Migration Guide

Repository: MomCareAPI

Generated at: 7/24/2025, 10:39:24 PM

This document provides a detailed migration guide for your project, including code analysis and recommendations.

Table of Contents

48. src/main/java/com/example/momcare/models/MenuCategory.java
49. src/main/java/com/example/momcare/models/MomHealthIndex.java
50. src/main/java/com/example/momcare/models/Music.java
51. src/main/java/com/example/momcare/models/MusicCategory.java
52. src/main/java/com/example/momcare/models/Notification.java
53. src/main/java/com/example/momcare/models/NotificationType.java
54. src/main/java/com/example/momcare/models/PeriodicCheck.java
55. src/main/java/com/example/momcare/models/Reaction.java
56. src/main/java/com/example/momcare/models/Role.java
57. src/main/java/com/example/momcare/models/SocialComment.java
58. src/main/java/com/example/momcare/models/SocialPost.java
59. src/main/java/com/example/momcare/models/SocialReaction.java
60. src/main/java/com/example/momcare/models/SocialStory.java
61. src/main/java/com/example/momcare/models/StandardsIndex.java
62. src/main/java/com/example/momcare/models/Tracking.java
63. src/main/java/com/example/momcare/models/User.java
64. src/main/java/com/example/momcare/models/UserStory.java
65. src/main/java/com/example/momcare/models/Video.java
66. src/main/java/com/example/momcare/models/VideoCategory.java
67. src/main/java/com/example/momcare/models/VideoMedia.java
68. src/main/java/com/example/momcare/models/WarningHealth.java
69. src/main/java/com/example/momcare/payload/request/AddUserFollowerRequest.java
70. src/main/java/com/example/momcare/payload/request/BabyHealthIndexRequest.java
71. src/main/java/com/example/momcare/payload/request/ChangePasswordRequest.java
72. src/main/java/com/example/momcare/payload/request/CreatePasswordRequest.java
73. src/main/java/com/example/momcare/payload/request/DiaryRequest.java
74. src/main/java/com/example/momcare/payload/request/MomHealthIndexRequest.java
75. src/main/java/com/example/momcare/payload/request/NotificationHandlerRequest.java
76. src/main/java/com/example/momcare/payload/request/NotificationRequest.java
77. src/main/java/com/example/momcare/payload/request/OPTRequest.java
78. src/main/java/com/example/momcare/payload/request/ShareResquest.java
79. src/main/java/com/example/momcare/payload/request/SocialCommentDeleteRequest.java
80. src/main/java/com/example/momcare/payload/request/SocialCommentNewRequest.java
81. src/main/java/com/example/momcare/payload/request/SocialCommentUpdateRequest.java
82. src/main/java/com/example/momcare/payload/request/SocialPostNewRequest.java
83. src/main/java/com/example/momcare/payload/request/SocialPostUpdateResquest.java
84. src/main/java/com/example/momcare/payload/request/SocialReactionDeleteRequest.java
85. src/main/java/com/example/momcare/payload/request/SocialReactionNewRequest.java
86. src/main/java/com/example/momcare/payload/request/SocialReactionUpdateRequest.java
87. src/main/java/com/example/momcare/payload/request/SocialStoryNewRequest.java
88. src/main/java/com/example/momcare/payload/request/StandIndexRequest.java
89. src/main/java/com/example/momcare/payload/request/UserHandlerRequest.java
90. src/main/java/com/example/momcare/payload/request/UserLoginRequest.java
91. src/main/java/com/example/momcare/payload/request/UserRequest.java
92. src/main/java/com/example/momcare/payload/request/UserSignUpRequest.java
93. src/main/java/com/example/momcare/payload/request/UserStoryDeleteRequest.java
94. src/main/java/com/example/momcare/payload/request/UserStoryNewRequest.java
95. src/main/java/com/example/momcare/payload/response/HandBookDetailResponse.java
96. src/main/java/com/example/momcare/payload/response/MenuDetailResponse.java

97. src/main/java/com/example/momcare/payload/response/NotificationResponse.java
98. src/main/java/com/example/momcare/payload/response/Response.java
99. src/main/java/com/example/momcare/payload/response/SocialCommentResponse.java
100. src/main/java/com/example/momcare/payload/response/SocialPostResponse.java
101. src/main/java/com/example/momcare/payload/response/SocialReactionResponse.java
102. src/main/java/com/example/momcare/payload/response/
StandardsBabyIndexResponse.java
103. src/main/java/com/example/momcare/payload/response/
StandardsMomIndexResponse.java
104. src/main/java/com/example/momcare/payload/response/TrackingWeekDetailResponse.java
105. src/main/java/com/example/momcare/payload/response/TrackingWeekResponse.java
106. src/main/java/com/example/momcare/payload/response/UserLoginResponse.java
107. src/main/java/com/example/momcare/payload/response/UserProfile.java
108. src/main/java/com/example/momcare/payload/response/UserResponse.java
109. src/main/java/com/example/momcare/payload/response/UserStoryResponse.java
110. src/main/java/com/example/momcare/repository/DiaryRepository.java
111. src/main/java/com/example/momcare/repository/HandBookCategoryRepository.java
112. src/main/java/com/example/momcare/repository/HandBookCollectionRepository.java
113. src/main/java/com/example/momcare/repository/HandBookRepository.java
114. src/main/java/com/example/momcare/repository/MenuCategoryRepository.java
115. src/main/java/com/example/momcare/repository/MenuRepository.java
116. src/main/java/com/example/momcare/repository/MusicCategoryRepository.java
117. src/main/java/com/example/momcare/repository/MusicRepository.java
118. src/main/java/com/example/momcare/repository/NotificationRepository.java
119. src/main/java/com/example/momcare/repository/PeriodicCheckRepository.java
120. src/main/java/com/example/momcare/repository/SocialCommentRepository.java
121. src/main/java/com/example/momcare/repository/SocialPostRepository.java
122. src/main/java/com/example/momcare/repository/TrackingRepository.java
123. src/main/java/com/example/momcare/repository/UserRepository.java
124. src/main/java/com/example/momcare/repository/UserStoryRepository.java
125. src/main/java/com/example/momcare/repository/VideoCategoryRepository.java
126. src/main/java/com/example/momcare/repository/VideoRepository.java
127. src/main/java/com/example/momcare/security/CheckAccount.java
128. src/main/java/com/example/momcare/security/Encode.java
129. src/main/java/com/example/momcare/security/JwtAuthenticationFilter.java
130. src/main/java/com/example/momcare/security/JwtService.java
131. src/main/java/com/example/momcare/service/BabyHealthIndexService.java
132. src/main/java/com/example/momcare/service/DiaryService.java
133. src/main/java/com/example/momcare/service/EmailService.java
134. src/main/java/com/example/momcare/service/HandBookCategoryService.java
135. src/main/java/com/example/momcare/service/HandBookCollectionService.java
136. src/main/java/com/example/momcare/service/HandBookService.java
137. src/main/java/com/example/momcare/service/MenuCategoryService.java
138. src/main/java/com/example/momcare/service/MenuService.java
139. src/main/java/com/example/momcare/service/MomHealthIndexService.java
140. src/main/java/com/example/momcare/service/MusicService.java
141. src/main/java/com/example/momcare/service/NotificationService.java
142. src/main/java/com/example/momcare/service/PeriodicCheckService.java
143. src/main/java/com/example/momcare/service/SocialCommentService.java

144. src/main/java/com/example/momcare/service/SocialPostService.java
145. src/main/java/com/example/momcare/service/TrackingService.java
146. src/main/java/com/example/momcare/service/UserDetailCustom.java
147. src/main/java/com/example/momcare/service/UserService.java
148. src/main/java/com/example/momcare/service/UserStoryService.java
149. src/main/java/com/example/momcare/service/VideoService.java
150. src/main/java/com/example/momcare/util/Constant.java
151. src/main/java/com/example/momcare/util/SecurityUtil.java
152. src/main/java/com/example/momcare/util/Validator.java
153. src/main/resources/application.properties
154. src/test/java/com/example/momcare/MomcareApplicationTests.java
155. system.properties

1. .gitignore

Migration Guide: Java to JavaScript

File: HELP.md

Summary:
- The file contains exclusion patterns for various development environments.

Migration Plan:
1. Rewrite exclusion patterns for JavaScript project structures.
2. Update the file with JavaScript-specific exclusion patterns.

Code Snippets:
Before:
```md
HELP.md
target/
!.mvn/wrapper/maven-wrapper.jar
!**/src/main/**/target/
!**/src/test/**/target/
```

After:
```md
HELP.md
node_modules/
dist/
```

Recommended Libraries/Frameworks:
- None needed for this file.

Best Practices:
- Use common JavaScript project directory structures.

Pitfalls to Avoid:
- Forgetting to update exclusion patterns for JavaScript project directories.

File: .gitignore

Summary:
- Excludes certain files and directories from version control.

Migration Plan:
1. Translate Java-specific exclusions to JavaScript equivalents.
2. Modify the file to match JavaScript project requirements.

Code Snippets:
Before:
```gitignore
### STS ###
.apt_generated
.classpath
.factorypath
.project
.settings
.springBeans
.sts4-cache
```

After:
```gitignore
### Node.js ###
node_modules/
dist/
```

Recommended Libraries/Frameworks:
- None needed for this file.

Best Practices:
- Keep the .gitignore file updated with JavaScript project needs.

Pitfalls to Avoid:
- Overlooking specific JavaScript project files that need to be excluded.

General Recommendations:

Libraries/Frameworks:
- Use Node.js for server-side JavaScript development.
- Utilize Express.js for building RESTful APIs.
- Consider React or Vue.js for frontend development.
- Jest for testing JavaScript code.

Best Practices:
- Follow ES6 syntax and features.
- Use async/await instead of callbacks or Promises.
- Leverage npm for package management.
- Implement modularization using ES6 modules.

Pitfalls to Avoid:
- Mixing Java and JavaScript coding styles.
- Neglecting to update dependencies and libraries for JavaScript compatibility.

