🎤 Complete English Presentation Speech

Employee Management System with Architecture & Code Explanations

Duration: 12-13 Minutes

📋 PRESENTATION STRUCTURE

Total Time: 12-13 minutes

1. **Introduction** (1.5 min)
2. **System Architecture with Diagram** (2.5 min)
3. **Live Demo with Code Explanations** (6 min)
4. **Technical Discussion** (2 min)
5. **Q&A** (1 min)

SECTION 1: INTRODUCTION (1.5 minutes)

Opening

Good morning everyone. Thank you for giving me this opportunity to present today.

My name is [Your Name], and I'm currently working as an intern here at IRIS. Today, I'm excited to present the Employee Management System that I've developed during my internship.

Project Overview

This is a full-stack web application designed for managing employee information. The system provides complete CRUD functionality - Create, Read, Update, and Delete operations for employee records.

Key features include:

- User registration and authentication
- Adding new employees to the database
- Searching and filtering through employee records with real-time updates
- Updating employee information including department and retirement status
- Soft delete implementation for data retention and recovery

The application is built using modern web technologies. On the frontend, we have React.js with TypeScript for type safety. On the backend, we're using InterSystems IRIS database with ObjectScript for business logic.

Presentation Goals

In the next 12 minutes, I'll walk you through three main areas:

First, the system architecture - I'll explain how the three-tier design provides separation of concerns and scalability.

Second, a complete live demonstration - I'll show you the entire user journey from registration to employee management, and explain the key code implementations behind each feature.

Third, technical decisions and future improvements - I'll discuss the choices I made and how this system could be enhanced for production use.

Let's begin with the architecture.

SECTION 2: SYSTEM ARCHITECTURE (2.5 minutes)

Part 2.1: Architecture Diagram Explanation (1.5 minutes)

[Display: THREE_TIER_ARCHITECTURE.png]

Let me show you the system architecture. Our application follows a three-tier architecture, which is an industry-standard design pattern that separates concerns into distinct layers.

Presentation Tier (Top Layer)

[Point to the top of diagram - Presentation Tier]

At the top, we have the Presentation Tier. This is where users interact with the system through their web browser.

The frontend is a single-page application built with React 18. We're using TypeScript for compile-time type checking, which helps catch errors during development rather than in production. The UI components come from Material-UI, which provides professional, accessible components following Google's Material Design guidelines. And React Router handles client-side navigation without page reloads.

All of this runs in the user's browser on port 5173 during development.

Application Tier (Middle Layer)

[Point to the middle of diagram - Application Tier]

In the middle is the Application Tier - this is our REST API layer.

During development, there's a Vite proxy that forwards requests from port 5173 to port 52773. This solves CORS issues and provides a seamless development experience. In production, both tiers would typically be on the same domain.

The REST API handles three main responsibilities:

- First, authentication - user registration and login
- Second, employee CRUD operations - all the create, read, update, delete functionality
- Third, data formatting - converting between frontend JSON and backend database formats

Data Tier (Bottom Layer)

[Point to the bottom of diagram - Data Tier]

At the bottom is the Data Tier - the InterSystems IRIS database.

IRIS is a multi-model database that supports objects, SQL, and key-value storage simultaneously. We're using ObjectScript, which is IRIS's native language. ObjectScript provides direct database integration through persistent classes.

When I define a class that extends %Persistent, IRIS automatically creates the database table, generates SQL queries, and provides methods like %Save, %OpenId, and %Delete. This tight integration between code and database is very powerful.

One important technical detail is UTF-8 character encoding. Since we're working with Japanese employee names and addresses, I use the $ZCONVERT function throughout the backend to ensure Japanese characters display correctly without mojibake.

Data Flow

[Point to the arrows connecting the tiers]

The data flow is straightforward and follows standard REST principles:

The user interacts with the React interface, which sends HTTP requests. These go through the Vite proxy to the REST API. The API processes the requests, performs database operations using ObjectScript, and returns JSON responses. The frontend receives these responses and updates the UI accordingly.

