🎤 English Presentation Speech with Code Explanations

Employee Management System - 12-13 Minutes

📋 Presentation Structure

Total Time: 12-13 minutes

- Section 1: Introduction (1.5 min)
- Section 2: System Architecture (2.5 min)
- Section 3: Live Demo with Code (6 min)
- Section 4: Technical Discussion (2 min)
- Section 5: Q&A (1 min)

SECTION 1: INTRODUCTION (1.5 minutes)

Opening

Good morning/afternoon 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 allows users to:

- Register and authenticate themselves
- Add new employees to the database
- Search and filter through employee records
- Update employee information
- And manage employee lifecycle including retirement status

The application is built using modern web technologies - React.js with TypeScript on the frontend, and InterSystems IRIS with ObjectScript on the backend.

Presentation Objectives

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

1. The system architecture and technology choices
2. A live demonstration of the complete workflow
3. Key technical implementations in the code
4. And areas for future improvements

Let's begin with the architecture.

SECTION 2: SYSTEM ARCHITECTURE (2.5 minutes)

Part 2.1: High-Level Overview (1 minute)

The application follows a three-tier architecture.

[Show architecture diagram if available]

At the top layer, we have the **Frontend** - a React single-page application running in the browser. This is what the user interacts with directly.

In the middle, we have the **REST API layer** - this handles all communication between the frontend and backend. It processes HTTP requests, validates data, and formats responses.

At the bottom layer is the **Database** - InterSystems IRIS stores all our persistent data including user accounts and employee records.

Communication flows like this: The user interacts with the React interface, which sends HTTP requests to our REST API. The API processes these requests, performs database operations, and sends JSON responses back to the frontend. All of this happens through a Vite development proxy that routes requests from port 5173 to our IRIS server on port 52773.

Part 2.2: Technology Stack Details (1.5 minutes)

Let me dive a bit deeper into the technologies.

Frontend Stack:

We're using **React 18** with **TypeScript**. React gives us a component-based architecture and efficient rendering through its virtual DOM. TypeScript adds compile-time type checking, which helps catch errors before they reach production.

For the UI framework, I chose **Material-UI** because it provides professional-looking, accessible components out of the box. It follows Google's Material Design guidelines and saves significant development time.

The code uses **React Hooks** throughout - useState for managing component state, useEffect for side effects like API calls, and useMemo for performance optimization in the employee list where we're filtering and sorting large datasets.

For routing, we use **React Router v6** which provides client-side navigation without page reloads. I've implemented a ProtectedRoute component that guards authenticated pages - if a user tries to access the employee list without logging in, they're automatically redirected to the signin page.

Backend Stack:

On the backend, we're running **InterSystems IRIS** - a multi-model database that supports objects, SQL, and key-value storage simultaneously.

The business logic is written in **ObjectScript**, which is IRIS's native language. ObjectScript provides direct integration with the database through persistent classes. This is very powerful - when I define a class that extends %Persistent, IRIS automatically creates the corresponding database table, generates SQL queries, and provides methods like %Save(), %OpenId(), and %Delete().

For the API layer, I'm using IRIS's **%CSP.REST framework**. This provides built-in REST capabilities with URL routing through an XData block. I can define routes with HTTP methods, URL patterns including parameters, and map them to ClassMethod handlers.

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 code - with "I" direction when receiving data from the frontend, and "O" direction when sending data back. This ensures Japanese characters like 山田太郎 display correctly without mojibake.

Now let's see this system in action.

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

Setup Introduction (15 seconds)

I'll demonstrate the complete user journey from registration to employee management. As we go through each step, I'll briefly explain the key code implementations behind each feature.

Let me start with a fresh browser session to show the complete flow.

Phase 1: User Registration (60 seconds)

[Navigate to signup page]

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

[Show SignUp component code briefly]

This SignUp component uses React's controlled components pattern. Each input field is bound to a state variable using useState - so when the user types, the component state updates immediately, and React re-renders the input with the new value.

[Fill in the form]

- Name: 田中太郎
- Email: tanaka.taro@example.com
- Password: password123

[Point to code]

When the form is submitted, the handleSubmit function first prevents the default browser behavior, then performs client-side validation. We check that all fields are filled, that the email contains an @ symbol, and that the password is at least 8 characters. This provides immediate feedback to the user.