This migration guide provides a structured approach to converting Java code to JavaScript, ensuring a smooth transition and adherence to best practices in the JavaScript ecosystem.

2. .mvn/wrapper/maven-wrapper.jar

An error occurred while generating the migration guide for this file.

3. .mvn/wrapper/maven-wrapper.properties

Java to JavaScript Code Migration Guide

Maven Configuration Migration

Summary:
The code snippet provided is a Maven configuration file that specifies the distribution and wrapper URLs for Maven.

Migration Plan:
1. Define the URLs in a JavaScript configuration file.
2. Use the URLs as needed in the JavaScript code.

Code Snippets:
Before Migration (Java):
```properties
distributionUrl=https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.9.5/apache-maven-3.9.5-bin.zip
wrapperUrl=https://repo.maven.apache.org/maven2/org/apache/maven/wrapper/maven-wrapper/3.2.0/maven-wrapper-3.2.0.jar
```

After Migration (JavaScript):
```javascript
const distributionUrl = 'https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.9.5/apache-maven-3.9.5-bin.zip';
const wrapperUrl = 'https://repo.maven.apache.org/maven2/org/apache/maven/wrapper/maven-wrapper/3.2.0/maven-wrapper-3.2.0.jar';
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries/frameworks needed for this migration.

Best Practices and Idioms in JavaScript:
- Use `const` or `let` for variable declarations instead of `var`.
- Use single quotes for string literals.

Common Pitfalls to Avoid:
- Ensure the URLs are correctly formatted and accessible in the JavaScript environment.

Conclusion
By following the provided migration guide, you can successfully convert the Maven configuration code from Java to JavaScript. Remember to test the migrated code thoroughly to ensure its functionality.

4. README.md

Migration Guide: Java to JavaScript

MomCareAPI

Summary:
The MomCareAPI is a Java application using SDK 17 and MongoDB for data storage.

Migration Plan:
1. **Setup Environment:**
   - Install Node.js and npm for JavaScript development.
   - Choose a suitable IDE for JavaScript development (e.g., Visual Studio Code).

2. **Code Conversion:**
   - Start by converting Java classes to JavaScript classes or functions.
   - Update import statements to require statements in JavaScript.
   - Refactor MongoDB queries to use a compatible JavaScript library like Mongoose.

3. **Testing:**
   - Write unit tests using a JavaScript testing framework like Jest or Mocha.
   - Test the API endpoints using a tool like Postman.

4. **Deployment:**
   - Choose a hosting platform like AWS or Heroku for deploying the JavaScript application.
   - Configure the deployment process for the new JavaScript application.

Before/After Code Snippets:

Java - Before:
```java
import com.mongodb.MongoClient;
import com.mongodb.client.MongoDatabase;
```

JavaScript - After:
```javascript
const mongoose = require('mongoose');
```

```

**Java - Before:**
```java
MongoClient mongoClient = new MongoClient("localhost", 27017);
MongoDatabase database = mongoClient.getDatabase("momcare");
```

JavaScript - After:
```javascript
mongoose.connect('mongodb://localhost:27017/momcare', { useNewUrlParser: true,
useUnifiedTopology: true });
const db = mongoose.connection;
```

Recommended Libraries/Frameworks:
- **Express.js:** For building RESTful APIs in JavaScript.
- **Mongoose:** MongoDB object modeling tool for Node.js.
- **Jest/Mocha:** Testing frameworks for JavaScript.

Best Practices and Idioms:
- Use asynchronous programming with Promises or async/await.
- Follow the Node.js module system for code organization.
- Use ES6 features like arrow functions and template literals.

Common Pitfalls:
- Be cautious of differences in data types between Java and JavaScript.
- Ensure compatibility between Java SDK libraries and JavaScript libraries.
- Watch out for callback hell when dealing with nested callbacks in JavaScript.

Conclusion:
By following this migration guide, you can successfully convert the MomCareAPI from Java to JavaScript, leveraging the power of Node.js and MongoDB in your application. Remember to test thoroughly and follow best practices to ensure a smooth transition.
```

5. mvnw

Summary:
The provided code is a shell script for starting the Apache Maven Wrapper, which is used to manage and build Java projects. It includes environment variable setups, Java executable path resolution, Maven wrapper download functionality, and launching the Maven wrapper main class.

Migration Plan:
1. **Environment Setup:**
   - Identify equivalent environment variables in JavaScript.
   - Use `process.env` to access environment variables in Node.js.

2. **Java Executable Path Resolution:**
   - Use `which` command equivalent in Node.js to find Java executable.
   - Handle platform-specific path formatting differences.

3. **Download Functionality:**
   - Use Node.js `http` or `https` module for downloading resources.
   - Handle basic authentication if required.
   - Check for availability of `wget`, `curl`, or fall back to Java download.

4. **Launching Maven Wrapper:**
   - Use child_process module to spawn a new process for executing Java commands.
   - Set up classpath and command line arguments appropriately.

Before/After Code Snippets:
```javascript
// Before
if [ -z "$JAVA_HOME" ]; then
  javaExecutable="$(which javac)"
  javaHome="$(dirname "\"$javaExecutable\"")"
  JAVA_HOME="$javaHome"
  export JAVA_HOME
fi

// After
const { execSync } = require('child_process');
let javaExecutable = execSync('which javac').toString();
```

```
let javaHome = javaExecutable.substring(0, javaExecutable.lastIndexOf('/'));
process.env.JAVA_HOME = javaHome;
```

Recommended Libraries/Frameworks in JavaScript:
1. **Request-Promise-Native:** For HTTP/HTTPS requests.
2. **Child Process:** For spawning new processes.

Best Practices and Idioms in JavaScript:
1. **Use Promises:** Use asynchronous code to handle HTTP requests.
2. **Error Handling:** Implement proper error handling for HTTP requests and process executions.
3. **Environment Variables:** Use `process.env` to access environment variables.

Common Pitfalls to Avoid:
1. **Platform Differences:** Handle platform-specific path formatting and commands.
2. **Download Security:** Ensure secure downloading of resources.
3. **Error Handling:** Properly handle errors during process execution and HTTP requests.

By following the migration plan and recommendations, you can successfully convert the provided shell script to JavaScript for usage in Node.js environments.

6. mvnw.cmd

Migration Guide from Java to JavaScript

Summary
The given code is a batch script used as an Apache Maven Wrapper startup script. It sets up environment variables, validates Java installation, downloads Maven wrapper, and launches Maven with specified configurations.

Migration Plan
1. **Environment Variables Setup**
   - Translate batch script commands to JavaScript equivalents.
   - Use `process.env` to handle environment variables.

2. **Java Validation and Maven Setup**
   - Replace batch script logic with JavaScript logic.
   - Handle file operations using Node.js file system module.

3. **Downloading Maven Wrapper**
   - Utilize Node.js `https` or `node-fetch` library to download resources.
   - Handle file downloads and validations accordingly.

4. **Launching Maven**
   - Execute Maven commands using child process in Node.js.
   - Manage command line arguments and classpath appropriately.

Code Migration Snippets

Environment Variables Setup
```javascript
// Set equivalent of %HOME% to $HOME
if (!process.env.HOME) {
  process.env.HOME = process.env.HOMEDRIVE + process.env.HOMEPATH;
}
```

Java Validation and Maven Setup
```javascript
```

```
// Validate JAVA_HOME
if (!process.env.JAVA_HOME) {
  console.error('Error: JAVA_HOME not found in your environment.');
  console.error('Please set the JAVA_HOME variable in your environment.');
  process.exit(1);
}

// Check for Java executable
const javaExecutable = path.join(process.env.JAVA_HOME, 'bin', 'java.exe');
if (!fs.existsSync(javaExecutable)) {
  console.error('Error: JAVA_HOME is set to an invalid directory.');
  console.error(`JAVA_HOME = "${process.env.JAVA_HOME}"`);
  process.exit(1);
}
```

Downloading Maven Wrapper
```javascript
const wrapperJarPath = path.join(mavenProjectBaseDir, '.mvn', 'wrapper', 'maven-wrapper.jar');
const wrapperUrl = 'https://repo.maven.apache.org/maven2/org/apache/maven/wrapper/maven-wrapper/3.2.0/maven-wrapper-3.2.0.jar';

// Download Maven wrapper
const downloadWrapper = async () => {
  const file = fs.createWriteStream(wrapperJarPath);
  const response = await fetch(wrapperUrl);
  response.body.pipe(file);
};
```

Launching Maven
```javascript
const { exec } = require('child_process');

const mavenCmdLineArgs = process.argv.slice(2).join(' ');

exec(`"${javaExecutable}" ${mavenCmdLineArgs}`, (error, stdout, stderr) => {
  if (error) {
```

```
    console.error(error);
    process.exit(1);
  }
});
```

Recommended Libraries/Frameworks in JavaScript
- **Node.js**: For server-side JavaScript runtime environment.
- **`node-fetch`**: For making HTTP requests.
- **`child_process`**: For executing shell commands.
- **`path`**: For handling file paths.
- **`fs`**: For file system operations.

Best Practices and Idioms in JavaScript
- Use `const` and `let` for variable declarations based on immutability.
- Follow asynchronous programming using promises or async/await.
- Utilize modules to encapsulate code and promote reusability.
- Adhere to Node.js error-first callback convention for error handling.

Common Pitfalls to Avoid
- Beware of synchronous I/O operations blocking the event loop.
- Ensure proper error handling to prevent uncaught exceptions.
- Be cautious of security risks while downloading and executing external resources.
- Validate inputs and outputs thoroughly to avoid unexpected behavior.

By following this migration guide, you can efficiently convert the given Java batch script to a JavaScript equivalent, ensuring functionality and maintainability in the target environment.

7. pom.xml

Migration Guide: Java to JavaScript

Project POM File: pom.xml

Summary:
This XML file is a Maven Project Object Model (POM) file that defines the configuration of a Java project, including dependencies and build settings.

Migration Plan:
1. Convert the XML structure to a JSON object for easier manipulation in JavaScript.
2. Identify equivalent libraries or frameworks in JavaScript for dependencies.
3. Update the build configuration for JavaScript projects.

Before/After Code Snippets:
```xml
// Before (Java - pom.xml)
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
...
</project>
```

```javascript
// After (JavaScript - package.json)
{
  "name": "momcare",
  "version": "0.0.1",
  "description": "momcare",
  "dependencies": {
    // JavaScript dependencies here
  },
  "devDependencies": {
    // Development dependencies here
```

```
  },
  "scripts": {
    // Build scripts here
  }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for web server functionality
- Mongoose for MongoDB interactions
- Jest for testing

Best Practices and Idioms in JavaScript:
- Use ES6 syntax for modern JavaScript features.
- Follow async/await pattern for asynchronous operations.
- Use npm for package management.

