🎤 Presentation Speech - English Version

Employee Management System Demo

For Senior Engineers (12-13 minutes)

📋 Speech Structure

Total Time: 12-13 minutes

- Introduction: 1.5 minutes
- System Architecture: 2.5 minutes
- Live Demo: 6 minutes
- Technical Discussion: 2 minutes
- Q&A: 1 minute

🎬 SECTION 1: Introduction (1.5 minutes)

Good morning/afternoon everyone. My name is [Your Name], and I'm currently an intern working with InterSystems IRIS and React.js.

Today, I'll be presenting a simple Employee Management System that I've developed as a learning project. This system demonstrates a full-stack web application using modern technologies—specifically React.js with TypeScript on the frontend and InterSystems IRIS with ObjectScript on the backend.

[Pause and show the landing page]

The main purpose of this presentation is to demonstrate:

- A complete CRUD application workflow
- Integration between React and IRIS
- Practical implementation of REST APIs
- Modern frontend development practices

I'll walk you through the entire flow—from user signup to employee management operations—and I'm happy to answer any technical questions you might have along the way.

Let's begin with the system architecture.

🏗️ SECTION 2: System Architecture (2.5 minutes)

2.1 High-Level Overview (1 minute)

[Draw or show architecture diagram if available]

This application follows a three-tier architecture:

First, the Frontend Layer:

- Built with React 18 and TypeScript
- Uses Material-UI for component styling
- Vite as the build tool for fast development
- Axios for HTTP communication

Second, the API Layer:

- RESTful API design following standard HTTP methods
- JSON data format for communication
- Seven main endpoints for authentication and CRUD operations

Third, the Backend Layer:

- InterSystems IRIS as the database engine
- ObjectScript for business logic
- Two main tables: tblAccount for users and tblEmployee for employee records

The frontend runs on port 5173, and we use Vite's proxy feature to communicate with the IRIS backend on port 52773. This avoids CORS issues during development.

2.2 Technical Stack Details (1.5 minutes)

Frontend Technologies:

React was chosen for its component-based architecture, which makes the UI modular and maintainable. We're using functional components exclusively with Hooks—primarily useState for state management, useEffect for side effects like API calls, and useMemo for performance optimization of filtering operations.

TypeScript adds type safety, which catches errors at compile time rather than runtime. For example, our Employee interface defines exactly what properties an employee object should have, and the TypeScript compiler enforces this throughout the application.

Material-UI provides pre-built components that follow Google's Material Design guidelines, ensuring a consistent and professional user interface.

Backend Technologies:

On the backend, we're using InterSystems IRIS, which is a multi-model database. It supports object persistence, which means we can work with ObjectScript classes that directly map to database tables.

Our REST API is built using IRIS's %CSP.REST framework. The API handler class extends this framework and defines URL routes using XML configuration. Each route maps to an ObjectScript ClassMethod that handles the request.

One important feature we implemented is UTF-8 conversion using ZCONVERT for proper Japanese character handling, since this system is designed for Japanese employees.

[Transition to demo]

Now, let me demonstrate how these technologies work together in practice.

💻 SECTION 3: Live Demo (6 minutes)

3.1 User Signup (1 minute)

[Navigate to signup page]

First, I'll create a new user account. This is our signup page.

[Start filling the form while talking]

Let me enter a test account:

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

[Talk about validation while typing]

Notice that the frontend performs validation. For example, if I don't include the @ symbol in the email, the system won't let me submit. The password must be at least 8 characters.

[Click Register button]

When I click Register, the frontend sends a POST request to /sem/signup. The backend validates this data again—we never trust client-side validation alone—and then creates a new record in the tblAccount table.

[Show success and redirect]

After successful registration, the system redirects us to the login page. This is a common UX pattern.

3.2 User Login (40 seconds)

[Fill in login form]

Now I'll log in with the credentials we just created.

[Submit login]

The system sends a POST request to /sem/signin. The backend queries the database to verify the email exists and the password matches. In a production system, we would hash passwords using bcrypt, but for this learning project, we're storing them directly.

[After successful login]

Upon successful authentication, the system stores a session flag in localStorage and redirects to the employee list page. Notice the navigation bar appears with a logout button—this is our Layout component which wraps all authenticated pages.

3.3 Add New Employee (1.5 minutes)

[Click New Registration button]

Let's add a new employee. This button navigates to /employees/new.

[Start filling form]

I'll create an employee named 山田花子 (Yamada Hanako):

- Employee ID: 12345 (must be exactly 5 digits)
- Name: 山田花子
- Kana Name: ヤマダハナコ
- Sex: Female (女性)
- Post Code: 100-0001
- Address: 東京都千代田区千代田1-1
- Phone: 090-1234-5678
- Department: 営業部 (Sales Department)
- Retire Flag: unchecked (currently employed)

[Explain while filling]

The form uses controlled components—each input's value is tied to a state variable using React's useState hook. When the user types, we update the state, which then updates the input value. This gives us complete control over the form data.

