💻 Complete Code Explanation - Part 5

EmployeeDetail Component (Add/Edit/Delete)

---

15. src/pages/EmployeeDetail.tsx

Purpose: Single component that handles THREE modes:

1. **Add Mode** - Create new employee
2. **Edit Mode** - Modify existing employee
3. **Delete Mode** - Soft delete employee

This is the MOST COMPLEX component in the entire application!

```typescript
```

```tsx
import { useState, useEffect, FormEvent } from 'react';
import { useNavigate, useParams } from 'react-router-dom';
import {
  Container,
  Paper,
  Box,
  TextField,
  Button,
  Typography,
  Alert,
  FormControl,
  FormLabel,
  RadioGroup,
  FormControlLabel,
  Radio,
  Checkbox,
  Dialog,
  DialogTitle,
  DialogContent,
  DialogContentText,
  DialogActions,
} from '@mui/material';
import Layout from '../components/Layout';
import { employeeAPI } from '../services/api';
import type { Employee } from '../types';

function EmployeeDetail() {
  const { id } = useParams<{ id: string }>();
  const navigate = useNavigate();

  // Determine mode
  const isNewEmployee = id === 'new';
```

```
const employeeId = isNewEmployee ? null : id;

// Form state
const [formData, setFormData] = useState({
  employeeId: '',
  name: '',
  kanaName: '',
  sex: 1,
  postCode: '',
  address: '',
  phoneNumber: '',
  department: '',
  retireFlg: false,
});

// UI state
const [loading, setLoading] = useState(false);
const [error, setError] = useState<string | null>(null);
const [confirmDialog, setConfirmDialog] = useState({
  open: false,
  type: '' as 'create' | 'update' | 'delete',
});

// Load employee data in edit mode
useEffect(() => {
  if (!isNewEmployee && employeeId) {
    loadEmployeeData(employeeId);
  }
}, [isNewEmployee, employeeId]);

const loadEmployeeData = async (id: string) => {
  setLoading(true);
```

```
  try {
    const response = await employeeAPI.getById(id);
    const emp = response.data.data;

    setFormData({
      employeeId: emp.EmployeeId,
      name: emp.Name,
      kanaName: emp.KanaName || '',
      sex: emp.Sex,
      postCode: emp.PostCode || '',
      address: emp.Address || '',
      phoneNumber: emp.PhoneNumber || '',
      department: emp.Department || '',
      retireFlg: emp.RetireFlg,
    });
  } catch (err: any) {
    setError('従業員データの読み込みに失敗しました');
  } finally {
    setLoading(false);
  }
};

const handleInputChange = (field: string, value: any) => {
  setFormData(prev => ({
    ...prev,
    [field]: value,
  }));
};

const validateForm = (): boolean => {
  if (!formData.employeeId || formData.employeeId.length !== 5) {
    setError('社員番号は5桁で入力してください');
```

```
    return false;
  }

  if (!formData.name) {
    setError('氏名を入力してください');
    return false;
  }

  return true;
};

const handleSubmit = async (e: FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  setError(null);

  if (!validateForm()) {
    return;
  }

  // Show confirmation dialog
  setConfirmDialog({
    open: true,
    type: isNewEmployee ? 'create' : 'update',
  });
};

const handleConfirm = async () => {
  setConfirmDialog({ open: false, type: '' });
  setLoading(true);

  try {
    const employeeData: Partial<Employee> = {
```

```
      EmployeeId: formData.employeeId,
      Name: formData.name,
      KanaName: formData.kanaName,
      Sex: formData.sex,
      PostCode: formData.postCode,
      Address: formData.address,
      PhoneNumber: formData.phoneNumber,
      Department: formData.department,
      RetireFlg: formData.retireFlg,
    };

    if (isNewEmployee) {
      await employeeAPI.create(employeeData as Omit<Employee, 'id'>);
    } else {
      await employeeAPI.update(employeeId!, employeeData);
    }

    navigate('/employees');
  } catch (err: any) {
    setError(
      err.response?.data?.message ||
      (isNewEmployee ? '登録に失敗しました' : '更新に失敗しました')
    );
  } finally {
    setLoading(false);
  }
};

const handleDelete = () => {
  setConfirmDialog({
    open: true,
    type: 'delete',
```

```
  });
};

const handleConfirmDelete = async () => {
  setConfirmDialog({ open: false, type: '' });
  setLoading(true);

  try {
    await employeeAPI.delete(employeeId!);
    navigate('/employees');
  } catch (err: any) {
    setError('削除に失敗しました');
  } finally {
    setLoading(false);
  }
};

const handleCancel = () => {
  navigate('/employees');
};

return (
  <Layout>
    <Container maxWidth="md">
      <Paper sx={{ p: 4, mt: 4 }}>
        <Typography variant="h5" component="h1" gutterBottom>