Common Pitfalls to Avoid:
- Ensure compatibility of versions between libraries and frameworks.
- Handle asynchronous operations properly to avoid callback hell.

Database Configuration: MongoDB

Summary:
The project uses MongoDB as the database and includes the MongoDB Java driver as a dependency.

Migration Plan:
1. Install and configure MongoDB for the JavaScript environment.
2. Use a JavaScript MongoDB driver like Mongoose for interacting with the database.

Before/After Code Snippets:
```xml
// Before (Java - pom.xml)
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.12.14</version>
```

</dependency>
```

```javascript
// After (JavaScript - package.json)
{
  "dependencies": {
    "mongoose": "^6.0.9"
  }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose for MongoDB interactions

Best Practices and Idioms in JavaScript:
- Use Mongoose schemas for defining MongoDB data models.
- Use async/await with Mongoose queries.

Common Pitfalls to Avoid:
- Ensure proper connection handling with MongoDB in JavaScript.

API Security Configuration: OAuth2

Summary:
The project includes OAuth2 resource server and security dependencies for API security.

Migration Plan:
1. Implement OAuth2 security in JavaScript using frameworks like Passport.js.
2. Configure security middleware for API endpoints.

Before/After Code Snippets:
```xml
// Before (Java - pom.xml)
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    <version>3.3.0</version>

</dependency>
```

```javascript
// After (JavaScript - Example with Passport.js)
const passport = require('passport');
const OAuth2Strategy = require('passport-oauth2');

app.use(passport.initialize());
app.use(passport.session());

passport.use(new OAuth2Strategy({
    // OAuth2 configuration here
}));
```

Recommended Libraries/Frameworks in JavaScript:
- Passport.js for authentication middleware

Best Practices and Idioms in JavaScript:
- Use middleware for handling authentication and authorization.
- Store sensitive information securely, like tokens.

Common Pitfalls to Avoid:
- Handle token expiration and refresh properly in OAuth2 implementations.

By following this migration guide, you can successfully convert the Java project configuration to JavaScript, ensuring compatibility and best practices in the process.
```

8. src/main/java/com/example/momcare/MomcareApplication.java

Java to JavaScript Migration Guide

MomcareApplication.java

Summary:
The given Java code is a Spring Boot application class that starts the Momcare application using SpringApplication.

Migration Plan:
1. **Setup Node.js Environment:**
   - Make sure Node.js is installed on the system.

2. **Install Required Libraries:**
   - Use npm to install necessary libraries like express, nodemon, etc.

3. **Convert MomcareApplication Class:**
   - Create a JavaScript file (e.g., app.js) to serve as the entry point.
   - Use Express.js to set up the application and listen on a port.

Before (Java):
```java
package com.example.momcare;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MomcareApplication {

— V lic static void main(String[] args) {
™SpringApplication.run(MomcareApplication.class, args);
—Ð

}
```

After (JavaScript):
```javascript
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building web applications.
- Nodemon for automatic server restarts during development.

Best Practices and Idioms in JavaScript:
- Use ES6 syntax features like arrow functions, const, and let.
- Follow modularization and separation of concerns.

Common Pitfalls to Avoid:
- Be mindful of asynchronous operations in JavaScript.
- Ensure proper error handling and data validation.

Database/API Migration:
If the Java code interacts with a database or API:
- **Database Migration:**
  - Consider using MongoDB with Mongoose ORM for a NoSQL database.
  - Use Sequelize ORM for SQL databases like MySQL or PostgreSQL.
- **API Migration:**
  - Use Express.js to create API endpoints.
  - Axios for making HTTP requests to external APIs.

By following this migration guide, you can successfully convert the given Java code to JavaScript while leveraging the power and flexibility of the Node.js ecosystem.

9. src/main/java/com/example/momcare/config/AppConfig.java

Java to JavaScript Migration Guide

com.example.momcare.config.AppConfig

Summary:
This Java code is a configuration class defining beans for password encoding, security filter chain, JWT authentication converter, JWT decoder, and JWT encoder.

Migration Plan:
1. **Update Package Import Statements:**
   - Update the import statements to match JavaScript conventions.

2. **Convert Bean Definitions:**
   - Convert `@Value` annotation to JavaScript equivalent for setting `jwtKey`.
   - Convert `@Bean` methods to regular function definitions in JavaScript.

3. **Security Configuration:**
   - Translate security configurations for HTTP requests to JavaScript syntax.
   - Convert `Customizer` and `CustomAuthenticationEntryPoint` to JavaScript functions.
   - Handle exceptions and form login configurations accordingly.

4. **JWT Authentication and Encoding:**
   - Translate JWT authentication converter, decoder, and encoder configurations.
   - Convert `NimbusJwtDecoder` methods to JavaScript-compatible functions.

5. **Secret Key Handling:**
   - Update the secret key handling logic using Base64 encoding to JavaScript.

Code Snippets:

Before Migration (Java):
```java
package com.example.momcare.config;

import com.example.momcare.util.Constant;
import com.example.momcare.util.SecurityUtil;
```

```
import com.nimbusds.jose.jwk.source.ImmutableSecret;
import com.nimbusds.jose.util.Base64;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.oauth2.jwt.*;
import
org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationConverter;
import org.springframework.security.oauth2.server.resource.authentication.JwtGrantedAuthoritie
sConverter;
import org.springframework.security.web.SecurityFilterChain;

import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

@Configuration
public class AppConfig {

    @Value("${backend.jwt.base64-secret}")
    private String jwtKey;

    // Other bean definitions and configurations
}
```

After Migration (JavaScript):
```javascript
// Import required modules and libraries

const jwtKey = process.env.BACKEND_JWT_BASE64_SECRET;

function passwordEncoder() {
    return bcrypt.hashSync('password', 10);
```

```
}

function filterChain(http, customAuthenticationEntryPoint, customAccessDeniedHandler) {
    // Security filter chain configuration in JavaScript
}

function jwtAuthenticationConverter() {
    // JWT authentication converter logic
}

function jwtDecoder() {
    // JWT decoder logic
}

function jwtEncoder() {
    // JWT encoder logic
}

function getSecretKey() {
    // Secret key handling logic in JavaScript
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling HTTP requests.
- bcrypt.js for password hashing.
- jsonwebtoken for JWT token operations.

Best Practices and Idioms in JavaScript:
- Use `const` or `let` for variable declarations.
- Avoid using synchronous operations for I/O tasks.
- Follow modular design patterns for better code organization.

Common Pitfalls to Avoid:
- Ensure proper error handling for asynchronous operations.
- Watch out for differences in data types and method signatures between Java and JavaScript.
- Test thoroughly after migration to catch any compatibility issues.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and security standards.

10. src/main/java/com/example/momcare/config/CustomAccessDeniedHandler.java

Java to JavaScript Migration Guide

Summary
The provided Java code is a custom Access Denied Handler class that implements the Spring Security `AccessDeniedHandler` interface. It handles access denied scenarios by returning a JSON response with a status code of 403 Forbidden.

Migration Plan
1. **Create a JavaScript equivalent of the custom Access Denied Handler class.**
2. **Update the syntax and libraries to align with JavaScript standards.**
3. **Ensure proper handling of access denied scenarios in JavaScript.**

Migration Steps
1. **Create a JavaScript module for the custom Access Denied Handler.**
2. **Update the syntax to JavaScript standards.**
3. **Handle access denied scenarios in JavaScript.**

Before/After Code Snippets
Before (Java)
```java
// Java code snippet
```

After (JavaScript)
```javascript
// JavaScript code snippet
```

Recommended Libraries/Frameworks in JavaScript
- **Express.js**: for building web applications in Node.js.
- **Axios**: for making HTTP requests in the browser and Node.js.
- **jsonwebtoken**: for handling JSON Web Tokens in Node.js applications.

Best Practices and Idioms in JavaScript
- **Use Promises or async/await for asynchronous operations.**

- **Follow ES6 syntax and features for cleaner code.**
- **Use arrow functions for concise function definitions.**

Common Pitfalls to Avoid
- **Avoid using synchronous operations in Node.js for performance reasons.**
- **Be mindful of callback hell when dealing with asynchronous code.**
- **Handle errors properly to prevent unexpected behavior.**

Migration Advice for Database/API Usage
- **Use libraries like Mongoose for MongoDB or Sequelize for SQL databases in Node.js.**
- **For APIs, consider using Express.js to create API endpoints.**
- **Ensure proper error handling and validation when interacting with databases or APIs.**

By following this migration guide, you can convert the provided Java code to JavaScript effectively while adhering to best practices and avoiding common pitfalls.

11. src/main/java/com/example/momcare/config/CustomAuthenticationEntryPoint.java

Java to JavaScript Migration Guide

Summary:
The given Java code represents a custom authentication entry point class that implements the `AuthenticationEntryPoint` interface. It handles unauthorized requests by returning a JSON response with status code 401.

Migration Plan:
1. **Dependencies**: Ensure you have the necessary libraries or frameworks in JavaScript for handling HTTP requests and JSON serialization.
2. **Class Structure**: Convert the class structure and method implementations to JavaScript syntax.
3. **Handling Responses**: Use appropriate JavaScript functions for setting response headers and writing JSON responses.
4. **Testing**: Thoroughly test the migrated code to ensure it behaves the same way as the original Java code.

Migration Steps:
1. **Dependencies**:
   - **Before** (Java):
   ```java
   import com.fasterxml.jackson.databind.ObjectMapper;
   import jakarta.servlet.ServletException;
   import jakarta.servlet.http.HttpServletRequest;
   import jakarta.servlet.http.HttpServletResponse;
   import org.springframework.security.core.AuthenticationException;
   import org.springframework.security.web.AuthenticationEntryPoint;
   import org.springframework.stereotype.Component;
   ```
   - **After** (JavaScript):
   Ensure you have libraries like `axios` for HTTP requests and `JSON.stringify` for JSON serialization.

2. **Class Structure**:
   - **Before** (Java):