This happens asynchronously, so the user interface remains responsive even during database operations.

Part 2.2: Architecture Benefits (30 seconds)

This three-tier architecture provides several key benefits:

Separation of concerns - each tier has a specific responsibility. The frontend focuses on user experience, the API handles business logic, and the database manages data persistence.

Independent scalability - if we need more database capacity, we can scale just the data tier without touching the frontend. If we need to handle more concurrent users, we can add more API servers.

Technology flexibility - we could replace React with Vue.js without changing the backend. Or migrate from IRIS to PostgreSQL without touching the frontend. Each tier is loosely coupled.

Easier maintenance - bugs are easier to isolate and fix when concerns are separated. A frontend developer can work independently from a backend developer.

Now let's see this architecture in action with a live demonstration.

SECTION 3: LIVE DEMO WITH CODE EXPLANATIONS (6 minutes)

Demo Introduction (15 seconds)

I'll now demonstrate the complete user journey from account creation to employee management. As we go through each feature, I'll briefly explain the key code implementations that make it work.

Let me start with a fresh browser session.

Phase 1: User Registration (60 seconds)

[Navigate to: http://localhost:5173/signup]

First, a new user needs to create an account. Here's our registration form.

UI Explanation

This is the SignUp component. Let me explain how it works technically.

This component uses React's controlled components pattern. Each input field - name, email, and password - is bound to a state variable using the useState hook. When the user types, React immediately updates the state and re-renders the component with the new value. This gives us complete control over the form data.

[Fill in the form as you explain]

- Name: 田中太郎 (Tanaka Taro)
- Email: tanaka.taro@example.com
- Password: password123

Validation Code

When I click the register button, the handleSubmit function executes. First, it calls e.preventDefault() to stop the browser's default form submission behavior, which would cause a page reload. We want JavaScript to handle everything.

Then it runs client-side validation. The code checks three things: all fields must be filled, the email must contain an @ symbol, and the password must be at least 8 characters long. This provides immediate feedback to the user without making a server request.

[Click Register button]

API Communication

If validation passes, the frontend calls our REST API using axios. The request is a POST to /sem/signup with the form data as JSON in the request body.

The Vite proxy intercepts this request and forwards it to the IRIS server on port 52773.

Backend Processing

On the backend, the AccountRegistration method receives the JSON, parses it using %FromJSON, validates the data again server-side - we never trust client-side validation alone - then checks if the email already exists using a SQL query.

If everything is valid, it creates a new tblAccount object, sets the properties, and calls %Save() to persist it to the database. IRIS automatically assigns an ID and creates the database record.

[Show success message and redirect]

Perfect! The account is created. Notice we're automatically redirected to the signin page. This is React Router's Navigate component - it's client-side navigation, no page reload, just instant transition.

Phase 2: User Login (45 seconds)

[Now on: http://localhost:5173/signin]

Now let's authenticate with the credentials we just created.

Authentication Flow

The signin page is similar to signup but simpler. We only need email and password.

[Fill in credentials]

- Email: tanaka.taro@example.com
- Password: password123

[Click Login button]

Backend Authentication

The API call goes to /sem/signin. The backend performs a SQL query to find the account by email, retrieves the stored password, and compares it with the submitted password.

Quick security note - in the current implementation, passwords are stored in plain text for demonstration purposes. In production, we would use bcrypt hashing with a salt, and this comparison would be done using bcrypt's verify function.

Session Management

On successful authentication, the frontend calls setAuthData, which stores two items in localStorage: isLoggedIn is set to 'true', and userEmail stores the user's email address. This maintains the authentication state even if the user refreshes the page.

[Show redirect to employee list]

We're automatically redirected to /employees. The ProtectedRoute component checked our authentication status by reading from localStorage and allowed access. If we weren't logged in, it would redirect us back to signin.

