status = models.CharField(
        max_length=20,
        choices=[
            ('RUNNING', 'Running'),
            ('WAITING', 'Waiting for Approval'),
            ('COMPLETED', 'Completed'),
            ('FAILED', 'Failed'),
            ('CANCELLED', 'Cancelled'),
        ],
        default='RUNNING'
    )

    # Data context
    context_data = models.JSONField(
        default=dict,
        help_text="Workflow variables and data"
    )

    # Timestamps
    started_at = models.DateTimeField(auto_now_add=True)
    completed_at = models.DateTimeField(null=True, blank=True)

    # Initiator
    started_by = models.ForeignKey(
        'users.User',
        on_delete=models.PROTECT,
        related_name='started_workflows'
    )

    class Meta:
        ordering = ['-started_at']
        indexes = [
            models.Index(fields=['company', 'workflow', 'status']),
            models.Index(fields=['target_model', 'target_id']),
        ]

class WorkflowLog(models.Model):
```

```
    """
    Audit log for workflow execution
    """
    instance = models.ForeignKey(
        WorkflowInstance,
        on_delete=models.CASCADE,
        related_name='logs'
    )

    event_type = models.CharField(
        max_length=50,
        choices=[
            ('STATE_ENTER', 'Entered State'),
            ('STATE_EXIT', 'Exited State'),
            ('TRANSITION', 'Transition Executed'),
            ('ACTION_START', 'Action Started'),
            ('ACTION_SUCCESS', 'Action Succeeded'),
            ('ACTION_FAILED', 'Action Failed'),
            ('APPROVAL_REQUESTED', 'Approval Requested'),
            ('APPROVED', 'Approved'),
            ('REJECTED', 'Rejected'),
            ('ERROR', 'Error Occurred'),
        ]
    )

    message = models.TextField()
    details = models.JSONField(null=True, blank=True)

    actor = models.ForeignKey(
        'users.User',
        on_delete=models.SET_NULL,
        null=True,
        blank=True
    )

    timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['timestamp']
```

6. Dynamic Form Rendering Service

```
# backend/apps/form_builder/services/renderer.py
from django.apps import apps
from apps.form_builder.models import Form, FormField

class FormRenderer:
    """
    Dynamically render forms based on definition
    """

    def __init__(self, form):
        self.form = form
```

```
    def render_json(self):
        """
        Generate JSON representation for frontend
        """
        return {
            'form_id': self.form.id,
            'name': self.form.name,
            'code': self.form.code,
            'category': self.form.category,
            'sections': self._build_sections(),
            'validation_rules': self.form.validation_rules,
            'calculated_fields': self.form.calculated_fields,
        }

    def _build_sections(self):
        """Build form sections with fields"""
        sections = []

        for section_def in self.form.layout.get('sections', []):
            section = {
                'title': section_def.get('title'),
                'description': section_def.get('description'),
                'collapsible': section_def.get('collapsible', False),
                'fields': []
            }

            # Get fields for this section
            field_names = section_def.get('fields', [])
            for field_name in field_names:
                field = self.form.fields.filter(field_name=field_name).first()
                if field:
                    section['fields'].append(self._render_field(field))

            sections.append(section)

        return sections

    def _render_field(self, field):
        """Render individual field"""
        field_def = {
            'name': field.field_name,
            'type': field.field_type,
            'label': field.label,
            'placeholder': field.placeholder,
            'help_text': field.help_text,
            'required': field.is_required,
            'validation': self._build_validation(field),
            'width': field.width,
        }

        # Add type-specific properties
        if field.field_type in ['select', 'radio', 'multiselect']:
            field_def['options'] = field.options

        if field.field_type == 'lookup':
            field_def['lookup'] = {
```

```
                'model': field.lookup_model,
                'field': field.lookup_field,
                'filters': field.lookup_filters,
            }

        if field.field_type == 'calculated':
            field_def['formula'] = field.formula

        if field.visibility_condition:
            field_def['visibility_condition'] = field.visibility_condition

        return field_def

    def _build_validation(self, field):
        """Build validation rules"""
        validation = {
            'required': field.is_required,
            'unique': field.is_unique,
        }

        if field.min_length:
            validation['min_length'] = field.min_length
        if field.max_length:
            validation['max_length'] = field.max_length
        if field.min_value:
            validation['min_value'] = float(field.min_value)
        if field.max_value:
            validation['max_value'] = float(field.max_value)
        if field.regex_pattern:
            validation['pattern'] = field.regex_pattern

        return validation
```

7. Workflow Execution Engine

```
# backend/apps/workflows/services/executor.py
from django.apps import apps
from django.db import transaction
from apps.workflows.models import (
    Workflow, WorkflowInstance, WorkflowLog
)

class WorkflowExecutor:
    """
    Execute workflow logic
    """

    def __init__(self, workflow):
        self.workflow = workflow

    def start_instance(self, target_model, target_id, user, context=None):
        """
        Start new workflow instance
        """
```

```
        # Get start state
        start_state = self.workflow.state_definitions.filter(
            state_type='START'
        ).first()

        if not start_state:
            raise ValueError("Workflow has no start state")

        # Create instance
        instance = WorkflowInstance.objects.create(
            workflow=self.workflow,
            company=user.company,
            target_model=target_model,
            target_id=target_id,
            current_state=start_state,
            status='RUNNING',
            context_data=context or {},
            started_by=user
        )