          {isNewEmployee ? '従業員登録' : '従業員編集'}
        </Typography>

        {error && (
          <Alert severity="error" sx={{ mb: 2 }}>
            {error}
```

```
  </Alert>
)}

<Box component="form" onSubmit={handleSubmit} noValidate>
  {/* Employee ID */}
  <TextField
    margin="normal"
    required
    fullWidth
    label="社員番号（5桁）"
    value={formData.employeeId}
    onChange={(e) => handleInputChange('employeeId', e.target.value)}
    disabled={!isNewEmployee}
    inputProps={{ maxLength: 5 }}
  />

  {/* Name */}
  <TextField
    margin="normal"
    required
    fullWidth
    label="氏名"
    value={formData.name}
    onChange={(e) => handleInputChange('name', e.target.value)}
  />

  {/* Kana Name */}
  <TextField
    margin="normal"
    fullWidth
    label="カナ氏名"
    value={formData.kanaName}
```

```
  onChange={(e) => handleInputChange('kanaName', e.target.value)}
/>

{/* Sex */}
<FormControl component="fieldset" margin="normal" required>
  <FormLabel component="legend">性別</FormLabel>
  <RadioGroup
    row
    value={formData.sex}
    onChange={(e) => handleInputChange('sex', parseInt(e.target.value))}
  >
    <FormControlLabel value={1} control={<Radio />} label="男性" />
    <FormControlLabel value={2} control={<Radio />} label="女性" />
  </RadioGroup>
</FormControl>

{/* Post Code */}
<TextField
  margin="normal"
  fullWidth
  label="郵便番号"
  placeholder="100-0001"
  value={formData.postCode}
  onChange={(e) => handleInputChange('postCode', e.target.value)}
/>

{/* Address */}
<TextField
  margin="normal"
  fullWidth
  label="住所"
  value={formData.address}
```

```
  onChange={(e) => handleInputChange('address', e.target.value)}
  multiline
  rows={2}
/>

{/* Phone Number */}
<TextField
  margin="normal"
  fullWidth
  label="電話番号"
  placeholder="090-1234-5678"
  value={formData.phoneNumber}
  onChange={(e) => handleInputChange('phoneNumber', e.target.value)}
/>

{/* Department */}
<TextField
  margin="normal"
  fullWidth
  label="部署"
  value={formData.department}
  onChange={(e) => handleInputChange('department', e.target.value)}
/>

{/* Retire Flag */}
<FormControlLabel
  control={
    <Checkbox
      checked={formData.retireFlg}
      onChange={(e) => handleInputChange('retireFlg', e.target.checked)}
    />
  }
```

```
  label="退職済み"
  sx={{ mt: 2 }}
/>

{/* Action Buttons */}
<Box sx={{ mt: 3, display: 'flex', gap: 2 }}>
  <Button
    type="submit"
    variant="contained"
    disabled={loading}
    fullWidth
  >
    {loading
      ? '処理中...'
      : isNewEmployee
      ? '登録'
      : '更新'}
  </Button>

  <Button
    variant="outlined"
    onClick={handleCancel}
    disabled={loading}
    fullWidth
  >
    キャンセル
  </Button>
</Box>

{/* Delete Button (Edit mode only) */}
{!isNewEmployee && (
  <Box sx={{ mt: 2 }}>
```

```
          <Button
            variant="outlined"
            color="error"
            onClick={handleDelete}
            disabled={loading}
            fullWidth
          >
            削除
          </Button>
        </Box>
      )}
    </Box>
  </Paper>

  {/* Confirmation Dialog */}
  <Dialog
    open={confirmDialog.open}
    onClose={() => setConfirmDialog({ open: false, type: '' })}
  >
    <DialogTitle>
      {confirmDialog.type === 'create' && '登録確認'}
      {confirmDialog.type === 'update' && '更新確認'}
      {confirmDialog.type === 'delete' && '削除確認'}
    </DialogTitle>
    <DialogContent>
      <DialogContentText>
        {confirmDialog.type === 'create' &&
          'この内容で従業員を登録してもよろしいですか？'}
        {confirmDialog.type === 'update' &&
          'この内容で従業員情報を更新してもよろしいですか？'}
        {confirmDialog.type === 'delete' &&
          'この従業員を削除してもよろしいですか？この操作は取り消せません。'}
```

```
            </DialogContentText>
          </DialogContent>
          <DialogActions>
            <Button
              onClick={() => setConfirmDialog({ open: false, type: '' })}
            >
              キャンセル
            </Button>
            <Button
              onClick={
                confirmDialog.type === 'delete'
                  ? handleConfirmDelete
                  : handleConfirm
              }
              color={confirmDialog.type === 'delete' ? 'error' : 'primary'}
              autoFocus
            >
              {confirmDialog.type === 'delete' ? '削除' : 'はい'}
            </Button>
          </DialogActions>
        </Dialog>
      </Container>
    </Layout>
  );
}

export default EmployeeDetail;
```

