cafeMuji 注文管理システム テスト計画書

🚀 概要

cafeMuji注文管理システムの品質保証のための包括的なテスト計画書です。 単体テストから統合テスト、ユーザビリティテストまで、全工程でのテスト戦略を定義します。

📋 テスト戦略

テストの目的

- **品質保証**: システムの動作保証
- **バグ発見**: 開発段階での問題早期発見
- **リグレッション防止**: 既存機能の破綻防止
- **ユーザビリティ向上**: ユーザー体験の改善

テストレベル

1. **単体テスト**: 個別の関数・クラスのテスト
2. **統合テスト**: コンポーネント間の連携テスト
3. **システムテスト**: システム全体の動作テスト
4. **ユーザビリティテスト**: ユーザー体験のテスト

🧪 単体テスト

1. モデルテスト

1.1 FoodOrderモデルテスト

```
# tests/test_models.py
from django.test import TestCase
from django.utils import timezone
from food.models import FoodOrder

class FoodOrderModelTest(TestCase):
    def setUp(self):
        """テストデータの準備"""
        self.food_order = FoodOrder.objects.create(
            menu='からあげ丼',
            quantity=2,
            eat_in=True,
            clip_color='yellow',
            clip_number=1,
            group_id='test_group_001'
        )

    def test_food_order_creation(self):
        """注文作成のテスト"""
```

```
        self.assertEqual(self.food_order.menu, 'からあげ丼')
        self.assertEqual(self.food_order.quantity, 2)
        self.assertTrue(self.food_order.eat_in)
        self.assertEqual(self.food_order.status, 'ok')
        self.assertFalse(self.food_order.is_completed)

    def test_food_order_string_representation(self):
        """文字列表現のテスト"""
        expected = f"{self.food_order.menu} ×{self.food_order.quantity}"
        self.assertEqual(str(self.food_order), expected)

    def test_food_order_validation(self):
        """バリデーションのテスト"""
        # 数量が0以下の場合
        with self.assertRaises(ValueError):
            FoodOrder.objects.create(
                menu='からあげ丼',
                quantity=0,
                eat_in=True,
                clip_color='yellow',
                clip_number=1,
                group_id='test_group_001'
            )

    def test_food_order_completion(self):
        """注文完了のテスト"""
        self.food_order.is_completed = True
        self.food_order.completed_at = timezone.now()
        self.food_order.save()

        self.assertTrue(self.food_order.is_completed)
        self.assertIsNotNone(self.food_order.completed_at)
```

1.2 Orderモデルテスト（アイスクリーム）

```
# tests/test_ice_models.py
from django.test import TestCase
from ice.models import Order

class OrderModelTest(TestCase):
    def setUp(self):
        """テストデータの準備"""
        self.order = Order.objects.create(
            group_id='test_group_001',
            size='S',
            container='cup',
            flavor1='jersey',
            clip_color='white',
            clip_number=2
        )

    def test_order_creation(self):
        """注文作成のテスト"""
        self.assertEqual(self.order.size, 'S')
        self.assertEqual(self.order.container, 'cup')
        self.assertEqual(self.order.flavor1, 'jersey')
        self.assertIsNone(self.order.flavor2)

    def test_double_flavor_order(self):
        """ダブルフレーバー注文のテスト"""
        double_order = Order.objects.create(
            group_id='test_group_002',
            size='W',
            container='cone',
            flavor1='jersey',
            flavor2='mango',
            clip_color='yellow',
            clip_number=3
        )

        self.assertEqual(double_order.size, 'W')
        self.assertEqual(double_order.flavor2, 'mango')
```

2. ビューテスト

2.1 フード注文ビューテスト

```
# tests/test_food_views.py
from django.test import TestCase, Client
from django.urls import reverse
from food.models import FoodOrder

class FoodViewsTest(TestCase):
    def setUp(self):
        """テストクライアントとデータの準備"""
        self.client = Client()
        self.food_order = FoodOrder.objects.create(
            menu='からあげ丼',
            quantity=1,
            eat_in=True,
            clip_color='yellow',
            clip_number=1,
            group_id='test_group_001'
        )

    def test_food_register_view(self):
        """注文登録画面の表示テスト"""
        response = self.client.get(reverse('food_register'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'food/food_register.html')

    def test_add_temp_food_view(self):
        """仮注文追加のテスト"""
        response = self.client.post(reverse('add_temp_food'), {
            'menu': 'ルーロー飯',
            'quantity': '2',
            'eat_in': '1'
        })

        self.assertEqual(response.status_code, 302)  # リダイレクト
        # セッションに仮注文が追加されているか確認
        session = self.client.session
        self.assertIn('temp_food', session)
        temp_food = session['temp_food']
        self.assertEqual(len(temp_food), 1)
        self.assertEqual(temp_food[0]['menu'], 'ルーロー飯')

    def test_food_kitchen_view(self):
        """キッチン画面の表示テスト"""
        response = self.client.get(reverse('food_kitchen'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'food/food_kitchen.html')
        self.assertContains(response, 'からあげ丼')
```