[Click Register button]

Before submitting, the system shows a confirmation dialog. This prevents accidental submissions.

[Click Yes in dialog]

The frontend sends a POST request with the employee data. Let's see what happens on the backend:

[Explain API flow]

1. The IRIS REST handler receives the JSON payload
2. It validates required fields
3. It checks for duplicate Employee IDs using an SQL query
4. It converts the Japanese text to UTF-8 using ZCONVERT
5. It creates a new tblEmployee object
6. It sets all properties including deleteFlg=0 and the current timestamp
7. It calls %Save() to persist the object to the database
8. It returns a success response

[Show success and redirect]

We're redirected back to the list, and our new employee appears in the table.

3.4 Search Functionality (30 seconds)

[Type in search box]

The search feature demonstrates client-side filtering. Let me type "山田" in the search box.

[Show filtering happening]

Notice how the table updates immediately—there's no API call. This is because we load all employees once and then filter them locally using React's useMemo hook. This pattern works well for small datasets. For large datasets, we'd implement server-side pagination and search.

The search checks Employee ID, Name, and Kana Name fields. It's case-insensitive and uses simple substring matching.

[Clear search]

When I clear the search, all employees reappear.

3.5 Edit Employee (1.5 minutes)

[Click Add another employee first - Employee ID 67890, name 佐藤次郎]

Let me quickly add one more employee so I can demonstrate the edit function.

[After adding, click Edit icon]

Now I'll click the edit icon for this employee. Notice we're navigating to /employees/67890—the employee's database ID becomes part of the URL. This is a RESTful pattern.

[Show pre-filled form]

The component recognizes it's in edit mode because the URL parameter is not "new". It makes a GET request to fetch the employee data and pre-fills the form.

One important detail: the Employee ID field is disabled during editing. We don't want users to change the Employee ID of an existing employee because it might be referenced elsewhere.

[Make changes]

Let me make some changes:

- Department: I'll change from 開発部 to 技術部
- Retire Flag: I'll check this to mark the employee as retired

[Click Update button]

The system sends a PUT request to /sem/employee/67890. Notice we're using PUT instead of POST—this follows REST conventions. PUT is for updating existing resources.

[Confirm in dialog]

After confirming, the backend uses ObjectScript's %OpenId method to load the existing employee object, modifies the properties, and saves it back to the database.

[Show success]

The updated employee now shows "技術部" as the department and has the retire flag set.

3.6 Delete Employee (40 seconds)

[In edit mode, click Delete button]

Finally, let's delete this employee. The delete button is only available in edit mode.

[Explain before confirming]

This system implements soft delete. We don't actually remove the record from the database. Instead, we set deleteFlg to 1. This has several advantages:

- Data recovery is possible
- We maintain audit trails

- Referential integrity is preserved
- It's better for regulatory compliance

[Confirm deletion]

The backend sends a DELETE request, the system sets deleteFlg=1, and the employee disappears from the list. However, if we query the database directly, the record is still there.

[Show empty list or remaining employee]

The list query filters with "WHERE deleteFlg = 0", so deleted employees are automatically excluded.

3.7 Logout (20 seconds)

[Click Logout button]

Finally, let me demonstrate logout. When I click the logout button, the system clears localStorage, removing the session flag.

[Show redirect to login]

We're redirected to the login page. If I try to navigate directly to /employees, the ProtectedRoute component will catch this and redirect me back to login. This is our authentication guard mechanism.

🔧 SECTION 4: Technical Discussion (2 minutes)

4.1 Key Technical Decisions (1 minute)

Let me briefly discuss some key technical decisions in this project:

React Hooks over Class Components: We use functional components exclusively with Hooks. This is the modern React pattern—it's more concise and easier to understand. useState manages component state, useEffect handles side effects like API calls, and useMemo optimizes expensive calculations.

TypeScript for Type Safety: TypeScript catches many bugs before runtime. For example, if I try to pass a string where a number is expected, the compiler immediately tells me. This is especially valuable in larger projects.

Soft Delete Pattern: Instead of physically deleting records, we use a flag. This is common in production systems for data recovery and audit purposes.

Client-Side vs Server-Side Operations: Search and filtering happen client-side for better UX, but validation and data persistence always happen server-side for security.

4.2 Areas for Improvement (1 minute)

As this is a learning project, there are several areas that would need improvement for production:

Security:

- Passwords should be hashed using bcrypt
- Should implement JWT tokens instead of localStorage
- Need HTTPS in production
- Add CSRF protection
- Implement rate limiting on the API

Performance:

- Add pagination for large datasets
- Implement caching (Redis)
- Optimize database queries with indexes
- Consider lazy loading for images

Features:

- Add role-based access control
- Implement audit logging
- Add data export functionality
- Better error messages
- Email notifications

Testing:

- Unit tests for components
- Integration tests for API
- End-to-end tests with Playwright

❓ SECTION 5: Q&A (1 minute)

Thank you for your attention. I'm happy to answer any questions you might have about:

- The architecture decisions
- The implementation details
- React or IRIS specifics

- Any part of the demo you'd like me to clarify

📝 BACKUP: Common Questions & Answers

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

A: Vite offers significantly faster hot module replacement (HMR) and build times. It uses native ES modules during development, which eliminates the bundling step. The development experience is much smoother, especially for TypeScript projects. Additionally, Vite has better support for modern features and smaller bundle sizes.