Detailed Line-by-Line Explanation:

Line 27-29: useParams Hook

```typescript
const { id } = useParams<{ id: string }>();
```

- **useParams**: React Router hook to get URL parameters
- **<{ id: string }>**: TypeScript generic for type safety
- Extracts `id` from URL

URL Examples:

```typescript
// URL: /employees/new
id = "new"

// URL: /employees/42
id = "42"

// URL: /employees/abc123
id = "abc123"
```

Line 31-33: Determine Mode

```typescript
const isNewEmployee = id === 'new';
const employeeId = isNewEmployee ? null : id;
```

- **isNewEmployee**: Boolean flag for add vs edit mode

- **employeeId**: null for new, ID string for edit

Mode Logic:

```typescript
// URL: /employees/new
isNewEmployee = true
employeeId = null
→ ADD MODE

// URL: /employees/42
isNewEmployee = false
employeeId = "42"
→ EDIT MODE
```

Line 35-46: Form State (Object)

```typescript
const [formData, setFormData] = useState({
  employeeId: '',
  name: '',
  kanaName: '',
  sex: 1,
  postCode: '',
  address: '',
  phoneNumber: '',
  department: '',
  retireFlg: false,
});
```

Why single object instead of individual states?

```typescript
// ❌ Individual states (12 useState calls!):
const [employeeId, setEmployeeId] = useState('');
const [name, setName] = useState('');
const [kanaName, setKanaName] = useState('');
// ... 9 more!

// ✅ Single object state (cleaner!):
const [formData, setFormData] = useState({ ... });
```

Benefits of object state:

- Easier to manage related data
- Single source of truth
- Easier to reset form
- Easier to submit (already in object form)

Line 48-52: UI State

```typescript
const [loading, setLoading] = useState(false);
const [error, setError] = useState<string | null>(null);
const [confirmDialog, setConfirmDialog] = useState({
  open: false,
  type: '' as 'create' | 'update' | 'delete',
});
```

- **loading**: Disable buttons during API calls
- **error**: Error message display
- **confirmDialog**: Control confirmation dialog
  - **open**: Show/hide dialog
  - **type**: Which action to confirm (create/update/delete)

Type annotation explained:

```typescript
type: '' as 'create' | 'update' | 'delete'
//   ↑
// Initial value is empty string
// But TypeScript knows it can be one of these three strings
```