2.2 アイスクリーム注文ビューテスト

```
# tests/test_ice_views.py
from django.test import TestCase, Client
from django.urls import reverse
from ice.models import Order

class IceViewsTest(TestCase):
    def setUp(self):
        """テストクライアントとデータの準備"""
        self.client = Client()
        self.order = Order.objects.create(
            group_id='test_group_001',
            size='S',
            container='cup',
            flavor1='jersey',
            clip_color='white',
            clip_number=1
        )
```

```
    def test_ice_view(self):
        """アイスクリーム注文画面の表示テスト"""
        response = self.client.get(reverse('ice'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'ice/ice.html')

    def test_ice_register_view(self):
        """アイスクリーム注文登録のテスト"""
        response = self.client.post(reverse('ice_register'), {
            'size': 'W',
            'container': 'cone',
            'flavor1': 'jersey',
            'flavor2': 'mango',
            'clip_color': 'yellow',
            'clip_number': '2'
        })

        self.assertEqual(response.status_code, 200)
        # 注文が作成されているか確認
        orders = Order.objects.filter(group_id='test_group_001')
        self.assertEqual(orders.count(), 2)
```

3. フォームテスト

3.1 注文フォームテスト

```
# tests/test_forms.py
from django.test import TestCase
from food.forms import FoodOrderForm

class FoodOrderFormTest(TestCase):
    def test_valid_food_order_form(self):
        """有効な注文フォームのテスト"""
        form_data = {
            'menu': 'からあげ丼',
            'quantity': 2,
            'eat_in': True,
            'clip_color': 'yellow',
            'clip_number': 1,
            'group_id': 'test_group_001'
        }
        form = FoodOrderForm(data=form_data)
        self.assertTrue(form.is_valid())

    def test_invalid_food_order_form(self):
        """無効な注文フォームのテスト"""
        # 数量が0の場合
        form_data = {
            'menu': 'からあげ丼',
            'quantity': 0,
            'eat_in': True,
            'clip_color': 'yellow',
            'clip_number': 1,
            'group_id': 'test_group_001'
        }
        form = FoodOrderForm(data=form_data)
        self.assertFalse(form.is_valid())
        self.assertIn('quantity', form.errors)
```

🔗 統合テスト

1. 注文フローテスト

1.1 完全な注文フロー

```
# tests/test_integration.py
from django.test import TestCase, Client
from django.urls import reverse
from food.models import FoodOrder

class OrderFlowIntegrationTest(TestCase):
    def setUp(self):
        """テストクライアントの準備"""
        self.client = Client()

    def test_complete_order_flow(self):
        """完全な注文フローのテスト"""
        # 1. 注文登録画面にアクセス
        response = self.client.get(reverse('food_register'))
        self.assertEqual(response.status_code, 200)

        # 2. 仮注文を追加
        response = self.client.post(reverse('add_temp_food'), {
            'menu': 'からあげ丼',
            'quantity': '2',
            'eat_in': '1'
        })
        self.assertEqual(response.status_code, 302)

        # 3. 仮注文がセッションに保存されているか確認
        session = self.client.session
        self.assertIn('temp_food', session)
        temp_food = session['temp_food']
        self.assertEqual(len(temp_food), 1)

        # 4. 本注文を確定
        response = self.client.post(reverse('confirm_food'), {
            'clip_color': 'yellow',
            'clip_number': '1'
        })
        self.assertEqual(response.status_code, 302)

        # 5. データベースに注文が保存されているか確認
        orders = FoodOrder.objects.all()
        self.assertEqual(orders.count(), 1)
        order = orders.first()
        self.assertEqual(order.menu, 'からあげ丼')
        self.assertEqual(order.quantity, 2)
        self.assertTrue(order.eat_in)

        # 6. キッチン画面で注文が表示されるか確認
        response = self.client.get(reverse('food_kitchen'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'からあげ丼')

        # 7. 注文を完了
        response = self.client.post(reverse('complete_food'), {
            'order_id': order.id
        })
        self.assertEqual(response.status_code, 302)

        # 8. 完了状態が更新されているか確認
        order.refresh_from_db()
        self.assertTrue(order.is_completed)
        self.assertIsNotNone(order.completed_at)
```

