OnBoarding Feature - Simple Guide

What is OnBoarding?

OnBoarding creates a complete company structure in ONE request instead of creating each part separately.

Example: Create "Tech Solutions" company with all its offices, departments, and teams in one go!

The Flow

```
User Request
   ↓
Step 1: Create Organization
Step 2: Create Companies
Step 3: Create Branches
Step 4: Create Departments
Step 5: Create Teams
Step 6: Create Scopes
   ↓
Done!
```

Step 1: Create Organization

What it does: Creates the main organization

Code:

```csharp
var organization = request.Adapt<Organization>();
await _repository.AddAsync(organization, currentUserId);
return organization.Id;
```

Input: Organization name

Output: Organization ID

Example:

- Input: "Tech Solutions"

- Output: "org-12345"

Why return OrganizationId?

- Next step (Companies) needs this ID to link all companies to this organization

---

Step 2: Create Companies

What it does:

1. Creates companies
2. Finds default cities for companies without branches

Code:

csharp

```
// Create companies
var companies = request.companies.Adapt<List<Company>>();
companies.ForEach(c => c.OrganizationId = request.OrganizationId);
await _repository.AddRangeAsync(companies);

// Get default cities only for companies without branches
var countryIds = request.companies
    .Where(c => c.Branches == null || !c.Branches.Any())
    .Select(c => c.CountryId)
    .Distinct()
    .ToList();

var defaultCities = await GetDefaultCities(countryIds);

// Return company data with default city
return new CompaniesResponseDto {
    CompanyId = company.Id,
    CompanyName = company.Name,
    DefaultCityId = defaultCities.GetValueOrDefault(countryId),
    Branches = company.Branches
};
```

Why DefaultCityId?

If a company has no branches, we'll create a default branch. That branch needs a city location!

Step 3: Create Branches

What it does: Creates branches OR creates a default branch if none provided

Code:

```csharp
var branchesDtos = request.Companies
    .SelectMany(company =>
        // Does company have branches?
        (company.Branches != null && company.Branches.Count != 0)
            ? company.Branches  // Yes - use them
            : new[] {           // No - create default
                new BranchesDto(
                    $"{company.CompanyName} Branch",  // Name
                    null,                             // Phone
                    company.DefaultCityId,            // City from step 2
                    company.CompanyId,
                    new List<DepartmentsDto>()
                )
            }
    ).ToList();

var branches = branchesDtos.Adapt<List<Branch>>();
await _repository.AddRangeAsync(branches);

// Build response
return branches.Select((branch, index) => new BranchesResponseDto {
    BranchId = branch.Id,
    BranchName = branch.Name,
    CompanyId = branch.CompanyId,
    Departments = branchesDtos[index].Departments
}).ToList();
```

Example:

- Company has branches → Use them
- Company has NO branches → Create "Tech Solutions Branch" automatically

What gets returned:

```csharp
```

```
BranchesResponseDto {
    BranchId = branch.Id,              // New ID generated after saving
    BranchName = branch.Name,          // Branch name
    CompanyId = branch.CompanyId,      // Which company this branch belongs to
    Departments = branchesDtos[index].Departments  // Departments data for next step
}
```

Why return these?

- `BranchId` - The new ID created after saving to database
- `BranchName` - For creating default department name ("BranchName Department")
- `CompanyId` - Needed for scope generation later
- `Departments` - Next step needs this data to create departments

Step 4: Create Departments

What it does: Creates departments OR creates default department

Important: Keeps track of CompanyId (needed for scopes later)

Code:

csharp

```
var departmentsWithScope = request.Branches
    .SelectMany(branch =>
        // Does branch have departments?
        (branch.Departments != null && branch.Departments.Count != 0)
            ? branch.Departments.Select(d => new {
                Department = d with { BranchId = branch.BranchId },
                branch.CompanyId  // Keep CompanyId for later
            })
            : new[] {  // No - create default
                new {
                    Department = new DepartmentsDto(
                        $"{branch.BranchName} Department",
                        "Default department",
                        branch.BranchId,
                        new List<TeamsDto>()
                    ),
                    branch.CompanyId  // Keep CompanyId for later
                }
            }
    ).ToList();

var departments = departmentsWithScope
    .Select(x => x.Department)
    .Adapt<List<Department>>();

await _repository.AddRangeAsync(departments);

// Build response
return departments.Select((dept, index) => new DepartmentsResponseDto {
    DepartmentId = dept.Id,
    DepartmentName = dept.Name,
    BranchId = dept.BranchId,
    CompanyId = departmentsWithScope[index].CompanyId,
    Teams = departmentsWithScope[index].Department.Teams
}).ToList();
```

Why `new { Department, CompanyId }`?