Q2: "How does the proxy configuration work?"

A: In vite.config.ts, we configure a proxy that forwards any request starting with /sem to the IRIS backend at localhost:52773. This happens transparently—the browser thinks it's talking to localhost:5173, but the requests are proxied. This avoids CORS issues during development. In production, we'd use a reverse proxy like nginx.

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

A: For small datasets (hundreds of records), client-side filtering is faster because there's no network latency. However, for thousands of records, we'd need server-side pagination and filtering. React's useMemo hook helps by memoizing the filtered results—it only recalculates when dependencies change.

Q4: "Why use Material-UI instead of custom CSS?"

A: Material-UI provides:

- Consistent design language (Material Design)
- Accessibility built-in (ARIA attributes, keyboard navigation)
- Responsive components out of the box
- Theming support
- Well-tested components
- Faster development time

For a learning project or MVP, it's a good choice. For a highly customized design, custom CSS might be better.

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

A: Several approaches:

1. **Backend**: Add IRIS replicas for read operations, implement connection pooling
2. **Caching**: Redis for session data and frequently accessed records

3. **CDN**: Serve static assets (JS, CSS) from a CDN
4. **Load Balancing**: Multiple backend instances behind a load balancer
5. **Database**: Add indexes on frequently queried fields, consider sharding for very large datasets
6. **API**: Implement rate limiting and request throttling
7. **Frontend**: Code splitting to reduce initial bundle size

Q6: "What about SQL injection risks?"

A: IRIS's %SQL.Statement with parameterized queries (the ? placeholders) prevents SQL injection. The parameters are escaped automatically. However, we should also:

- Validate input types (TypeScript helps here)
- Sanitize user input
- Implement rate limiting
- Add WAF (Web Application Firewall) in production

Q7: "Why didn't you use Redux for state management?"

A: For this application's complexity, useState and props are sufficient. Redux adds overhead—boilerplate code, learning curve, and complexity. We'd consider Redux when:

- State is shared across many components
- State changes are complex
- We need time-travel debugging
- The team is already familiar with Redux

Modern alternatives like Zustand or React Context API are also lighter options.

Q8: "How do you handle concurrent edits?"

A: Currently, we don't—this is a known limitation. For production, we'd implement:

- **Optimistic locking**: Add a version field, check it before updating
- **Pessimistic locking**: Lock the record during editing
- **Last-write-wins**: Current behavior (simpler but can lose data)
- **Conflict resolution UI**: Show both versions and let user decide

The choice depends on business requirements. Financial systems usually need strong consistency, while social media can use eventual consistency.

Q9: "What about mobile responsiveness?"

A: Material-UI components are responsive by default. The Grid system and breakpoints handle different screen sizes. However, for a truly mobile-first experience, we might:

- Optimize the table for mobile (cards instead of table rows)
- Add touch-friendly controls
- Implement progressive web app (PWA) features
- Consider a separate mobile app

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

A: Add a role field to tblAccount (Admin, Manager, Employee). Then:

1. **Backend**: Check role before each operation
2. **Frontend**: Hide/disable UI elements based on role
3. **API**: Return different data based on role
4. **Database**: Create views with row-level security

Example:

```typescript
// Frontend
{user.role === 'Admin' && <Button>Delete</Button>}

// Backend
If role '= "Admin" {
  // Allow delete
} Else {
  Return "Unauthorized"
}
```

🎯 Closing Statement

Once again, thank you for your time. This project has been a valuable learning experience in full-stack development, and I appreciate the opportunity to present it to you today. I look forward to your feedback and suggestions for improvement.

📌 Presentation Tips for Delivery

Voice & Pace:

- Speak clearly and at a moderate pace
- Pause after key points
- Vary your tone to maintain interest
- Don't rush through technical terms

Body Language:

- Maintain eye contact with audience
- Use hand gestures to emphasize points
- Stand confidently but naturally
- Face the audience, not the screen

Handling Technical Issues:

- Stay calm if demo fails
- Have screenshots as backup
- Explain what should happen
- Offer to show code instead

Engaging Audience:

- Ask if they can see the screen clearly
- Invite questions during demo
- Watch for confused expressions
- Adjust pace based on reactions

Time Management:

- Glance at clock periodically
- Have a shortened version ready
- Know which sections can be cut
- Save time for Q&A

You've got this! Speak confidently about what you learned, be honest about limitations, and show enthusiasm for the technology. Good luck! 🚀