Line 54-59: Load Data in Edit Mode

```typescript
useEffect(() => {
  if (!isNewEmployee && employeeId) {
    loadEmployeeData(employeeId);
  }
}, [isNewEmployee, employeeId]);
```

Conditional effect:

- Only runs in EDIT mode (not new)
- Only runs if employeeId exists

- Runs when component mounts or if employeeId changes

Flow in Edit Mode:

```
1. Component mounts with URL /employees/42
   ↓
2. useParams extracts id = "42"
   ↓
3. isNewEmployee = false, employeeId = "42"
   ↓
4. useEffect runs
   ↓
5. Condition: !false && "42" = true
   ↓
6. loadEmployeeData("42") called
   ↓
7. API fetches employee data
   ↓
8. Form pre-filled with employee data
```

Flow in Add Mode:

1. Component mounts with URL /employees/new
   ↓
2. useParams extracts id = "new"
   ↓
3. isNewEmployee = true, employeeId = null
   ↓
4. useEffect runs
   ↓
5. Condition: !true && null = false
   ↓
6. loadEmployeeData NOT called
   ↓
7. Form remains empty (ready for new data)

Line 61-83: loadEmployeeData Function

typescript

```
const loadEmployeeData = async (id: string) => {
  setLoading(true);
  try {
    const response = await employeeAPI.getById(id);
    const emp = response.data.data;

    setFormData({
      employeeId: emp.EmployeeId,
      name: emp.Name,
      kanaName: emp.KanaName || '',
      sex: emp.Sex,
      postCode: emp.PostCode || '',
      address: emp.Address || '',
      phoneNumber: emp.PhoneNumber || '',
      department: emp.Department || '',
      retireFlg: emp.RetireFlg,
    });
  } catch (err: any) {
    setError('従業員データの読み込みに失敗しました');
  } finally {
    setLoading(false);
  }
};
```

Process:

1. Set loading = true
2. Fetch employee by ID
3. Extract employee data from response
4. Update formData state with all fields

5. Handle optional fields with `|| ''`
6. On error: show error message
7. Finally: set loading = false

Why `|| ''` for optional fields?

```typescript
// If backend returns null or undefined:
kanaName: emp.KanaName || ''

// emp.KanaName is null:
kanaName: null || '' → ''

// emp.KanaName is undefined:
kanaName: undefined || '' → ''

// emp.KanaName is "ヤマダ":
kanaName: "ヤマダ" || '' → "ヤマダ"
```

TextFields don't like null/undefined as values!

```typescript
// ❌ Error: uncontrolled to controlled component
<TextField value={null} />

// ✅ Works: always a string
<TextField value={kanaName || ''} />
```

Line 85-90: handleInputChange Function

```typescript
const handleInputChange = (field: string, value: any) => {
  setFormData(prev => ({
    ...prev,
    [field]: value,
  }));
};
```

This is a GENERIC input handler!

How it works:

```typescript
// Called from onChange:
onChange={(e) => handleInputChange('name', e.target.value)}

// Function executes:
setFormData(prev => ({
  ...prev,          // Keep all existing fields
  ['name']: value,   // Update specific field
}))
```

Example:

```typescript
```

```
// Current state:
formData = {
  employeeId: '12345',
  name: '山田',
  sex: 1,
  // ... other fields
}

// User types '太' in name field:
handleInputChange('name', '山田太')

// New state:
formData = {
  employeeId: '12345',
  name: '山田太',     // ← Updated!
  sex: 1,
  // ... other fields
}
```

Why `prev =>`?

```
typescript
```

```
// ❌ Without prev (can lose updates):
setFormData({
  ...formData,  // Uses OLD state
  [field]: value,
});

// ✅ With prev (always uses LATEST state):
setFormData(prev => ({
  ...prev,      // Uses CURRENT state
  [field]: value,
}));
```

React batches updates, so without `prev =>`, rapid changes can be lost!