Department doesn't have CompanyId in the database, but we need it for scopes. So we wrap both together temporarily.

What gets returned:

```
csharp
```

```
DepartmentsResponseDto {
    DepartmentId = dept.Id,           // New ID generated after saving
    DepartmentName = dept.Name,       // Department name
    BranchId = dept.BranchId,         // Which branch this department belongs to
    CompanyId = departmentsWithScope[index].CompanyId,  // From wrapped object
    Teams = departmentsWithScope[index].Department.Teams  // Teams data for next step
}
```

Why return these?

- `DepartmentId` - The new ID created after saving to database
- `DepartmentName` - For creating default team name ("DepartmentName Team")
- `BranchId` - Needed for scope generation
- `CompanyId` - Needed for scope generation (passed through from branch)
- `Teams` - Next step needs this data to create teams

Step 5: Create Teams

What it does: Creates teams OR creates default team, then builds complete scope hierarchy

Code:

```csharp
```

```
var teamsWithScopes = request.Departments
    .SelectMany(dept =>
        // Does department have teams?
        (dept.Teams != null && dept.Teams.Count != 0)
            ? dept.Teams.Select(t => new {
                Team = t with { DepartmentId = dept.DepartmentId },
                dept.BranchId,
                dept.CompanyId
            })
            : new[] {  // No - create default
                new {
                    Team = new TeamsDto($"{dept.DepartmentName} Team", dept.DepartmentId),
                    dept.BranchId,
                    dept.CompanyId
                }
            }
    ).ToList();

var teams = teamsWithScopes
    .Select(x => x.Team)
    .Adapt<List<Team>>();

await _repository.AddRangeAsync(teams);

// Build response with complete hierarchy
return teams.Select((team, index) => new TeamsResponseDto {
    TeamId = team.Id,
    DepartmentId = team.DepartmentId,
    BranchId = teamsWithScopes[index].BranchId,
    CompanyId = teamsWithScopes[index].CompanyId,
    OrganizationId = request.OrganizationId
}).ToList();
```

Result: Each team now has the complete path:

Organization → Company → Branch → Department → Team

What gets returned:

csharp

```
TeamsResponseDto {
    TeamId = team.Id,                    // New ID generated after saving
    DepartmentId = team.DepartmentId,    // Which department this team belongs to
    BranchId = teamsWithScopes[index].BranchId, // From wrapped object
    CompanyId = teamsWithScopes[index].CompanyId, // From wrapped object
    OrganizationId = request.OrganizationId      // From command input
}
```

Why return these?

- `TeamId` - The new ID created after saving to database
- `DepartmentId` - From team entity
- `BranchId` - Passed through from department (for scopes)
- `CompanyId` - Passed through from department (for scopes)
- `OrganizationId` - Passed from orchestrator (for scopes)

This is the complete hierarchy needed for scope generation!

Step 6: Generate Scopes

What it does: Creates scope records for access control

Code:

```csharp
var scopes = request.scopesData.Adapt<List<ScopeBase>>();
await _repository.AddRangeAsync(scopes);
return scopes.Count;
```

What are Scopes?

Scopes control what users can see. Each scope is one complete path:

```
Scope = {
  OrganizationId: "org-123",
  CompanyId: "comp-456",
  BranchId: "branch-789",
  DepartmentId: "dept-101",
  TeamId: "team-202"
}
```

Example:

User with this scope can ONLY see employees in "team-202", not the entire company.

Important Patterns

1. Why Commands Return Lists (Not Single Items)

Question: Why does each command return `List<CompaniesResponseDto>` instead of just IDs?

Answer: Because ONE organization can have MULTIPLE companies, branches, departments, and teams!