```java
@Component
public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {
    private final ObjectMapper mapper;

    public CustomAuthenticationEntryPoint(ObjectMapper mapper) {
        this.mapper = mapper;
    }

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
AuthenticationException authException) throws IOException, ServletException {
        // Implementation
    }
}
```
- **After** (JavaScript):
```javascript
class CustomAuthenticationEntryPoint {
    constructor(mapper) {
        this.mapper = mapper;
    }

    commence(request, response, authException) {
        // Implementation
    }
}
```

3. **Handling Responses**:
   - **Before** (Java):
   ```java
   response.setContentType("application/json;charset=UTF-8");
   response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
   Response res = new Response("401", null, "Unauthorized");
   mapper.writeValue(response.getWriter(), res);
   ```
   - **After** (JavaScript):

```javascript
response.setHeader('Content-Type', 'application/json;charset=UTF-8');
response.status(401).json({ code: '401', data: null, message: 'Unauthorized' });
```

4. **Recommended Libraries**:
   - Use `axios` for making HTTP requests.
   - Use `express` for building web servers.

5. **Best Practices**:
   - Use ES6 classes and modules for better code organization.
   - Utilize Promises or async/await for handling asynchronous operations.

6. **Common Pitfalls**:
   - Ensure proper error handling for HTTP requests in JavaScript.
   - Pay attention to differences in object serialization between Java and JavaScript.

Conclusion:
By following this migration guide and paying attention to the details in each step, you should be able to successfully convert the given Java code to JavaScript while maintaining its functionality. Remember to test thoroughly to ensure the migrated code works as expected.

12. src/main/java/com/example/momcare/config/MailConfig.java

Migration Guide: Java to JavaScript for MailConfig

Summary:
The given Java code is a configuration class for setting up a mail sender using JavaMailSender in Spring Boot. It reads properties for mail configuration from application.properties file and sets up a JavaMailSender bean.

Migration Plan:
1. **Property Reading:** In JavaScript, properties can be read from a configuration file or environment variables.
2. **Setting up MailSender:** Use a library like `nodemailer` in JavaScript to set up a mail sender.
3. **Configuration:** Define the configuration for the mail sender including host, port, username, password, and properties.
4. **Bean Creation:** Create a function to initialize and return the mail sender object.

Before Migration:
```java
// Java code
// Include necessary imports and annotations
@Configuration
public class MailConfig {
    // Define properties with @Value annotation
    // Define JavaMailSender bean
}
```

After Migration:
```javascript
// JavaScript code
// Include necessary imports or require statements
// Define mail configuration
// Define mail sender function
```

Recommended Libraries/Frameworks in JavaScript:

- **Nodemailer:** For sending emails in Node.js.

Best Practices and Idioms in JavaScript:
- **Use ES6 Syntax:** Use arrow functions, const, let, etc.
- **Promises or Async/Await:** For handling asynchronous operations.
- **Modular Code:** Break code into reusable modules.

Common Pitfalls to Avoid:
- **Synchronous vs Asynchronous:** Be mindful of asynchronous operations in JavaScript.
- **Variable Scope:** Understand and manage variable scope properly.

Database/API Migration Advice:
- If interacting with a database or API, consider using libraries like `sequelize` for databases and `axios` for API calls.

By following this migration guide, you can successfully convert the given Java code for MailConfig to JavaScript while leveraging the best practices and libraries available in the JavaScript ecosystem.

13. src/main/java/com/example/momcare/config/VNPayConfig.java

Migration Guide: Java to JavaScript

VNPayConfig Class

Summary:
The VNPayConfig class contains methods for generating MD5 and SHA-256 hash values, creating a hash of all fields, generating a HMAC-SHA512 hash, retrieving the IP address, and generating a random number.

Migration Plan:
1. Convert the `VNPayConfig` class to a JavaScript module.
2. Use Node.js crypto module for cryptographic operations.
3. Modify method signatures to align with JavaScript conventions.
4. Update exception handling to JavaScript error handling.

Before/After Code Snippets:
```java
// Java
package com.example.momcare.config;
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Random;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.servlet.http.HttpServletRequest;

public class VNPayConfig {
    // Methods and constants
}
```

```

```javascript
// JavaScript
import crypto from 'crypto';

const secretKey = "BZBUPMSTNUAUQCAVQLXOEWUHIFTWLWWX";

export const md5 = (message) => {
    try {
        const hash = crypto.createHash('md5').update(message, 'utf8').digest('hex');
        return hash;
    } catch (error) {
        return "";
    }
};

export const sha256 = (message) => {
    try {
        const hash = crypto.createHash('sha256').update(message, 'utf8').digest('hex');
        return hash;
    } catch (error) {
        return "";
    }
};

export const hashAllFields = (fields) => {
    // Implementation
};

export const hmacSHA512 = (key, data) => {
    // Implementation
};

export const getIpAddress = () => {
    let ipAddress;
    try {
        ipAddress = "127.0.0.1";
```
```

```
    } catch (error) {
        ipAddress = "Invalid IP:" + error.message;
    }
    return ipAddress;
};

export const getRandomNumber = (len) => {
    // Implementation
};
```

Recommended Libraries/Frameworks:
- Node.js crypto module for cryptographic operations.

Best Practices and Idioms:
- Use ES6 arrow functions for method definitions.
- Utilize Node.js modules for cryptographic operations.

Common Pitfalls to Avoid:
- Ensure proper error handling in JavaScript.
- Watch for differences in encoding and hashing algorithms between Java and JavaScript.

Database/API Usage:
No database or API interaction detected in the provided code.

By following this migration guide, you can successfully convert the given Java code to JavaScript, maintaining functionality and best practices along the way.

14. src/main/java/com/example/momcare/config/WebSocketConfig.java

Migration Guide: Java to JavaScript for WebSocketConfig

Summary:
The Java code provided is a configuration class for setting up WebSocket handlers in a Spring application. It defines WebSocket endpoints for handling notifications and tutorials.

Migration Plan:
1. **WebSocket Configuration:**
   - Translate the WebSocketConfig class to a JavaScript module.
   - Use WebSocket API in JavaScript for handling WebSocket connections.

2. **Dependency Injection:**
   - Handle dependencies using JavaScript modules or ES6 classes.
   - Simulate constructor injection in JavaScript.

3. **Bean Configuration:**
   - Replace `@Bean` annotations with equivalent setup in JavaScript.

4. **WebSocket Handler Registration:**
   - Register WebSocket handlers for `/notification` and `/tutorial` endpoints.
   - Set allowed origins for WebSocket connections.

5. **WebSocket Container Configuration:**
   - Configure WebSocket container settings like session timeout.

Before/After Code Snippets:

Java Code:
```java
// Java code snippet provided
```

JavaScript Migration:
```javascript
// JavaScript equivalent migration
```

Recommended Libraries/Frameworks:
- **WebSocket API:** Use the native browser WebSocket API for handling WebSocket connections.
- **Express.js:** For building Node.js applications with WebSocket support.

Best Practices and Idioms:
- **Use ES6 Classes:** Utilize ES6 class syntax for defining WebSocket handlers.
- **Asynchronous Programming:** Use async/await for handling asynchronous operations in WebSocket communication.

Common Pitfalls to Avoid:
- **Dependency Injection:** Simulate constructor injection for handling dependencies in JavaScript.
- **WebSocket Security:** Ensure proper security measures for WebSocket connections to prevent vulnerabilities.

Database/API Migration Advice:
- If the application interacts with a database or API, ensure compatibility with JavaScript libraries like Axios for API requests.
- Consider using Node.js frameworks like Express.js for building API endpoints.

By following this migration guide, you can successfully convert the provided Java WebSocketConfig class to JavaScript while ensuring functionality and maintainability.

15. src/main/java/com/example/momcare/controllers/AuthController.java

Migration Guide from Java to JavaScript

AuthController.java

Summary:
The `AuthController` class handles user authentication and authorization through login and signup functionalities.

Migration Plan:
1. Convert the class definition and imports.
2. Migrate the constructor and dependency injection.
3. Translate the `@PostMapping` annotations to their JavaScript equivalents.
4. Replace Java specific security and authentication methods with JavaScript alternatives.
5. Update response handling and return statements.

Before Migration:
```java
package com.example.momcare.controllers;

// Imports omitted for brevity

@RestController
@RequestMapping("/auth")
public class AuthController {
    // Class fields and dependencies

    public AuthController(AuthenticationManagerBuilder authenticationManagerBuilder,
SecurityUtil securityUtil, PasswordEncoder passwordEncoder, UserService userService,
UserRepository userRepository, CheckAccount checkAccount, EmailService emailService) {
        // Constructor
    }

    @PostMapping("/login")
    public ResponseEntity<Response> login(@RequestBody UserLoginRequest
userLoginRequest) {
        // Login functionality
```

```
    }

    @PostMapping("/signup")
    public Response signUpAccount(@RequestBody UserSignUpRequest userSignUpRequest) {
        // Signup functionality
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
1. Express.js for building RESTful APIs.
2. bcryptjs for password hashing.
3. jsonwebtoken for token generation and authentication.

Best Practices and Idioms in JavaScript:
1. Use async/await for handling asynchronous operations.
2. Utilize middleware functions in Express for request handling.
3. Follow RESTful API design principles.

Common Pitfalls to Avoid:
1. Handling asynchronous code without proper error handling.
2. Not securely storing user passwords.
3. Overcomplicating the routing logic.