[Show validation code]

If validation passes, we call our API using axios. The request goes through the Vite proxy to the IRIS backend at /sem/signup.

[Click Register]

[Backend code explanation - show briefly]

On the backend, the AccountRegistration method receives the JSON request, parses it using %FromJSON, validates the data again server-side, checks if the email already exists using a SQL query, and if everything is valid, creates a new tblAccount object and saves it to the database using %Save().

[Show success message]

Perfect! The account is created and we're redirected to the signin page. Notice this is client-side navigation - no page reload, just instant transition thanks to React Router.

Phase 2: Login (40 seconds)

[Now on signin page]

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

[Show SignIn component]

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

[Point to auth flow]

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

[Security note]

A quick security note - in the current implementation, passwords are stored in plain text, which is fine for this demonstration but would need bcrypt hashing in production.

[Click Login]

[Show localStorage code briefly]

On successful authentication, the frontend calls setAuthData, which stores two items in localStorage: an isLoggedIn flag set to 'true', and the user's email. This maintains the session state across page refreshes.

[Show redirect]

And we're automatically redirected to the employee list page. The ProtectedRoute component checked our authentication status and allowed access.

Phase 3: Add New Employee (90 seconds)

[Now on employee list]

Here's our main employee list page. Let me walk through its key features before adding a new employee.

[Show EmployeeList component code]

This is one of the most complex components. It uses three React hooks working together:

- useState manages the employee data and UI states
- useEffect loads employees when the component mounts
- useMemo optimizes the filtering and sorting operations

[Point to features]

Notice we have a search box, a checkbox to show retired employees, and table headers that support sorting. The pagination at the bottom is handled by Material-UI's TablePagination component.

[Explain data flow]

When the component loads, useEffect calls the API to fetch all employees. The backend queries the database with a WHERE clause filtering deleteFlg = 0, so we only get active employees. The data flows back as JSON, and we update the state with setEmployees.

[Click Add Employee button]

Now let's add a new employee. This navigates to /employees/new.

[Show EmployeeDetail component code]

The EmployeeDetail component is interesting because it handles three different modes - add, edit, and delete - all in one component. It detects the mode by checking the URL parameter: if id equals "new", we're in add mode; otherwise, we're in edit mode for that specific employee ID.

[Fill in employee form]

- Employee ID: 12345
- Name: 山田花子
- Kana Name: ヤマダハナコ
- Sex: Female (select radio button)
- Phone: 090-1234-5678
- Department: 営業部
- Post Code: 100-0001
- Address: 東京都千代田区千代田1-1
- Retirement Status: Unchecked (currently employed)

[Point to validation]

The form validates that Employee ID is exactly 5 digits and that required fields like Name and Sex are filled in.

[Click Register button]

[Show confirmation dialog code]

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

[Click Yes in dialog]

[Backend explanation]

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 can reuse IDs from deleted employees. Then it creates a new tblEmployee object, sets all the properties using $ZCONVERT for Japanese text, sets deleteFlg to 0 for active status, sets the current timestamp with $ZDATETIME, and saves to the database.

[Show success and redirect]

Perfect! The employee is created, and we're back at the list. The new employee appears at the top because we're sorting by update timestamp in descending order.

Phase 4: Search Functionality (30 seconds)

[Back on employee list]

Let me demonstrate the search feature.

[Show search implementation code]

The search is implemented using useMemo for performance optimization. When the search keyword changes, useMemo recalculates the filtered list instead of running the filter on every render.

[Type in search box: "山田"]

[Explain filter logic]

The filter checks three fields - Employee ID, Name, and Kana Name. We convert everything to lowercase for case-insensitive matching. Notice the search is client-side, not server-side - we filter the data we already have in memory. This is fast for small datasets but would need pagination and server-side filtering for thousands of records.

[Show instant results]

The table updates instantly as I type because React efficiently re-renders only what changed.

[Clear search]

Phase 5: Edit Employee (90 seconds)

[Click edit icon on an employee]

Now let's edit this employee. The URL changes to /employees/42 with the database ID.

[Show component mode detection code]

The EmployeeDetail component detects we're in edit mode because the URL parameter is not "new". It immediately calls useEffect which loads the employee data using the backend's GetEmployeeById method.

[Backend code explanation]

GetEmployeeById uses %OpenId to load the persistent object from the database. It checks if the object exists, checks if it's not soft-deleted, then builds a response object with all the employee properties converted to UTF-8 for output.