Example:

```csharp
// User sends 3 companies in one request
{
  "companies": [
    { "name": "Tech USA" },
    { "name": "Tech UK" },
    { "name": "Tech India" }
  ]
}

// Companies command returns List with 3 items
[
  { companyId: "comp-1", companyName: "Tech USA", branches: [...] },
  { companyId: "comp-2", companyName: "Tech UK", branches: [...] },
  { companyId: "comp-3", companyName: "Tech India", branches: [...] }
]

// Next step needs ALL 3 companies' data
```

Why return the full object (not just IDs)?

```csharp
// ❌ BAD - Return only IDs
return ["comp-1", "comp-2", "comp-3"];
// Problem: Next step needs CompanyName and Branches data
// Would need to query database again!

// ✅ GOOD - Return full data
return [
  { companyId: "comp-1", companyName: "Tech USA", branches: [...] },
  { companyId: "comp-2", companyName: "Tech UK", branches: [...] },
  { companyId: "comp-3", companyName: "Tech India", branches: [...] }
];
// Next step has everything it needs!
```

The same applies to all steps:

- Companies command → Returns List (multiple companies)
- Branches command → Returns List (multiple branches from multiple companies)
- Departments command → Returns List (multiple departments from multiple branches)
- Teams command → Returns List (multiple teams from multiple departments)

---

2. Data Passing Between Steps

Each step returns data that the next step needs:

```
Step 1 → OrganizationId
Step 2 → CompanyId, DefaultCityId, Branches
Step 3 → BranchId, CompanyId, Departments
Step 4 → DepartmentId, BranchId, CompanyId, Teams
Step 5 → Complete hierarchy (all IDs)
Step 6 → Uses all IDs to create scopes
```

Why?

To avoid querying the database again. We pass data forward through the chain.

---

2. Default Values

Rule: Every parent must have at least one child

```csharp
// Pattern used everywhere:
(collection != null && collection.Count != 0)
    ? collection            // Use provided data
    : [create default]      // Create default
```

Examples:

- Company with no branches → Create "CompanyName Branch"
- Branch with no departments → Create "BranchName Department"
- Department with no teams → Create "DepartmentName Team"

---

3. Keeping CompanyId Through the Chain

Problem: Teams don't directly know their CompanyId

```
Database Structure:
Organization → Company → Branch → Department → Team
                                        ↑
                                Only knows DepartmentId
```

Solution: Pass CompanyId through each step using anonymous objects

```csharp
// Step 4: Wrap Department with CompanyId
new { Department = d, branch.CompanyId }

// Step 5: Wrap Team with BranchId and CompanyId
new { Team = t, dept.BranchId, dept.CompanyId }

// Final response has everything
TeamsResponseDto {
    TeamId,
    DepartmentId,
    BranchId,
    CompanyId,
    OrganizationId
}
```

4. The Index Pattern

Why use `.Select((item, index) =>`?

```csharp
var teams = [...];          // Created entities with new IDs
var teamsWithScope = [...]; // Original data with CompanyId/BranchId

// Match them by index
var response = teams.Select((team, index) => new {
    TeamId = team.Id,                    // From created entity
    CompanyId = teamsWithScope[index].CompanyId  // From original data
});
```

Both lists have the same order and count, so index matches them perfectly:

```
teams[0] matches teamsWithScope[0]
teams[1] matches teamsWithScope[1]
teams[2] matches teamsWithScope[2]
```

Common Mistake: Empty List vs Null

```csharp
// ❌ WRONG - doesn't work for empty lists
company.Branches?.Where(b => b != null) ?? [default]

// When Branches = [], Where returns []
// [] is NOT null, so ?? doesn't trigger!

// ✅ CORRECT
(company.Branches != null && company.Branches.Count != 0)
    ? company.Branches
    : [default]
```

Complete Example

Request:

```json
{
  "organization": {
    "name": "Tech Solutions",
    "companies": [
      {
        "name": "Tech USA",
        "countryId": "USA",
        "branches": null
      }
    ]
  }
}
```

What Happens:

1. **Step 1:** Create Organization "Tech Solutions" → "org-123"
2. **Step 2:** Create Company "Tech USA" → "comp-456"
   - No branches provided
   - Fetch default city for USA → "NYC-001"
3. **Step 3:** Create default Branch
   - Name: "Tech USA Branch"
   - City: "NYC-001"
   - ID: "branch-789"
4. **Step 4:** Create default Department
   - Name: "Tech USA Branch Department"
   - ID: "dept-101"
   - Keep CompanyId: "comp-456"
5. **Step 5:** Create default Team
   - Name: "Tech USA Branch Department Team"
   - ID: "team-202"
   - Complete hierarchy built
6. **Step 6:** Create Scope

- org-123 → comp-456 → branch-789 → dept-101 → team-202

Result: Complete company structure created automatically!