After Migration:
```javascript
const express = require('express');
const router = express.Router();
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');

class AuthController {
    constructor(userService, userRepository, checkAccount, emailService) {
        this.userService = userService;
        this.userRepository = userRepository;
        this.checkAccount = checkAccount;
        this.emailService = emailService;
    }
```

```
    router.post('/login', async (req, res) => {
        // Login functionality
    });

    router.post('/signup', async (req, res) => {
        // Signup functionality
    });
}

module.exports = AuthController;
```

Conclusion:
By following this migration guide, you can successfully convert the Java `AuthController` class to JavaScript, ensuring a seamless transition while leveraging the best practices and libraries of the JavaScript ecosystem.

16. src/main/java/com/example/momcare/controllers/BabyHealthIndexController.java

Migration Guide: Java to JavaScript

BabyHealthIndexController

Summary:
The `BabyHealthIndexController` in Java handles CRUD operations for baby health indices using RESTful endpoints.

Migration Plan:
1. Create a JavaScript file for the controller.
2. Use Express.js to handle routing and request/response handling.
3. Replace Java annotations with JavaScript equivalents.
4. Update method implementations to JavaScript syntax.
5. Use Promises or async/await for handling asynchronous operations.
6. Implement error handling using try/catch blocks.

Before/After Code Snippets:
Before (Java):
```java
@RestController
@RequestMapping("/babyindex")
public class BabyHealthIndexController {
    // Controller methods
}
```

After (JavaScript):
```javascript
const express = require('express');
const router = express.Router();

router.post('/new', (req, res) => {
    // Implementation for createIndex
});
```

```
// Define other REST endpoints

module.exports = router;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for routing and middleware handling.
- Axios for making HTTP requests to backend services.
- Jest for testing the JavaScript code.

Best Practices and Idioms in JavaScript:
- Use arrow functions for concise syntax.
- Follow RESTful conventions for API design.
- Utilize ES6 features like Promises and async/await for asynchronous operations.

Common Pitfalls to Avoid:
- Be mindful of differences in error handling between Java and JavaScript.
- Ensure proper data validation and sanitization to prevent security vulnerabilities.
- Test thoroughly to catch any migration-related issues.

Database/Backend Service Migration:
If the Java code interacts with a database or external API:
1. Use Node.js modules like `mysql` or `mongoose` for database interactions.
2. Refactor database queries to use asynchronous functions.
3. Update API calls to use `axios` or built-in `fetch` for HTTP requests.

Overall, the migration from Java to JavaScript involves adapting the code structure, syntax, and libraries to the Node.js ecosystem while maintaining the functionality and behavior of the original application.

17. src/main/java/com/example/momcare/controllers/DiaryController.java

Migration Guide from Java to JavaScript

Summary:
The provided Java code is a controller class `DiaryController` that handles requests related to diary entries. It includes methods for creating, finding, updating, and deleting diary entries, as well as retrieving top/newest entries based on different criteria.

Step-by-Step Migration Plan:
1. **Set Up Environment:**
   - Ensure you have Node.js installed for JavaScript development.

2. **Translate Controller Class:**
   - Create a new JavaScript file for the controller class.
   - Define the class and methods using JavaScript syntax.

3. **Handle Request Mapping:**
   - Update request mapping annotations to JavaScript route definitions.

4. **Update Service Calls:**
   - If `DiaryService` methods are not already implemented in JavaScript, create corresponding functions.

5. **Handle Responses:**
   - Update response objects and error handling to fit JavaScript conventions.

6. **Test and Debug:**
   - Test the JavaScript version of the controller with sample requests and ensure functionality matches the Java version.

Before/After Code Snippets:
Java (Before):
```java
@RestController
@RequestMapping("/diary")
public class DiaryController {
    // Controller methods here
```

}
```

#### JavaScript (After):
```javascript
const express = require('express');
const router = express.Router();

router.post('/new', (req, res) => {
    // Handle createDiary request
});

// Define other routes similarly

module.exports = router;
```

Recommended Libraries/Frameworks in JavaScript:
- **Express.js:** For building web applications and APIs.
- **Mongoose:** For MongoDB object modeling.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` for variable declarations.
- Follow asynchronous programming patterns using promises or async/await.
- Use arrow functions for cleaner syntax.

Common Pitfalls to Avoid:
- Not handling asynchronous operations properly.
- Inconsistent error handling between Java and JavaScript.
- Failing to install required dependencies for the JavaScript environment.

Migration Advice for Database/API Usage:
- If using a database, consider using Node.js libraries like Mongoose for MongoDB or Sequelize for SQL databases.
- Update API endpoints and request/response handling based on the chosen JavaScript framework.

By following this migration guide, you can successfully convert the provided Java code to
```

JavaScript while maintaining functionality and best practices in the new environment.

18. src/main/java/com/example/momcare/controllers/HandBookCategoryController.java

Java to JavaScript Migration Guide

HandBookCategoryController

Summary:
The Java code defines a controller class `HandBookCategoryController` that handles requests related to handbook categories. It has methods to find all categories and find categories by collection.

Migration Plan:
1. Set up a Node.js environment.
2. Create a JavaScript file for the controller.
3. Refactor the code to JavaScript syntax.
4. Use Express.js for routing.
5. Use Axios for making HTTP requests if needed.

Before Migration:
```java
package com.example.momcare.controllers;

import com.example.momcare.payload.response.Response;
import com.example.momcare.service.HandBookCategoryService;
import com.example.momcare.util.Constant;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HandBookCategoryController {

    private final HandBookCategoryService service;

    public HandBookCategoryController(HandBookCategoryService service) {
        this.service = service;
```

```
    }

    @GetMapping("/category")
    public Response findAll(){
        return new Response(HttpStatus.OK.getReasonPhrase(), service.findAllCategories(),
Constant.SUCCESS);
    }
    @GetMapping("/categoryByCollection")
    public Response findCategoryByCollection(@RequestParam ("collectionId")String
collectionId){
        return new Response(HttpStatus.OK.getReasonPhrase(),
service.findCategoriesByCollection(collectionId), Constant.SUCCESS);
    }
}
```

After Migration:
```javascript
const express = require('express');
const router = express.Router();

class HandBookCategoryController {

    constructor(service) {
        this.service = service;
    }

    router.get('/category', (req, res) => {
        const response = {
            message: 'OK',
            data: this.service.findAllCategories(),
            status: 'SUCCESS'
        };
        res.status(200).json(response);
    });

    router.get('/categoryByCollection', (req, res) => {
        const collectionId = req.query.collectionId;
```

```
        const response = {
            message: 'OK',
            data: this.service.findCategoriesByCollection(collectionId),
            status: 'SUCCESS'
        };
        res.status(200).json(response);
    });
}

module.exports = HandBookCategoryController;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for routing.
- Axios for making HTTP requests.

Best Practices and Idioms in JavaScript:
- Use arrow functions for concise syntax.
- Utilize object destructuring for cleaner code.
- Use Promises or async/await for asynchronous operations.

Common Pitfalls to Avoid:
- Be mindful of the differences between Java and JavaScript syntax.
- Handle asynchronous operations correctly, especially when dealing with data fetching.

Database/API Migration:
- If the `HandBookCategoryService` interacts with a database, consider using a Node.js ORM like Sequelize or Mongoose for database operations.
- For APIs, ensure that the service methods are adapted to work with Node.js HTTP modules or Axios for making API requests.

19. src/main/java/com/example/momcare/controllers/HandBookCollectionController.java

Java to JavaScript Migration Guide

HandBookCollectionController

Summary:
The `HandBookCollectionController` is a Java class responsible for handling HTTP requests related to collections. It uses a `HandBookCollectionService` to retrieve collections and return a response.

Migration Plan:
1. **Create a JavaScript class for the controller.**
2. **Implement the `findAll` method to return a response.**
3. **Update import statements for libraries.**
4. **Consider using Express.js for handling HTTP requests.**

Before Migration (Java):
```java
package com.example.momcare.controllers;

import com.example.momcare.payload.response.Response;
import com.example.momcare.service.HandBookCollectionService;

import com.example.momcare.util.Constant;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HandBookCollectionController {
    HandBookCollectionService service;

    public HandBookCollectionController(HandBookCollectionService service) {
        this.service = service;
    }
```

```
    @GetMapping("/collections")
    public Response findAll(){
        return new Response(HttpStatus.OK.getReasonPhrase(), service.findAllCollection(),
Constant.SUCCESS);
    }
}
```

After Migration (JavaScript):
```javascript
// Import necessary libraries and modules
const HttpStatus = require('http-status-codes');
const HandBookCollectionService = require('./HandBookCollectionService');

class HandBookCollectionController {
    constructor(service) {
        this.service = service;
    }

    findAll(req, res) {
        const collections = this.service.findAllCollection();
        const response = {
            status: HttpStatus.OK,
            message: HttpStatus.getStatusText(HttpStatus.OK),
            data: collections,
            success: Constant.SUCCESS
        };
        res.json(response);
    }
}

module.exports = HandBookCollectionController;
```

Recommended Libraries/Frameworks:
- **Express.js:** For handling HTTP requests and routing.
- **http-status-codes:** For accessing HTTP status codes and messages.

Best Practices in JavaScript:
- Use `const` and `let` for variable declarations.
- Use Promises or async/await for handling asynchronous operations.

Common Pitfalls to Avoid:
- Ensure proper error handling in asynchronous operations.
- Be mindful of the differences in handling HTTP requests between Java and JavaScript.

Database/API Migration:
- If the `HandBookCollectionService` interacts with a database, consider using an ORM like Sequelize for JavaScript.
- For API calls, use libraries like Axios or Fetch for making HTTP requests.

By following this migration guide, you can successfully convert the given Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

20. src/main/java/com/example/momcare/controllers/HandBookController.java

Migration Guide from Java to JavaScript

HandBookController

Summary:
The `HandBookController` class in Java is a REST controller that handles various API endpoints related to handbooks. It interacts with a `HandBookService` to retrieve data and return responses.