[Show pre-filled form]

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

[Modify some fields]

- Department: Change to 技術部
- Retirement Status: Check the box

[Point to update logic]

When I submit, the same form component but now it calls UpdateEmployee instead of CreateEmployee. The backend opens the existing employee object with %OpenId, updates only the properties that changed, updates the timestamp, and saves.

[Click Update button]

[Show confirmation dialog]

Notice the dialog text changes based on operation type - it says "更新確認" (Update Confirmation) instead of "登録確認" (Registration Confirmation).

[Confirm update]

[Show updated list]

Perfect! The employee record is updated. Notice the department changed and there's now a "退職済み" (Retired) status shown.

Phase 6: Delete Employee (40 seconds)

[Click edit on the same employee]

Now let me show you the delete functionality. Notice there's a delete button at the bottom - this only appears in edit mode, not in add mode.

[Show soft delete explanation]

This application implements soft delete, not hard delete. This is a best practice in production systems.

[Click Delete button]

[Show confirmation dialog]

The dialog warns that this operation cannot be undone - though technically with soft delete, we could undelete by setting the flag back to 0.

[Confirm delete]

[Backend explanation]

The DeleteEmployee method doesn't actually remove the record from the database. Instead, it 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 updated list]

The employee disappears from the list because our GetAllEmployees query filters WHERE deleteFlg = 0. If I check the "show retired employees" box, deleted employees still won't show because they're filtered out at the database level.

[Explain soft delete benefits]

This approach allows us to recover data if needed, maintain audit trails, preserve referential integrity, and comply with data retention policies. In the database, we have a composite unique index on (EmployeeId, deleteFlg), which means we can now reuse employee ID 12345 for a new employee because the old one has deleteFlg = 1.

Phase 7: Logout (20 seconds)

[Click logout button in header]

Finally, let's log out.

[Show logout code]

The handleLogout function calls clearAuthData, which removes the isLoggedIn and userEmail items from localStorage. Then it navigates to the signin page with replace: true, which replaces the history entry so the back button won't return to the protected page.

[Show signin page]

We're back at the signin page. If I try to navigate directly to /employees now...

[Try to access /employees]

[Show ProtectedRoute code]

The ProtectedRoute component checks isAuthenticated(), which returns false because localStorage is 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 (1 minute)

Let me highlight some key technical decisions in this implementation.

1. React Hooks over Class Components I chose functional components with hooks throughout the application because they're more concise, easier to test, and align with modern React best practices. Hooks like useMemo in the EmployeeList component prevent unnecessary recalculations of filtered and sorted data.

2. TypeScript for Type Safety TypeScript interfaces define the shape of our data - User, Employee, and API responses. This caught several bugs during development where I was accessing properties incorrectly. The type checking happens at compile time, so there's no runtime overhead.

3. Soft Delete Pattern Instead of permanently deleting records, we set a deleteFlg. This is crucial for data integrity and compliance. The composite unique index on (EmployeeId, deleteFlg) is particularly clever - it allows one active employee per ID but multiple deleted employees with the same ID.

4. Client-side vs Server-side Operations The search and filter are client-side for quick responsiveness with small datasets. For production with thousands of employees, we'd implement server-side pagination and filtering with OFFSET/LIMIT clauses in SQL and pass page parameters from the frontend.

Areas for Improvement (1 minute)

In a production environment, several enhancements would be necessary:

Security:

- Password hashing with bcrypt instead of plain text storage
- JWT tokens for authentication instead of simple localStorage flags
- HTTPS enforcement for all communications
- CSRF protection and rate limiting on API endpoints
- Input sanitization to prevent XSS attacks

Performance:

- Server-side pagination for large employee lists
- Database query optimization with proper indexes
- Caching frequently accessed data
- Code splitting to reduce initial bundle size

Features:

- Role-based access control - different permissions for HR, managers, and regular users

- Audit logging to track who changed what and when
- Excel import/export functionality for bulk operations
- Advanced search with multiple filter criteria
- Email notifications for important changes

Testing:

- Unit tests for business logic
- Integration tests for API endpoints
- End-to-end tests for critical user workflows
- Load testing to ensure the system handles concurrent users

These improvements would make the system production-ready for an enterprise environment.