2. セッション管理テスト

2.1 セッションの永続性

```
# tests/test_session.py
from django.test import TestCase, Client
from django.urls import reverse

class SessionManagementTest(TestCase):
    def setUp(self):
```

```
        """テストクライアントの準備"""
        self.client = Client()

    def test_session_persistence(self):
        """セッションの永続性テスト"""
        # 1. 仮注文を追加
        response = self.client.post(reverse('add_temp_food'), {
            'menu': 'からあげ丼',
            'quantity': '1',
            'eat_in': '1'
        })

        # 2. 別の画面に移動
        response = self.client.get(reverse('food_kitchen'))

        # 3. 注文登録画面に戻る
        response = self.client.get(reverse('food_register'))

        # 4. セッションに仮注文が保持されているか確認
        session = self.client.session
        self.assertIn('temp_food', session)
        temp_food = session['temp_food']
        self.assertEqual(len(temp_food), 1)
        self.assertEqual(temp_food[0]['menu'], 'からあげ丼')
```

🖥️ システムテスト

1. エンドツーエンドテスト

1.1 全機能の動作確認

```
# tests/test_system.py
from django.test import TestCase, Client
from django.urls import reverse
from food.models import FoodOrder
from ice.models import Order
from shavedice.models import ShavedIceOrder

class SystemIntegrationTest(TestCase):
    def setUp(self):
        """テストクライアントの準備"""
        self.client = Client()

    def test_full_system_workflow(self):
        """全システムのワークフローテスト"""
        # 1. フード注文
        self.client.post(reverse('add_temp_food'), {
            'menu': 'からあげ丼',
            'quantity': '1',
            'eat_in': '1'
        })

        self.client.post(reverse('confirm_food'), {
            'clip_color': 'yellow',
            'clip_number': '1'
        })

        # 2. アイスクリーム注文
        self.client.post(reverse('ice_register'), {
            'size': 'S',
            'container': 'cup',
            'flavor1': 'jersey',
            'clip_color': 'white',
            'clip_number': '2'
        })

        # 3. かき氷注文
        self.client.post(reverse('shavedice_register'), {
            'flavor': '抹茶',
```

```
            'clip_color': 'yellow',
            'clip_number': '3'
        })

        # 4. 各注文がデータベースに保存されているか確認
        food_orders = FoodOrder.objects.all()
        ice_orders = Order.objects.all()
        shavedice_orders = ShavedIceOrder.objects.all()

        self.assertEqual(food_orders.count(), 1)
        self.assertEqual(ice_orders.count(), 1)
        self.assertEqual(shavedice_orders.count(), 1)

        # 5. キッチン画面で全注文が表示されるか確認
        response = self.client.get(reverse('food_kitchen'))
        self.assertContains(response, 'からあげ丼')

        response = self.client.get(reverse('ice_kitchen'))
        self.assertContains(response, 'jersey')

        response = self.client.get(reverse('shavedice_kitchen'))
        self.assertContains(response, '抹茶')
```

2. パフォーマンステスト

2.1 大量データ処理

```
# tests/test_performance.py
from django.test import TestCase, Client
from django.urls import reverse
from django.utils import timezone
from food.models import FoodOrder
import time

class PerformanceTest(TestCase):
    def setUp(self):
        """テストデータの準備"""
        self.client = Client()

        # 大量のテストデータを作成
        for i in range(100):
            FoodOrder.objects.create(
                menu='からあげ丼' if i % 2 == 0 else 'ルーロー飯',
                quantity=1,
                eat_in=True,
                clip_color='yellow' if i % 2 == 0 else 'white',
                clip_number=i + 1,
                group_id=f'test_group_{i:03d}',
                timestamp=timezone.now()
            )

    def test_large_data_loading(self):
        """大量データの読み込みテスト"""
        start_time = time.time()

        response = self.client.get(reverse('food_kitchen'))

        end_time = time.time()
        load_time = end_time - start_time

        self.assertEqual(response.status_code, 200)
        self.assertLess(load_time, 1.0)  # 1秒以内で読み込み完了

        # 注文数が正しく表示されているか確認
        self.assertContains(response, '100')
```

👥 ユーザビリティテスト

1. 画面表示テスト

1.1 レスポンシブデザイン

```
# tests/test_usability.py
from django.test import TestCase, Client
from django.urls import reverse

class UsabilityTest(TestCase):
    def setUp(self):
        """テストクライアントの準備"""
        self.client = Client()

    def test_responsive_design(self):
        """レスポンシブデザインのテスト"""
        # デスクトップサイズでの表示
        response = self.client.get(reverse('food_register'))
        self.assertEqual(response.status_code, 200)

        # モバイルサイズでの表示
        response = self.client.get(reverse('mobile_order'))
        self.assertEqual(response.status_code, 200)

    def test_accessible_navigation(self):
        """アクセシブルなナビゲーションのテスト"""
        # 各画面へのナビゲーションが可能か確認
        urls = [
            'food_register',
            'food_kitchen',
            'ice',
            'shavedice',
            'mobile_order'
        ]

        for url_name in urls:
            try:
                response = self.client.get(reverse(url_name))
                self.assertEqual(response.status_code, 200)
            except:
                # 一部のURLは認証が必要な場合がある
                pass
```