Line 92-104: validateForm Function

```typescript
const validateForm = (): boolean => {
  if (!formData.employeeId || formData.employeeId.length !== 5) {
    setError('社員番号は5桁で入力してください');
    return false;
  }

  if (!formData.name) {
    setError('氏名を入力してください');
    return false;
  }

  return true;
};
```

Validation Rules:

1. Employee ID must exist AND be exactly 5 digits
2. Name must not be empty
3. Other fields optional (no validation)

Return value:

- **true**: All validations passed
- **false**: Validation failed (error message set)

Line 106-119: handleSubmit Function

typescript

```typescript
const handleSubmit = async (e: FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  setError(null);

  if (!validateForm()) {
    return;
  }

  // Show confirmation dialog
  setConfirmDialog({
    open: true,
    type: isNewEmployee ? 'create' : 'update',
  });
};
```

NOT submitting immediately!

- First: Prevent default
- Second: Clear errors
- Third: Validate
- Fourth: Show confirmation dialog (ask user)
- Actual submission happens in `handleConfirm`

Two-step submission pattern:

```
User clicks Submit
  ↓
handleSubmit
  ├─ Validate
  ├─ Show confirmation dialog
  └─ Wait for user

User clicks "はい" (Yes)
  ↓
handleConfirm
  ├─ Make API call
  └─ Navigate on success
```

Line 121-153: handleConfirm Function (Actual Submission)

```
typescript
```

```
const handleConfirm = async () => {
  setConfirmDialog({ open: false, type: '' });
  setLoading(true);

  try {
    const employeeData: Partial<Employee> = {
      EmployeeId: formData.employeeId,
      Name: formData.name,
      KanaName: formData.kanaName,
      Sex: formData.sex,
      PostCode: formData.postCode,
      Address: formData.address,
      PhoneNumber: formData.phoneNumber,
      Department: formData.department,
      RetireFlg: formData.retireFlg,
    };

    if (isNewEmployee) {
      await employeeAPI.create(employeeData as Omit<Employee, 'id'>);
    } else {
      await employeeAPI.update(employeeId!, employeeData);
    }

    navigate('/employees');
  } catch (err: any) {
    setError(
      err.response?.data?.message ||
      (isNewEmployee ? '登録に失敗しました' : '更新に失敗しました')
    );
  } finally {
    setLoading(false);
```

```
  }
};
```

Process:

1. Close confirmation dialog
2. Set loading = true
3. Transform formData to Employee type
4. **Branch: Add or Edit?**
    - New: Call create API
    - Edit: Call update API
5. On success: Navigate to list
6. On error: Show error message
7. Finally: Set loading = false

Why transform data?

```typescript
```

```
// formData (camelCase):
{
  employeeId: '12345',
  name: '山田太郎',
  kanaName: 'ヤマダタロウ'
}

// Employee type (PascalCase):
{
  EmployeeId: '12345',
  Name: '山田太郎',
  KanaName: 'ヤマダタロウ'
}

// Need to match Employee interface!
```

Type assertion explained:

```
typescript

// Add mode:
employeeData as Omit<Employee, 'id'>
// "Trust me TypeScript, this is Employee without id"

// Edit mode:
employeeData
employeeId!
// "Trust me, employeeId is not null here"
```

Line 155-161: handleDelete Function

```typescript
const handleDelete = () => {
  setConfirmDialog({
    open: true,
    type: 'delete',
  });
};
```

- Shows confirmation dialog with type='delete'
- Similar pattern to create/update

Line 163-178: handleConfirmDelete Function

```typescript
const handleConfirmDelete = async () => {
  setConfirmDialog({ open: false, type: '' });
  setLoading(true);

  try {
    await employeeAPI.delete(employeeId!);
    navigate('/employees');
  } catch (err: any) {
    setError('削除に失敗しました');
  } finally {
    setLoading(false);
  }
};
```

- Close dialog

- Call delete API
- Navigate to list on success
- Show error on failure

Line 180-182: handleCancel Function

```typescript
const handleCancel = () => {
  navigate('/employees');
};
```

- Simply navigate back to list
- No saving, no confirmation needed
- User loses unsaved changes (could add warning)