Migration Plan:
1. **Convert Class Structure:**
   - Convert the class structure to a JavaScript ES6 class.

2. **Convert Constructor:**
   - Convert the constructor function to a JavaScript constructor method.

3. **Convert Request Mapping:**
   - Convert `@GetMapping` annotations to corresponding JavaScript route handlers.

4. **Handle Request Parameters:**
   - Handle request parameters appropriately in JavaScript.

5. **Response Handling:**
   - Create response objects similar to Java's `Response` class.

6. **Inject Service Dependency:**
   - Inject the `HandBookService` dependency using a similar approach.

Code Snippets:

Before (Java):
```java
@RestController
public class HandBookController {
```

```
    HandBookService service;

    public HandBookController(HandBookService service) {
        this.service = service;
    }

    @GetMapping("/handbookall")
    public Response findAllByCategory(@RequestParam("idCategory") String idCategory) {
        return new Response(HttpStatus.OK.getReasonPhrase(),
service.findHandBookByCategoryService(idCategory), Constant.SUCCESS);
    }

    // Other endpoint methods...
}
```

After (JavaScript):
```javascript
class HandBookController {
    constructor(service) {
        this.service = service;
    }

    async findAllByCategory(req, res) {
        const idCategory = req.query.idCategory;
        const data = await this.service.findHandBookByCategoryService(idCategory);
        const response = { status: 200, message: 'OK', data };
        res.json(response);
    }

    // Other endpoint methods...
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling routes and middleware.
- Axios for making HTTP requests.
- Jest for unit testing.

Best Practices and Idioms in JavaScript:
- Use async/await for asynchronous operations instead of callbacks.
- Follow RESTful API conventions for route naming and request handling.
- Use ES6 features like arrow functions and template literals.

Common Pitfalls to Avoid:
- Be cautious of data type conversions between Java and JavaScript.
- Handle exceptions and errors properly in asynchronous code.
- Ensure proper validation of request parameters to prevent security vulnerabilities.

Database and API Migration:

Database Usage:
If the `HandBookService` interacts with a database, consider migrating the database access layer to a JavaScript ORM like Sequelize for relational databases or Mongoose for MongoDB.

API Usage:
If the `HandBookService` interacts with external APIs, ensure that the equivalent API calls are made using libraries like Axios in JavaScript. Update the endpoints and data handling accordingly.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and adhering to best practices in the JavaScript ecosystem.

21. src/main/java/com/example/momcare/controllers/MenuCategoryController.java

Java to JavaScript Migration Guide

MenuCategoryController.java

Summary:
The `MenuCategoryController` class in Java is a REST controller that handles requests related to menu categories. It uses `MenuCategoryService` to fetch menu categories and returns a response object.

Migration Plan:
1. Create a new JavaScript file named `MenuCategoryController.js`.
2. Import necessary modules and define the `MenuCategoryService`.
3. Define the `MenuCategoryController` class and implement the `findAll` method to return a response.

Before Migration:
```java
package com.example.momcare.controllers;

import com.example.momcare.payload.response.Response;
import com.example.momcare.service.MenuCategoryService;

import com.example.momcare.util.Constant;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MenuCategoryController {

    MenuCategoryService service;

    public MenuCategoryController(MenuCategoryService service) {
        this.service = service;
    }
```

```
    @GetMapping("/menuCategory")
    public Response findAll(){
        return new Response(HttpStatus.OK.getReasonPhrase(), service.findAllMenuCategory(),
Constant.SUCCESS);
    }
}
```

After Migration:
```javascript
// MenuCategoryController.js

const Response = require('./response');
const MenuCategoryService = require('./menuCategoryService');
const Constant = require('./constant');

class MenuCategoryController {
    constructor(service) {
        this.service = service;
    }

    findAll() {
        return new Response(HttpStatus.OK.getReasonPhrase(),
this.service.findAllMenuCategory(), Constant.SUCCESS);
    }
}

module.exports = MenuCategoryController;
```

Recommended Libraries/Frameworks:
- Express.js for handling HTTP requests in Node.js.
- Axios for making HTTP requests to external services.

Best Practices in JavaScript:
- Use ES6 features like classes and arrow functions.
- Use `const` and `let` instead of `var` for variable declarations.

- Follow asynchronous programming using Promises or async/await.

Common Pitfalls to Avoid:
- Be mindful of the asynchronous nature of JavaScript while handling API calls.
- Ensure proper error handling to prevent uncaught exceptions.

Conclusion:
By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and best practices in the new environment.

22. src/main/java/com/example/momcare/controllers/MenuController.java

Java to JavaScript Migration Guide

MenuController

Summary:
The `MenuController` class in Java is a REST controller that handles requests related to menus. It has two methods for finding menus by category and by ID.

Migration Plan:
1. Remove Java-specific annotations and imports.
2. Implement similar functionality in JavaScript using Express.js.
3. Use appropriate libraries for handling HTTP requests and responses.

Before/After Code Snippets:
Java:
```java
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
```

JavaScript:
```javascript
const express = require('express');
const app = express();
const HttpStatus = require('http-status-codes');
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling HTTP requests.
- http-status-codes for HTTP status code constants.

Best Practices and Idioms in JavaScript:
- Use arrow functions for defining route handlers.

- Use `const` or `let` instead of `var` for variable declarations.

Common Pitfalls to Avoid:
- Incorrectly handling asynchronous code.
- Not properly handling error scenarios.

MenuService Migration:
If `MenuService` contains business logic or database interactions, migrate it to JavaScript as well using appropriate libraries like Sequelize for database interactions.

Database/API Migration:
If the code interacts with a database or external APIs, consider using Node.js libraries like Sequelize for databases or Axios for API calls.

By following this migration guide, you can successfully convert the Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

23. src/main/java/com/example/momcare/controllers/MomHealthIndexController.java

Migration Guide: Java to JavaScript

MomHealthIndexController

Summary:
The `MomHealthIndexController` is a Java class that handles RESTful API endpoints related to managing health indexes for moms. It uses Spring Boot annotations for mapping HTTP requests to methods in the controller.

Migration Plan:
1. Convert the class definition and annotations to JavaScript syntax using Node.js or Express.js.
2. Rewrite the methods to handle HTTP requests and responses in JavaScript.
3. Update the imports to use JavaScript modules or libraries.
4. Use asynchronous functions for handling asynchronous operations.
5. Consider using a framework like Express.js for routing and handling HTTP requests.

Code Snippets:

Before (Java):
```java
@RestController
public class MomHealthIndexController {
    // Controller methods
}
```

After (JavaScript):
```javascript
const express = require('express');
const router = express.Router();

// Define routes for MomHealthIndexController
```

Recommended Libraries/Frameworks:

- Express.js: A popular web framework for Node.js that simplifies routing and handling HTTP requests.
- Axios: A promise-based HTTP client for making API requests in JavaScript.

Best Practices in JavaScript:
- Use asynchronous functions (async/await) for handling asynchronous operations.
- Use arrow functions for concise syntax and lexical scoping.
- Follow naming conventions and module structure for better organization.

Common Pitfalls to Avoid:
- Mixing synchronous and asynchronous code can lead to unexpected behavior.
- Not handling errors properly in asynchronous functions can result in uncaught exceptions.

MomHealthIndexService

Summary:
The `MomHealthIndexService` is a Java class responsible for handling business logic related to mom health indexes, such as creating, updating, deleting, and retrieving indexes.

Migration Plan:
1. Convert the class definition and methods to JavaScript.
2. Implement the business logic using JavaScript syntax.
3. Update any dependencies or external services used in the service.
4. Consider using Promises or async/await for handling asynchronous operations.

Code Snippets:

Before (Java):
```java
public class MomHealthIndexService {
    // Service methods
}
```

After (JavaScript):
```javascript
class MomHealthIndexService {
    // Service methods
```

```
}
```

Recommended Libraries/Frameworks:
- None specific to mention, but consider using Jest for testing JavaScript code.

Best Practices in JavaScript:
- Use ES6 features like classes, modules, and destructuring for cleaner code.
- Avoid callback hell by using Promises or async/await for asynchronous operations.

Common Pitfalls to Avoid:
- Mutating objects directly can lead to unexpected side effects.
- Not handling Promise rejections can cause unhandled promise rejections.

Overall Recommendations:
- Use Node.js with Express.js for building the API server.
- Consider using a data validation library like Joi for input validation.
- Implement unit tests using Jest or Mocha for ensuring code quality and reliability.

By following this migration guide, you can successfully convert the Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

24. src/main/java/com/example/momcare/controllers/MusicController.java

Migration Guide: Java to JavaScript

MusicController

Summary:
The `MusicController` is a Java class that handles HTTP requests related to music. It uses the `MusicService` to fetch music data.

Migration Plan:
1. Create a new JavaScript file for the `MusicController`.
2. Use Express.js to handle HTTP requests.
3. Implement equivalent functions for `getRandomMusic`, `getMusicByCategory`, and `getCategory`.
4. Update the response format to match JavaScript conventions.
5. Update import statements and remove unnecessary annotations.

Before Migration (Java):
```java
// Java code
package com.example.momcare.controllers;

import com.example.momcare.payload.response.Response;
import com.example.momcare.service.MusicService;
import com.example.momcare.util.Constant;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MusicController {

    MusicService musicService;

    public MusicController(MusicService musicService) {
```

```
        this.musicService = musicService;
    }

    @GetMapping("music/random")
    public Response getRandomMusic(){
        return new Response(HttpStatus.OK.getReasonPhrase(),
musicService.getRandomMusic(), Constant.SUCCESS);
    }

    @GetMapping("music/category")
    public Response getMusicByCategory(@RequestParam String category){
        return new Response(HttpStatus.OK.getReasonPhrase(),
musicService.getMusicByCategory(category), Constant.SUCCESS);
    }

    @GetMapping("music/allcategory")
    public Response getCategory(){
        return new Response(HttpStatus.OK.getReasonPhrase(),  musicService.getCategory(),
Constant.SUCCESS);
    }
}
```

After Migration (JavaScript):
```javascript
// JavaScript code
const express = require('express');
const MusicService = require('./MusicService');

const router = express.Router();
const musicService = new MusicService();

router.get('/music/random', (req, res) => {
    const response = {
        status: 200,
        message: 'OK',
        data: musicService.getRandomMusic(),
        success: true
```

```
    };
    res.json(response);
});

router.get('/music/category', (req, res) => {
    const category = req.query.category;
    const response = {
        status: 200,
        message: 'OK',
        data: musicService.getMusicByCategory(category),
        success: true
    };
    res.json(response);
});

router.get('/music/allcategory', (req, res) => {
    const response = {
        status: 200,
        message: 'OK',
        data: musicService.getCategory(),
        success: true
    };
    res.json(response);
});

module.exports = router;
```

Recommended Libraries/Frameworks:
- Express.js: for handling HTTP requests and routing.
- Mongoose: for database operations if needed.