Phase 3: Employee List Features (45 seconds)

[Now on: http://localhost:5173/employees]

Here's our main employee list page. This is one of the most complex components in the application.

Component Architecture

The EmployeeList component uses three React hooks working together. First, useState manages the employee data, search keyword, filter settings, sort direction, and pagination state. Second, useEffect loads employees when the component mounts. Third, useMemo optimizes performance by memoizing the filtered and sorted results.

Data Loading

When this component loads, useEffect triggers the loadEmployees function. This makes a GET request to /sem/employees. The backend queries the database with a WHERE clause filtering deleteFlg = 0, so we only get active employees. The soft-deleted ones are hidden.

The data comes back as JSON, and we update the state with setEmployees. React re-renders the component, and the table displays the data.

Search Feature

[Type in search box: "山田"]

Let me demonstrate the search functionality. The search is implemented client-side using useMemo for performance. When the search keyword changes, useMemo recalculates the filtered list by checking if the keyword appears in the Employee ID, Name, or Kana Name fields. All comparisons are lowercase for case-insensitive matching.

[Show instant results]

The table updates instantly because React efficiently re-renders only the rows that changed.

[Clear search]

The filtering happens in memory with the data we already fetched. For small datasets like this, client-side filtering is very fast. For production systems with thousands of employees, we'd implement server-side pagination and filtering.

Phase 4: Add New Employee (90 seconds)

[Click "新規登録" (Add Employee) button]

Now let's add a new employee.

Multi-Mode Component

[Now on: http://localhost:5173/employees/new]

This is the EmployeeDetail component. It's interesting because it handles three different modes - add, edit, and delete - all in one component. The component detects the mode by checking the URL parameter. When the id is "new", we're in add mode. Otherwise, we're in edit mode for that specific employee.

Form Implementation

Let me fill in the employee information.

[Fill in form as you explain each field]

- Employee ID: 12345 - This is exactly 5 digits, validated both client and server side
- Name: 山田花子 (Yamada Hanako)
- Kana Name: ヤマダハナコ - The katakana pronunciation
- Sex: Female - Stored as integer 2 in the database
- Phone: 090-1234-5678
- Department: 営業部 (Sales Department)
- Post Code: 100-0001 - Japanese postal code format
- Address: 東京都千代田区千代田1-1

- Retirement Status: Unchecked - This employee is currently active

Client-Side Validation

[Click Register button]

The form validates that Employee ID is exactly 5 characters and that required fields like Name and Sex are filled in. This validation happens instantly without a server round-trip.

Confirmation Dialog

[Show dialog]

Before saving, we show a confirmation dialog. This is Material-UI's Dialog component with dynamic content based on the operation type - create, update, or delete.

[Click "はい" (Yes)]

Backend Processing

When confirmed, the CreateEmployee method on the backend first checks for duplicate Employee IDs among active records. This is important because with soft delete, we allow reusing IDs from deleted employees.

Then it creates a new tblEmployee object using %New(). The code sets all properties, using $ZCONVERT with "I" direction for Japanese text to convert from UTF-8 to IRIS's internal format. It sets deleteFlg to 0 for active status and upDateTime to the current timestamp using $ZDATETIME.

Finally, it calls %Save(). IRIS generates the SQL INSERT statement, executes it, assigns an auto-increment ID, and returns success.

[Show redirect to list with new employee]

Perfect! The employee is created and appears at the top of the list because we're sorting by update timestamp in descending order - newest first.

Phase 5: Edit Employee (60 seconds)

[Click edit icon on the employee we just created]

Let's edit this employee.

Loading Existing Data

[Now on: http://localhost:5173/employees/5]

The component detects we're in edit mode because the URL parameter is a number, not "new". It immediately triggers useEffect, which calls the backend's GetEmployeeById method.