Line 184-400: JSX Return (UI)

I'll focus on the interesting parts:

Line 192-194: Dynamic Title

```typescript
<Typography variant="h5" component="h1" gutterBottom>
  {isNewEmployee ? '従業員登録' : '従業員編集'}
</Typography>
```

- Add mode: "従業員登録" (Employee Registration)

- Edit mode: "従業員編集" (Employee Edit)

Line 206-213: Employee ID Field

```typescript
<TextField
  margin="normal"
  required
  fullWidth
  label="社員番号（5桁）"
  value={formData.employeeId}
  onChange={(e) => handleInputChange('employeeId', e.target.value)}
  disabled={!isNewEmployee}
  inputProps={{ maxLength: 5 }}
/>
```

Key attributes:

- **disabled={!isNewEmployee}**:
  - Add mode: Enabled (user can type)
  - Edit mode: Disabled (can't change ID)
- **inputProps={{ maxLength: 5 }}**: Browser enforces 5 char limit

Why disable in edit mode?

```typescript
```

```
// Employee ID is like a primary key
// Changing it would break database relationships
// So we disable the field in edit mode

// Add mode:
<TextField disabled={false} /> // Can type

// Edit mode:
<TextField disabled={true} />  // Grayed out, can't change
```

Line 252-262: Sex Radio Group

typescript

```
<FormControl component="fieldset" margin="normal" required>
  <FormLabel component="legend">性别</FormLabel>
  <RadioGroup
    row
    value={formData.sex}
    onChange={(e) => handleInputChange('sex', parseInt(e.target.value))}
  >
    <FormControlLabel value={1} control={<Radio />} label="男性" />
    <FormControlLabel value={2} control={<Radio />} label="女性" />
  </RadioGroup>
</FormControl>
```

Important: parseInt!

typescript

```
onChange={(e) => handleInputChange('sex', parseInt(e.target.value))}
//                                  ↑
//                          Convert string to number!

// Radio inputs return strings:
e.target.value = "1"  // String!

// We need number:
parseInt("1") = 1     // Number!

// Why? Database expects number (1 or 2)
```

Line 297-309: Address TextField (Multiline)

typescript

```
<TextField
  margin="normal"
  fullWidth
  label="住所"
  value={formData.address}
  onChange={(e) => handleInputChange('address', e.target.value)}
  multiline
  rows={2}
/>
```

- **multiline**: Renders as `<textarea>` instead of `<input>`
- **rows={2}**: Initial height (2 lines)
- Good for long text (addresses, comments)

Line 332-346: Retire Flag Checkbox

```typescript
<FormControlLabel
  control={
    <Checkbox
      checked={formData.retireFlg}
      onChange={(e) => handleInputChange('retireFlg', e.target.checked)}
    />
  }
  label="退職済み"
  sx={{ mt: 2 }}
/>
```

- **checked**: Boolean (true/false)
- **e.target.checked**: Boolean (not e.target.value!)
- Label: "Retired"

Line 348-374: Action Buttons

```typescript
```

```
<Box sx={{ mt: 3, display: 'flex', gap: 2 }}>
  <Button
    type="submit"
    variant="contained"
    disabled={loading}
    fullWidth
  >
    {loading
      ? '処理中...'
      : isNewEmployee
      ? '登録'
      : '更新'}
  </Button>

  <Button
    variant="outlined"
    onClick={handleCancel}
    disabled={loading}
    fullWidth
  >
    キャンセル
  </Button>
</Box>
```

Dynamic submit button text:

- Loading: "処理中..." (Processing...)
- Add mode: "登録" (Register)
- Edit mode: "更新" (Update)

Line 376-387: Delete Button (Edit Mode Only)

typescript

```
{!isNewEmployee && (
  <Box sx={{ mt: 2 }}>
    <Button
      variant="outlined"
      color="error"
      onClick={handleDelete}
      disabled={loading}
      fullWidth
    >
      削除
    </Button>
  </Box>
)}
```

- **Conditional rendering**: Only shows in edit mode
- **color="error"**: Red button (danger action)
- Triggers delete confirmation dialog