Best Practices and Idioms in JavaScript:
- Use ES6 syntax features like arrow functions and destructuring.
- Follow Node.js module patterns for importing and exporting modules.

Common Pitfalls to Avoid:
- Make sure to handle asynchronous operations properly using callbacks, promises, or async/await.

- Watch out for differences in error handling between Java and JavaScript.

Conclusion:
By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and adhering to best practices in the JavaScript ecosystem.

25. src/main/java/com/example/momcare/controllers/NotificationController.java

Migration Guide: Java to JavaScript

NotificationController

Summary:
The `NotificationController` class in Java handles HTTP requests related to notifications, such as saving, marking as read, getting notifications by receiver ID, getting unread notifications, and deleting notifications.

Migration Plan:
1. **Create a new JavaScript file for NotificationController.**
2. **Convert class declaration and imports:**
   - Replace `package` declaration with module exports.
   - Replace imports with equivalent in JavaScript.
3. **Convert class definition:**
   - Convert class declaration to JavaScript class syntax.
   - Implement constructor for the class.
4. **Convert request mappings:**
   - Update annotations for HTTP methods.
   - Convert method signatures to JavaScript syntax.
5. **Handle exceptions and responses:**
   - Update response handling logic.
   - Handle errors appropriately in JavaScript.

Before Migration:
```java
// Java code
package com.example.momcare.controllers;

// Imports

@RestController
@RequestMapping("/notifications")
public class NotificationController {
    // Class definition
```

```
    @PostMapping
    public Response save(@RequestBody NotificationRequest notificationRequest) {
        // Method implementation
    }

    // Other methods
}
```

After Migration:
```javascript
// JavaScript code
// Equivalent imports

class NotificationController {
    constructor(notificationService) {
        this.notificationService = notificationService;
    }

    save(notificationRequest) {
        // Method implementation
    }

    // Other methods
}

module.exports = NotificationController;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling HTTP requests.
- Axios for making HTTP requests to external services.

Best Practices and Idioms in JavaScript:
- Use ES6 classes for defining classes.
- Utilize Promises or async/await for handling asynchronous operations.

Common Pitfalls to Avoid:
- Ensure proper error handling to prevent uncaught exceptions.
- Pay attention to the differences in handling HTTP requests between Java and JavaScript.

Additional Migration Steps:
- **Database Usage:**
  - If database operations are involved, consider using an ORM like Sequelize for interacting with databases in JavaScript.
- **API Calls:**
  - For external API calls, use libraries like Axios to make HTTP requests.
- **Testing:**
  - Use Jest or Mocha for unit testing the JavaScript code.

By following this migration guide, you can successfully convert the given Java code for the `NotificationController` to JavaScript.

26. src/main/java/com/example/momcare/controllers/PeriodicCheckController.java

Migration Guide: Java to JavaScript

Summary:
The provided Java code is a controller class (`PeriodicCheckController`) that handles HTTP requests for periodic checks. It uses Spring annotations like `@RestController` and `@GetMapping` to define endpoints and handle requests. The controller interacts with a service class (`PeriodicCheckService`) to retrieve periodic check data.

Migration Plan:
1. **Convert Annotations**: Replace Spring annotations with equivalent Express.js routing methods in JavaScript.
2. **Rewrite Constructor**: Modify the constructor to work with JavaScript classes.
3. **Translate Method Logic**: Adapt method implementations for JavaScript syntax.
4. **Handle Dependencies**: Address any dependencies on external libraries or services.
5. **Test Endpoints**: Verify that endpoints work as expected post-migration.

Migration Steps:

1. Convert Annotations:
Replace Spring annotations with Express.js routing methods.

Before:
```java
@RestController
@GetMapping("/periodiccheck/all")
@GetMapping("/periodiccheck")
```

After:
```javascript
app.get('/periodiccheck/all', (req, res) => {
  // Handler logic for /periodiccheck/all endpoint
});

app.get('/periodiccheck', (req, res) => {
```

```
  // Handler logic for /periodiccheck endpoint
});
```

2. Rewrite Constructor:
Modify the constructor to a JavaScript class constructor.

Before:
```java
public PeriodicCheckController(PeriodicCheckService periodicCheckService) {
    this.periodicCheckService = periodicCheckService;
}
```

After:
```javascript
class PeriodicCheckController {
  constructor(periodicCheckService) {
    this.periodicCheckService = periodicCheckService;
  }

  // Methods here
}
```

3. Translate Method Logic:
Adapt method implementations for JavaScript.

Before:
```java
public Response getAll (){
    return new Response(HttpStatus.OK.getReasonPhrase(), periodicCheckService.getAll(),
Constant.SUCCESS);
}

public Response getByWeek (@RequestParam int weekFrom){
    return new Response(HttpStatus.OK.getReasonPhrase(),
periodicCheckService.findByWeekFromService(weekFrom), Constant.SUCCESS);
```

}
```

**After:**
```javascript
getAll(req, res) {
  return new Response('OK', this.periodicCheckService.getAll(), 'SUCCESS');
}

getByWeek(req, res) {
  const weekFrom = req.query.weekFrom;
  return new Response('OK', this.periodicCheckService.findByWeekFromService(weekFrom),
'SUCCESS');
}
```

4. Handle Dependencies:
Ensure that the `PeriodicCheckService` is properly imported and integrated with the controller.

5. Test Endpoints:
Verify that the endpoints work correctly by testing API calls using tools like Postman.

Recommended Libraries/Frameworks in JavaScript:
- **Express.js**: For routing and handling HTTP requests.
- **Mongoose**: For interacting with MongoDB if needed.
- **Jest/Mocha**: For unit testing.

Best Practices and Idioms in JavaScript:
- Use ES6 features like arrow functions and classes for cleaner code.
- Follow RESTful API conventions for endpoint design.
- Utilize async/await for handling asynchronous operations.

Common Pitfalls to Avoid:
- Beware of differences in parameter handling between Java and JavaScript.
- Ensure that data types are handled correctly during conversion.
- Pay attention to error handling and middleware usage in Express.js.

By following this migration guide, you can successfully convert the given Java code to
```

JavaScript while maintaining functionality and best practices.

27. src/main/java/com/example/momcare/controllers/SearchController.java

Migration Guide: Java to JavaScript

SearchController.java

Summary:
The Java code defines a SearchController class that handles searching for HandBooks, SocialPosts, and Users based on a keyword.

Migration Plan:
1. Create a JavaScript file for SearchController.
2. Define the SearchController class in JavaScript.
3. Implement search functions for HandBook, SocialPost, and User.
4. Use appropriate JavaScript syntax for request handling and response.
5. Include necessary libraries or frameworks for routing and handling HTTP requests.

Before Migration:
```java
// Java code for SearchController
// ...
```

After Migration:
```javascript
// JavaScript code for SearchController
// ...
```

Recommended Libraries/Frameworks:
- Express.js for routing
- Axios for making HTTP requests
- Jest for testing

Best Practices in JavaScript:
- Use async/await for handling asynchronous operations.
- Use arrow functions for concise syntax.

- Follow RESTful API design principles.

Common Pitfalls to Avoid:
- Mixing synchronous and asynchronous code.
- Not handling errors properly in asynchronous operations.

Overall Migration Advice:
- Pay attention to differences in syntax between Java and JavaScript.
- Utilize JavaScript's asynchronous nature for better performance.
- Test thoroughly to ensure functionality remains intact after migration.

28. src/main/java/com/example/momcare/controllers/SocialCommentController.java

Migration Guide from Java to JavaScript

Summary:
The given Java code represents a controller class `SocialCommentController` that handles various CRUD operations related to social comments such as fetching all comments, creating a new comment, updating a comment, adding/deleting reactions on comments, and deleting a comment.

Step-by-Step Migration Plan:
1. **Setup Environment:**
   - Ensure Node.js is installed for JavaScript development.
   - Use a code editor like Visual Studio Code for development.

2. **Dependencies:**
   - Identify equivalent libraries or frameworks in JavaScript for Spring dependencies.
   - Use Express.js for routing and handling HTTP requests.

3. **Code Migration:**
   - Convert Java syntax to JavaScript syntax.
   - Update import statements to require statements in JavaScript.
   - Replace annotations with corresponding Express.js methods.

4. **Testing:**
   - Test each API endpoint using tools like Postman to ensure functionality.

Before/After Code Snippets:

Java Code Snippet:
```java
package com.example.momcare.controllers;

// import statements

@RestController
@RequestMapping("/socialcomment")
```

```
public class SocialCommentController {

    // controller methods
}
```

JavaScript Equivalent:
```javascript
const express = require('express');
const router = express.Router();

router.get('/socialcomment/all', (req, res) => {
    // handler for getAll method
});

// define other routes

module.exports = router;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling HTTP requests and routing.
- Sequelize or Mongoose for database operations.
- Jest or Mocha for testing.

Best Practices and Idioms in JavaScript:
- Use asynchronous functions for handling operations like database queries.
- Follow RESTful API design principles for route naming and structure.
- Utilize ES6 features like arrow functions, destructuring, and promises.