The backend uses %OpenId to load the persistent object from the database by its ID. It checks if the object exists and if it's not soft-deleted, then builds a response object with all properties converted to UTF-8 for output using $ZCONVERT with "O" direction.

[Show pre-filled form]

The form is pre-filled with the existing data. Notice that the Employee ID field is disabled - we don't allow changing employee IDs because it could break referential integrity.

Updating Data

[Modify fields]

- Department: Change to 技術部 (Technical Department)
- Retirement Status: Check the box - This employee is now retiring

[Click Update button]

The same form component now calls UpdateEmployee instead of CreateEmployee. The backend opens the existing employee object with %OpenId, updates only the properties that changed, updates the timestamp, and calls %Save() again. IRIS generates an UPDATE SQL statement and executes it.

[Show confirmation and updated list]

The record is updated. Notice the department changed to Technical Department, and there's now a "退職済み" (Retired) badge displayed.

Phase 6: Soft Delete (40 seconds)

[Click edit on the same employee]

Now let me demonstrate the delete functionality.

Delete Button

[Show delete button at bottom]

Notice the delete button at the bottom - this only appears in edit mode, not in add mode.

Soft Delete Pattern

[Click Delete button and show dialog]

This application implements soft delete rather than hard delete, which is a best practice for production systems.

[Click confirm]

Instead of actually removing the record from the database, the DeleteEmployee method opens the employee object, sets deleteFlg to 1, updates the timestamp, and saves. The record still exists in the database - it's just marked as deleted.

[Show list - employee disappeared]

The employee disappears from the list because our GetAllEmployees query filters WHERE deleteFlg = 0.

Benefits of Soft Delete

This approach has several advantages. We can recover the data if it was deleted by mistake - just set the flag back to 0. We maintain a complete audit trail of all employees, even departed ones. We preserve referential integrity - no broken foreign key references. And we comply with data retention policies that may require keeping records for a certain number of years.

In the database, we have a composite unique index on EmployeeId and deleteFlg. This clever design allows us to reuse employee ID 12345 for a new employee because the database sees ("12345", 0) as different from ("12345", 1).

Phase 7: Logout (20 seconds)

[Click logout button in the navigation bar]

Finally, let's log out.

The handleLogout function calls clearAuthData, which removes the isLoggedIn and userEmail items from localStorage. Then it navigates to the signin page with the replace option set to true, which replaces the current history entry. This means the browser's back button won't take us back to the protected page.

[Show signin page]

We're back at signin. If I try to access /employees directly now...

[Type /employees in address bar]

The ProtectedRoute component checks isAuthenticated(), which returns false because localStorage was cleared, so it immediately redirects back to signin. This ensures all protected routes are properly secured.

SECTION 4: TECHNICAL DISCUSSION (2 minutes)

Key Technical Decisions (60 seconds)

Let me highlight some important technical decisions in this implementation.

1. React Hooks over Class Components

I chose functional components with hooks throughout the application. Hooks like useState, useEffect, and useMemo are more concise than class component lifecycle methods, easier to test, and align with modern React best practices. The useMemo hook in EmployeeList, for example, prevents unnecessary recalculations when the component re-renders for unrelated reasons.

2. TypeScript for Type Safety

TypeScript interfaces define the exact shape of our data - User, Employee, and API responses. The compiler caught several bugs during development where I was accessing properties that didn't exist or passing wrong parameter types. This type checking happens at compile time, so there's zero runtime performance overhead.

3. Soft Delete Implementation

The soft delete pattern with a deleteFlg is crucial for production systems. The composite unique index on EmployeeId and deleteFlg is particularly elegant - it enforces only one active employee per ID while allowing multiple deleted records with the same ID. This gives us the flexibility to reuse IDs when needed.

4. Client vs Server Operations

The current implementation uses client-side filtering and sorting for responsiveness with small datasets. As data grows, we would implement server-side pagination with SQL OFFSET and LIMIT clauses, passing page numbers and filter parameters from the frontend. The backend would return paginated results with total count, and we'd show page numbers in the UI.