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, or any technical decisions I made.

[Pause for questions]

BACKUP: Common Questions & Answers

Q1: Why did you choose Vite over Create React App?

Vite offers significantly faster development server startup and hot module replacement. It uses native ES modules in development and only bundles what's needed. The proxy configuration in vite.config.ts was straightforward to set up for routing API requests to IRIS.

Q2: How does the proxy work between frontend and backend?

The Vite dev server runs on port 5173. When the frontend makes a request to /sem/employees, the proxy configuration intercepts it and forwards it to http://localhost:52773/sem/employees where IRIS is running. This solves CORS issues during development. In production, you'd typically deploy both on the same domain or configure CORS headers.

Q3: What are the performance implications of client-side filtering?

For small to medium datasets (under 1000 records), client-side filtering is very fast and provides instant feedback. The useMemo hook ensures we only recalculate when dependencies change. However, for larger datasets, we'd need server-side filtering with API endpoints like GET /employees?search=keyword&page=1&limit=50.

Q4: Why Material-UI instead of other component libraries?

Material-UI provides comprehensive, accessible components that work well together. It has excellent TypeScript support and follows Google's Material Design guidelines which are familiar to users. The sx prop makes styling straightforward without writing separate CSS files.

Q5: How would you scale this to handle 10,000 concurrent users?

Several approaches:

1. Implement server-side pagination to limit data transfer
2. Add Redis caching for frequently accessed data
3. Use connection pooling for database connections
4. Implement horizontal scaling with load balancers
5. Add CDN for static assets
6. Optimize database queries with proper indexes
7. Consider API rate limiting per user

Q6: Are there SQL injection risks in your queries?

No, because I use parameterized queries throughout. For example, in the signup check: "SELECT ID FROM SEM.tblAccount WHERE Email = ?" and then pass the parameter separately in %Execute(). This ensures user input is always treated as data, not as part of the SQL command.

Q7: Why not use Redux for state management?

For this application's scope, Redux would be overkill. We're using React Query patterns with useState and useEffect, which is sufficient. Most state is local to components or passed through props. If the app grew significantly with complex state shared across many components, then Redux or Zustand would make sense.

Q8: How do 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. In production, we'd implement optimistic locking with a version field or timestamp checking, returning a conflict error if the data changed since it was loaded.

Q9: Is the application mobile-responsive?

Yes, Material-UI components are responsive by default. The Container component adjusts its max-width based on breakpoints. The table has horizontal scrolling on small screens. However, for a better mobile experience, we'd consider a different layout like cards instead of tables for the employee list.

Q10: How would you implement role-based access control?

I'd add a role field to the user account (e.g., admin, manager, employee). Store the role in the JWT token or session. Create higher-order components like RequireRole that check permissions before rendering components. On the backend, verify roles in each API method before allowing operations. For example, only HR admins can delete employees.

CLOSING REMARKS

If no more questions:

Thank you all for your time and attention. This project has been a valuable learning experience in full-stack development, and I'm grateful for the opportunity to work with modern technologies like React and IRIS. I look forward to applying these skills in future projects.

TIMING BREAKDOWN

Section	Time	Content
Introduction	1.5 min	Opening, project overview, objectives
Architecture	2.5 min	3-tier overview, frontend stack, backend stack
Demo: Signup	1 min	Registration with validation
Demo: Login	0.7 min	Authentication flow
Demo: Add Employee	1.5 min	Form, validation, confirmation
Demo: Search	0.5 min	Client-side filtering
Demo: Edit	1.5 min	Pre-fill, update, save
Demo: Delete	0.7 min	Soft delete pattern
Demo: Logout	0.3 min	Session cleanup
Technical Discussion	2 min	Key decisions & improvements
Q&A	1 min	Questions
TOTAL	**~13 min**	Complete presentation

PRESENTATION TIPS

1. **Speak clearly and at moderate pace** - You have time, don't rush
2. **Make eye contact** with different audience members
3. **Use hand gestures** when explaining architecture
4. **Point to code** but don't read it line by line
5. **Practice transitions** between sections
6. **Have backup plan** if demo fails (screenshots)
7. **Show enthusiasm** - you built this!
8. **Breathe** and pause between sections
9. **If stuck in Japanese**, switch to English (they understand)
10. **Smile** - confidence is contagious!

Good luck with your presentation! 🚀