        # Log start
        self._log_event(
            instance,
            'STATE_ENTER',
            f"Workflow started in state: {start_state.state_name}",
            user
        )

        # Execute on_enter actions
        self._execute_actions(instance, start_state.on_enter_actions, user)

        return instance

    def execute_transition(self, instance, transition_code, user):
        """
        Execute state transition
        """
        # Find transition
        transition = self.workflow.transition_definitions.filter(
            transition_code=transition_code,
            from_state=instance.current_state
        ).first()

        if not transition:
            raise ValueError(f"Invalid transition: {transition_code}")

        # Check condition
        if not self._evaluate_condition(instance, transition.condition):
            raise ValueError("Transition condition not met")

        with transaction.atomic():
            # Execute on_exit actions of current state
            self._execute_actions(
                instance,
                instance.current_state.on_exit_actions,
                user
```

```
            )

            # Execute transition actions
            self._execute_actions(
                instance,
                transition.on_transition_actions,
                user
            )

            # Change state
            old_state = instance.current_state
            instance.current_state = transition.to_state

            # Update status
            if transition.to_state.state_type == 'END':
                instance.status = 'COMPLETED'
                instance.completed_at = timezone.now()
            elif transition.to_state.requires_approval:
                instance.status = 'WAITING'

            instance.save()

            # Log transition
            self._log_event(
                instance,
                'TRANSITION',
                f"Transitioned from {old_state.state_name} to {transition.to_state.state_
                user
            )

            # Execute on_enter actions of new state
            self._execute_actions(
                instance,
                transition.to_state.on_enter_actions,
                user
            )

        return instance

    def _execute_actions(self, instance, actions, user):
        """Execute list of actions"""
        for action_def in actions:
            action = self.workflow.action_definitions.filter(
                action_code=action_def['action_code']
            ).first()

            if not action:
                continue

            try:
                self._log_event(
                    instance,
                    'ACTION_START',
                    f"Executing action: {action.action_name}",
                    user
                )
```

```
                self._execute_single_action(instance, action, user)

                self._log_event(
                    instance,
                    'ACTION_SUCCESS',
                    f"Action completed: {action.action_name}",
                    user
                )
            except Exception as e:
                self._log_event(
                    instance,
                    'ACTION_FAILED',
                    f"Action failed: {action.action_name} - {str(e)}",
                    user
                )

                if action.on_error == 'STOP':
                    instance.status = 'FAILED'
                    instance.save()
                    raise

    def _execute_single_action(self, instance, action, user):
        """Execute individual action"""
        if action.action_type == 'SEND_EMAIL':
            self._send_email(instance, action.config)

        elif action.action_type == 'SEND_NOTIFICATION':
            self._send_notification(instance, action.config)

        elif action.action_type == 'UPDATE_FIELD':
            self._update_field(instance, action.config)

        elif action.action_type == 'CREATE_RECORD':
            self._create_record(instance, action.config)

        elif action.action_type == 'ASSIGN_USER':
            self._assign_user(instance, action.config)

        # Add more action types as needed

    def _evaluate_condition(self, instance, condition):
        """Evaluate workflow condition"""
        if not condition:
            return True

        # Simple condition evaluation
        # Production: Use safe expression evaluator
        operator = condition.get('operator')
        field = condition.get('field')
        value = condition.get('value')

        current_value = instance.context_data.get(field)

        if operator == 'equals':
            return current_value == value
```

```
        elif operator == 'not_equals':
            return current_value != value
        elif operator == 'greater_than':
            return float(current_value) &gt; float(value)
        elif operator == 'less_than':
            return float(current_value) &lt; float(value)

        return True

    def _log_event(self, instance, event_type, message, user):
        """Log workflow event"""
        WorkflowLog.objects.create(
            instance=instance,
            event_type=event_type,
            message=message,
            actor=user
        )

    def _send_email(self, instance, config):
        """Send email action"""
        # Implementation for sending email
        pass

    def _send_notification(self, instance, config):
        """Send notification action"""
        # Implementation for in-app notification
        pass

    def _update_field(self, instance, config):
        """Update field action"""
        # Implementation for updating target record field
        pass

    def _create_record(self, instance, config):
        """Create new record action"""
        # Implementation for creating related record
        pass

    def _assign_user(self, instance, config):
        """Assign to user action"""
        instance.current_approver_id = config.get('user_id')
        instance.save()
```

8. Implementation Checklist

Weeks 1-3: Form Builder Foundation

- ☐ Create Form and FormField models
- ☐ Build FormSubmission for data storage
- ☐ Implement FormRenderer service
- ☐ Create form builder API endpoints

- ☐ Write unit tests

Weeks 4-5: Module Builder

- ☐ Create CustomModule models
- ☐ Implement dynamic model generator
- ☐ Build module API generator
- ☐ Create module publishing system
- ☐ Write tests

Weeks 6-8: Workflow Engine

- ☐ Create Workflow models (states, transitions, actions)
- ☐ Build WorkflowExecutor service
- ☐ Implement action handlers
- ☐ Create workflow monitoring UI
- ☐ Write integration tests

Weeks 9-10: UI & Testing

- ☐ Build drag-and-drop form designer UI
- ☐ Create workflow visual editor
- ☐ Implement real-time preview
- ☐ End-to-end testing
- ☐ Performance optimization

Document Control:

- **Version:** 1.0
- **Dependencies:** Phase 0-3 complete
- **Next Phase:** Phase 5 - AI Companion