Production Improvements (60 seconds)

For production deployment, several enhancements would be necessary.

Security Enhancements

First, password security. We need bcrypt hashing with a strong work factor instead of plain text storage. We need JWT tokens for authentication instead of simple localStorage flags. We need rate limiting on authentication endpoints to prevent brute force attacks. We need HTTPS enforcement for all communications. And we need input sanitization to prevent XSS attacks.

Performance Optimizations

Second, performance. We need server-side pagination and filtering for large datasets. We need database query optimization with proper indexes on frequently queried columns. We need Redis caching for frequently accessed data like department lists. And we need code splitting to reduce the initial JavaScript bundle size.

Additional Features

Third, enhanced functionality. We need role-based access control - different permissions for HR administrators, department managers, and regular employees. We need audit logging to track who changed what data and when. We need Excel import and export for bulk operations. We need advanced search with multiple filter criteria and saved searches. And we need email notifications for important events like employee status changes.

Testing Strategy

Fourth, comprehensive testing. We need unit tests for business logic functions using Jest. We need integration tests for API endpoints. We need end-to-end tests for critical user workflows using Playwright or Cypress. And we need load testing to ensure the system handles the expected number of concurrent users.

These improvements would transform this demonstration system into a production-ready enterprise application.

SECTION 5: Q&A (1 minute)

Thank you for your attention. I'm happy to answer any questions you might have about the implementation, architecture, technology choices, or any other aspect of the system.

[Pause and wait for questions]

BACKUP: COMMON Q&A RESPONSES

Q1: Why Vite instead of Create React App?

Vite provides significantly faster development server startup and hot module replacement compared to Create React App. It uses native ES modules in development and only bundles what's needed. The dev server starts in under a second, while CRA can take 30 seconds or more for large projects. The proxy configuration in vite.config.ts was straightforward to set up for routing API requests to IRIS.

Q2: How does the proxy work exactly?

The Vite development server runs on port 5173. In vite.config.ts, I configured a proxy that intercepts any request starting with /sem and forwards it to http://localhost:52773. This solves CORS issues during development because from the browser's perspective, both frontend and API are on the same origin. In production, we would deploy both on the same domain or configure proper CORS headers on the backend.

Q3: What about performance with client-side filtering?

For datasets under 1000 records, client-side filtering is actually very fast - usually under 10 milliseconds. The useMemo hook ensures we only recalculate when dependencies change. However, for larger datasets, we would implement server-side filtering with API endpoints like GET /employees?search=keyword&page=1&limit=50, and the backend would return paginated results.

Q4: How would you handle concurrent edits?

Currently it's last-write-wins. If two users edit the same employee simultaneously, whoever saves last overwrites the other's changes. For production, I would implement optimistic locking. Add a version field to the employee table. When loading for edit, store the version. When saving, include a WHERE clause checking the version hasn't changed. If it has, return a conflict error and ask the user to refresh and retry.

Q5: Why Material-UI over other UI libraries?

Material-UI provides a comprehensive set of professionally designed, accessible components that work well together. It has excellent TypeScript support with complete type definitions. It follows Google's Material Design guidelines, which users are already familiar with. And the sx prop makes styling straightforward without writing separate CSS files. The ecosystem is mature with good documentation and community support.

Q6: What about SQL injection risks?

No risk of SQL injection because I use parameterized queries throughout. For example, in the duplicate email check: "SELECT ID FROM SEM.tblAccount WHERE Email = ?" and then pass the parameter separately in %Execute(email). This ensures user input is always treated as data, never as part of the SQL command. The database driver escapes any special characters automatically.

Q7: How would you implement role-based access control?