2. エラーハンドリングテスト

2.1 不正な入力の処理

```
# tests/test_error_handling.py
from django.test import TestCase, Client
from django.urls import reverse

class ErrorHandlingTest(TestCase):
    def setUp(self):
        """テストクライアントの準備"""
        self.client = Client()

    def test_invalid_input_handling(self):
        """不正な入力の処理テスト"""
        # 数量が0の場合
        response = self.client.post(reverse('add_temp_food'), {
            'menu': 'からあげ丼',
            'quantity': '0',
            'eat_in': '1'
        })

        # エラーハンドリングが適切に行われているか確認
        self.assertEqual(response.status_code, 302)  # リダイレクト

        # セッションに不正なデータが保存されていないか確認
```

```
        session = self.client.session
        if 'temp_food' in session:
            temp_food = session['temp_food']
            for item in temp_food:
                self.assertGreater(item['quantity'], 0)

    def test_missing_required_fields(self):
        """必須フィールドの欠損テスト"""
        # メニューが指定されていない場合
        response = self.client.post(reverse('add_temp_food'), {
            'quantity': '1',
            'eat_in': '1'
        })

        self.assertEqual(response.status_code, 302)
```

🔧 テスト環境の設定

1. テスト設定

1.1 Django設定

```
# settings_test.py
from .settings import *

# テスト用データベース設定
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': ':memory:',
    }
}

# テスト用の設定
DEBUG = False
TEMPLATES[0]['OPTIONS']['debug'] = False

# テスト用のセッション設定
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
```

1.2 テストデータベース

```
# tests/conftest.py
import pytest
from django.conf import settings

@pytest.fixture(scope='session')
def django_db_setup(django_db_setup, django_db_blocker):
    with django_db_blocker.unblock():
        # テストデータベースの初期化
        from django.core.management import call_command
        call_command('migrate')
```

2. テスト実行

2.1 基本的なテスト実行

```
# 全テストの実行
python manage.py test

# 特定アプリのテスト
python manage.py test food
python manage.py test ice
```

```
python manage.py test shavedice

# 特定のテストクラスの実行
python manage.py test tests.test_models.FoodOrderModelTest

# 特定のテストメソッドの実行
python manage.py test tests.test_models.FoodOrderModelTest.test_food_order_creation
```

2.2 カバレッジ付きテスト

```
# カバレッジのインストール
pip install coverage

# カバレッジ付きテスト実行
coverage run --source='.' manage.py test

# カバレッジレポートの表示
coverage report

# HTMLレポートの生成
coverage html
```

📊 テストメトリクス

1. カバレッジ目標

- **コードカバレッジ**: 90%以上
- **ブランチカバレッジ**: 85%以上
- **関数カバレッジ**: 95%以上

2. 品質メトリクス

- **バグ密度**: 1000行あたり1件以下
- **テスト実行時間**: 全テスト5分以内
- **テスト成功率**: 99%以上

🚀 継続的テスト

1. CI/CD統合

1.1 GitHub Actions設定

```
# .github/workflows/test.yml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
```

```
        with:
          python-version: 3.11

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install coverage

      - name: Run tests
        run: |
          coverage run --source='.' manage.py test

      - name: Generate coverage report
        run: |
          coverage report
          coverage html

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v1
```

2. 自動テスト実行

2.1 事前コミットフック

```
#!/bin/bash
# .git/hooks/pre-commit

echo "Running tests before commit..."

# テストの実行
python manage.py test --keepdb

if [ $? -ne 0 ]; then
    echo "Tests failed. Commit aborted."
    exit 1
fi

echo "All tests passed. Proceeding with commit."
```

📝 テストドキュメント

1. テストケース管理

1.1 テストケーステンプレート

```
## テストケース: TC-001

### 概要
フード注文の作成機能のテスト

### 前提条件
- ユーザーがログインしている
- 注文登録画面が表示されている

### テスト手順
1. メニューを選択する
2. 数量を入力する
3. 店内/テイクアウトを選択する
4. 追加ボタンをクリックする

### 期待結果
- 仮注文がセッションに保存される
- 仮注文一覧に選択した商品が表示される
- エラーメッセージが表示されない
```

実際の結果
[テスト実行後に記入]

ステータス
- [ ] 成功
- [ ] 失敗
- [ ] ブロック

備考
[特記事項があれば記入]

2. テスト結果レポート

2.1 テスト実行レポート

```
# テスト結果の詳細レポート
python manage.py test --verbosity=2

# XML形式でのレポート出力
python manage.py test --output=test-results.xml
```

作成日: 2025年8月 **作成者**: 村岡 優次郎 **バージョン**: 1.0