Line 390-421: Confirmation Dialog

typescript

```
<Dialog
  open={confirmDialog.open}
  onClose={() => setConfirmDialog({ open: false, type: '' })}
>
  <DialogTitle>
    {confirmDialog.type === 'create' && '登録確認'}
    {confirmDialog.type === 'update' && '更新確認'}
    {confirmDialog.type === 'delete' && '削除確認'}
  </DialogTitle>
  <DialogContent>
    <DialogContentText>
      {confirmDialog.type === 'create' &&
       'この内容で従業員を登録してもよろしいですか？'}
      {confirmDialog.type === 'update' &&
       'この内容で従業員情報を更新してもよろしいですか？'}
      {confirmDialog.type === 'delete' &&
       'この従業員を削除してもよろしいですか？この操作は取り消せません。'}
    </DialogContentText>
  </DialogContent>
  <DialogActions>
    <Button
      onClick={() => setConfirmDialog({ open: false, type: '' })}
    >
      キャンセル
    </Button>
    <Button
      onClick={
        confirmDialog.type === 'delete'
          ? handleConfirmDelete
          : handleConfirm
      }
      color={confirmDialog.type === 'delete' ? 'error' : 'primary'}
```

```
    autoFocus
  >
    {confirmDialog.type === 'delete' ? '削除' : 'はい'}
  </Button>
 </DialogActions>
</Dialog>
```

Dynamic dialog content based on type:

Type	Title	Message	Confirm Button
create	登録確認	この内容で従業員を登録してもよろしいですか？	はい (Yes)
update	更新確認	この内容で従業員情報を更新してもよろしいですか？	はい (Yes)
delete	削除確認	この従業員を削除してもよろしいですか？この操作は取り消せません。	削除 (Delete)

Confirm button logic:

```typescript
```

```
onClick={
  confirmDialog.type === 'delete'
    ? handleConfirmDelete
    : handleConfirm
}

// If delete → handleConfirmDelete
// If create or update → handleConfirm
```

Complete Flow Diagrams:

Add Flow:

```
1. Navigate to /employees/new
   ↓
2. Component mounts
   ↓
3. id = "new", isNewEmployee = true
   ↓
4. useEffect runs but doesn't load data
   ↓
5. Form renders empty
   ↓
6. User fills form
   ↓
7. User clicks "登録"
   ↓
8. handleSubmit
   ├─ Validate
   └─ Show confirmation dialog
   ↓
9. User clicks "はい"
   ↓
10. handleConfirm
    ├─ API: POST /sem/employee
    └─ Navigate to /employees
```

Edit Flow:

```
1. Navigate to /employees/42
   ↓
2. Component mounts
   ↓
3. id = "42", isNewEmployee = false
   ↓
4. useEffect runs
   ↓
5. loadEmployeeData(42)
   ├─ API: GET /sem/employee/42
   └─ setFormData with employee data
   ↓
6. Form renders pre-filled
   ↓
7. User modifies fields
   ↓
8. User clicks "更新"
   ↓
9. handleSubmit
   ├─ Validate
   └─ Show confirmation dialog
   ↓
10. User clicks "はい"
    ↓
11. handleConfirm
    ├─ API: PUT /sem/employee/42
    └─ Navigate to /employees
```

Delete Flow:

```
1. In edit mode (/employees/42)
   ↓
2. User clicks "削除" button
   ↓
3. handleDelete
   └─ Show confirmation dialog (type='delete')
   ↓
4. User clicks "削除" (confirm)
   ↓
5. handleConfirmDelete
   ├─ API: DELETE /sem/employee/42
   └─ Navigate to /employees
```

This component is a masterclass in:

- **Mode detection** (add vs edit)
- **Conditional rendering** (different UI based on mode)
- **Form state management** (object state pattern)
- **Validation** (client-side checks)
- **Confirmation dialogs** (user safety)
- **Error handling** (user feedback)

That completes the frontend explanation! Would you like me to continue with the backend ObjectScript files next?