I would add a role field to tblAccount (admin, manager, employee). Store the role in the JWT token or session. Create a RequireRole higher-order component that checks permissions before rendering. On the backend, verify the role in each API method before allowing operations. For example, only HR admins can delete employees. Implement a permission matrix defining which roles can perform which actions on which resources.

Q8: What's your database backup strategy?

For production, I would implement a comprehensive backup strategy. Daily full backups with transaction log backups every 15 minutes for point-in-time recovery. Store backups in geographically separate locations. Test restore procedures monthly. Keep backups for 30 days. Use IRIS's built-in backup utilities which support online backups without downtime. For critical data, implement database mirroring or async replication to a standby server.

Q9: How do you ensure Japanese text displays correctly?

I use the $ZCONVERT function throughout the backend. When receiving data from the frontend, I use $ZCONVERT with "I" direction to convert from UTF-8 to IRIS's internal format. When sending data to the frontend, I use $ZCONVERT with "O" direction to convert back to UTF-8. The frontend sends charset=UTF-8 in HTTP headers, and IRIS respects this. This ensures Japanese characters like 山田太郎 display correctly without mojibake.

Q10: What happens if the API server goes down?

Currently, API errors are caught and displayed to the user as error messages. For production resilience, I would implement several strategies. First, retry logic with exponential backoff for transient failures. Second, circuit breaker pattern to fail fast when the backend is down. Third, load balancing across multiple API servers for high availability. Fourth, graceful degradation - show cached data with a warning that it might be stale. Fifth, health check endpoints to monitor server status.

CLOSING REMARKS

If no more questions:

Thank you all for your time and attention today. This project has been an excellent learning experience in full-stack development with modern technologies. I'm grateful for the opportunity to work with React, TypeScript, and InterSystems IRIS. The skills I've gained here - from frontend state management to database optimization to REST API design - will be valuable in future projects. I appreciate your feedback and look forward to applying your suggestions.

Thank you very much.

PRESENTATION TIPS

Before You Start:

1. Have the demo environment running and tested
2. Have backup screenshots in case demo fails
3. Practice transitions between sections
4. Time yourself - aim for 12-13 minutes
5. Prepare your laptop with all applications ready

During Presentation:

1. Speak clearly at a moderate pace
2. Make eye contact with different people
3. Use hand gestures when pointing to diagram
4. Don't read code line by line - explain concepts
5. If something breaks, stay calm and use screenshots
6. Pause briefly between sections
7. Check the time occasionally
8. Invite questions with open body language

For the Architecture Diagram:

1. Display it prominently on screen
2. Actually point to each tier as you explain it
3. Trace the data flow with your finger/pointer
4. Don't rush through it - this is important
5. Make sure everyone can see it clearly

For the Demo:

1. Type slowly so people can follow
2. Explain what you're doing before you do it
3. Point out UI elements as you use them
4. If you make a typo, it's okay - just fix it
5. Smile - show you're confident!

For Q&A:

1. Listen carefully to the full question
2. Repeat the question if needed
3. It's okay to say "That's a great question"
4. If you don't know, be honest
5. Connect answers back to what you demonstrated

TIMING BREAKDOWN

Section	Time	Key Points
Introduction	1.5 min	Who, what, why
Architecture Diagram	1.5 min	Three tiers, point and explain
Architecture Benefits	0.5 min	Separation, scaling, flexibility
Demo: Registration	1 min	Form, validation, API
Demo: Login	0.75 min	Auth, localStorage

Section	Time	Key Points
Demo: List Features	0.75 min	Loading, search
Demo: Add Employee	1.5 min	Form, validation, save
Demo: Edit Employee	1 min	Load, modify, update
Demo: Delete	0.7 min	Soft delete pattern
Demo: Logout	0.3 min	Security
Technical Decisions	1 min	Key choices
Production Improvements	1 min	Future enhancements
Q&A	1 min	Questions
TOTAL	**~13 min**	

Good luck with your presentation! You've got this! 🚀 💪