Common Pitfalls to Avoid:
- Handling asynchronous operations correctly to prevent callback hell.
- Proper error handling to avoid uncaught exceptions.
- Managing dependencies and ensuring they are up-to-date.

Migration Advice for Database/API Usage:
- Utilize Sequelize or Mongoose for ORM operations in JavaScript.
- Update database connection configurations to match JavaScript frameworks.

This guide provides a structured approach to migrating the given Java code to JavaScript, focusing on maintaining functionality while adhering to JavaScript best practices and idioms.

29. src/main/java/com/example/momcare/controllers/SocialPostController.java

Migration Guide: Java to JavaScript

SocialPostController

Summary:
The `SocialPostController` class is a controller in a Java Spring application responsible for handling HTTP requests related to social posts.

Migration Plan:
1. **Dependencies**: Ensure necessary libraries like Express.js are installed.
2. **Class Structure**: Convert class structure to JavaScript class syntax.
3. **Annotations**: Replace Spring annotations with Express.js route handling.
4. **Request Handling**: Modify request handling methods to work with JavaScript async/await.

Before/After Code Snippets:
Java:
```java
@RestController
@RequestMapping("/socialpost")
public class SocialPostController {
    // Controller methods
}
```

JavaScript:
```javascript
const express = require('express');
const router = express.Router();

class SocialPostController {
    // Controller methods
}

module.exports = SocialPostController;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for routing
- Mongoose for MongoDB integration

Best Practices and Idioms in JavaScript:
- Use async/await for asynchronous operations
- Use arrow functions for concise syntax

Common Pitfalls to Avoid:
- Handling asynchronous operations incorrectly
- Not setting up proper error handling middleware

getAllByUser

Summary:
The `getAllByUser` method retrieves all social posts by a specific user.

Migration Plan:
1. **Request Handling**: Modify method signature for Express.js request handling.
2. **Service Call**: Update method to call the equivalent JavaScript service function.
3. **Response Handling**: Return response using Express.js response object.

Before/After Code Snippets:
Java:
```java
@GetMapping("/getallbyuser")
public Response getAllByUser(@RequestParam String userId) {
    // Method implementation
}
```

JavaScript:
```javascript
async getAllByUser(req, res) {
    const userId = req.query.userId;
    try {
        const response = await socialPostService.getAllByUserService(userId);
```

```
        res.status(200).json({ message: 'Success', data: response });
    } catch (error) {
        res.status(417).json({ message: 'Error', data: [], error: error.message });
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for route handling
- Axios for HTTP requests

Best Practices and Idioms in JavaScript:
- Use destructuring for request parameters
- Handle errors with proper status codes and messages

Common Pitfalls to Avoid:
- Mixing synchronous and asynchronous code
- Not handling promise rejections

Conclusion:
By following this migration guide, you can successfully convert the Java code to JavaScript, ensuring functionality is preserved while leveraging the benefits of JavaScript's ecosystem.

30. src/main/java/com/example/momcare/controllers/TrackingController.java

Java to JavaScript Migration Guide

TrackingController

Summary:
The `TrackingController` class is a Java controller responsible for handling tracking related HTTP requests.

Migration Plan:
1. Create a JavaScript class to serve as the controller.
2. Use Express.js to handle routing and request handling.
3. Map the Java controller methods to corresponding Express.js routes.

Before/After Code Snippets:
Java (Before):
```java
@RestController
public class TrackingController {

    private final TrackingService service;

    public TrackingController(TrackingService service) {
        this.service = service;
    }

    @GetMapping("/trackingall")
    public Response trackingall(){
        return new Response(HttpStatus.OK.getReasonPhrase(), service.trackingAll(), Constant.SUCCESS);
    }
    @GetMapping("/tracking")
    public Response trackingWeek(@RequestParam("week") int week){
        return new Response(HttpStatus.OK.getReasonPhrase(), service.trackingWeek(week), Constant.SUCCESS);
    }
```

```
}
```

JavaScript (After):
```javascript
const express = require('express');
const router = express.Router();
const Response = require('./Response'); // Assuming Response class is implemented

class TrackingController {
    constructor(service) {
        this.service = service;
    }

    router.get('/trackingall', (req, res) => {
        const response = new Response('OK', this.service.trackingAll(), 'SUCCESS');
        res.status(200).json(response);
    });

    router.get('/tracking', (req, res) => {
        const week = req.query.week;
        const response = new Response('OK', this.service.trackingWeek(week), 'SUCCESS');
        res.status(200).json(response);
    });
}

module.exports = TrackingController;
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for routing and handling HTTP requests.
- Body-parser for parsing request data.

Best Practices and Idioms in JavaScript:
- Use ES6 classes for defining controllers.
- Utilize arrow functions for cleaner code.

Common Pitfalls to Avoid During Migration:

- Pay attention to differences in request handling between Java and JavaScript.
- Ensure the correct mapping of routes and request parameters.

31. src/main/java/com/example/momcare/controllers/UserController.java

Java to JavaScript Migration Guide

UserController

Summary:
The UserController class defines RESTful API endpoints for user-related operations like updating account information, changing passwords, handling login, and profile management.

Migration Plan:
1. Convert the class definition from Java to JavaScript.
2. Update the import statements to JavaScript equivalent.
3. Refactor the methods to use JavaScript syntax and idioms.
4. Replace Java specific libraries and annotations with JavaScript equivalents.

Code Snippets:
Before:
```java
@RestController
public class UserController {
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PutMapping("/user/update")
    public Response updateAccount(@RequestBody UserRequest userRequest) {
        // Method implementation
    }

    // Other methods
}
```

After:
```javascript
```

```
const UserService = require('./UserService');

class UserController {
    constructor(userService) {
        this.userService = userService;
    }

    updateAccount(userRequest) {
        // Method implementation
    }

    // Other methods
}

module.exports = UserController;
```

Recommended Libraries/Frameworks:
- Express.js for building RESTful APIs in JavaScript.
- Axios for making HTTP requests in Node.js.

Best Practices in JavaScript:
- Use `const` and `let` for variable declarations.
- Follow asynchronous programming using Promises or async/await.

Common Pitfalls to Avoid:
- Handling asynchronous operations incorrectly.
- Not handling error responses properly.

Migration Advice for Database/API:
- Replace Spring Data JPA with Sequelize for ORM in Node.js.
- Use Express.js for defining API routes and handling requests.

The rest of the code can be migrated in a similar manner following the outlined steps.

32. src/main/java/com/example/momcare/controllers/UserStoryController.java

Migration Guide: Java to JavaScript

UserStoryController

Summary:
The `UserStoryController` class in Java is responsible for handling HTTP requests related to user stories such as creating, deleting, and retrieving user stories.

Migration Plan:
1. **Setup Environment:**
   - Set up Node.js environment for JavaScript development.
   - Use a package manager like npm or yarn to manage dependencies.

2. **Convert Class Structure:**
   - Create a JavaScript class for `UserStoryController`.
   - Use ES6 class syntax for defining classes.

3. **Handle HTTP Requests:**
   - Use Express.js framework for handling HTTP requests in JavaScript.
   - Define route handlers for POST, PUT, and GET requests.

4. **Call UserStoryService Methods:**
   - Convert method calls to the `userStoryService` instance.

Code Snippets:
Before (Java):
```java
@RestController
@RequestMapping("/userStory")
public class UserStoryController {
    // Controller methods
}
```

After (JavaScript):

```javascript
const express = require('express');
const router = express.Router();

class UserStoryController {
    // Controller methods
}

module.exports = UserStoryController;
```

Recommended Libraries/Frameworks:
- Express.js: For building web applications and APIs in Node.js.
- Axios: For making HTTP requests in Node.js.

Best Practices in JavaScript:
- Use Promises or async/await for handling asynchronous operations.
- Use arrow functions for concise syntax and lexical `this`.

Common Pitfalls to Avoid:
- Beware of the differences in handling asynchronous operations between Java and JavaScript.
- Properly handle error responses in Express.js route handlers.

Conclusion:
By following this migration guide, you can successfully convert the Java code for `UserStoryController` to JavaScript using Express.js for handling HTTP requests. Remember to consider best practices and common pitfalls to ensure a smooth migration process.

33. src/main/java/com/example/momcare/controllers/VNPayController.java

Java to JavaScript Migration Guide

VNPayController.java

Summary:
The `VNPayController` class in Java handles payment creation and success responses using VNPayConfig and UserService.

Migration Plan:
1. **Update import statements** to reflect JavaScript conventions.
2. **Translate request mappings** to JavaScript syntax.
3. **Replace Java-specific functions** with JavaScript equivalents.
4. **Implement response handling** using JavaScript frameworks or vanilla JavaScript.

Before/After Code Snippets:
Before (Java):
```java
@RestController
public class VNPayController {
    @Autowired
    private UserService userService;
    @GetMapping("payment/create")
    public Response create(@RequestParam String id) throws UnsupportedEncodingException {
        // Payment creation logic
    }

    @GetMapping(value = "payment/success", produces = MediaType.TEXT_HTML_VALUE)
    public String successPayment(@RequestParam String id){
        // Success payment logic
    }
}
```

After (JavaScript):
```javascript
```

```
const userService = require('./UserService'); // Import UserService
const express = require('express');
const router = express.Router();

router.get('/payment/create', (req, res) => {
    const { id } = req.query;
    // Payment creation logic
});

router.get('/payment/success', (req, res) => {
    const { id } = req.query;
    // Success payment logic
});

module.exports = router;
```

Recommended Libraries/Frameworks in JavaScript:
- **Express.js** for web server implementation.
- **Axios** or **Fetch API** for HTTP requests.
- **CryptoJS** for cryptographic operations.

Best Practices and Idioms in JavaScript:
- Use **ES6 syntax** for improved readability and functionality.
- Implement **error handling** for asynchronous operations.
- Leverage **async/await** for asynchronous code.

Common Pitfalls to Avoid:
- **Not handling promises** properly can lead to unexpected behavior.
- **Ignoring asynchronous nature** of certain operations may cause issues.
- **Incorrect data type handling**, especially with strings and numbers.

Conclusion:
By following this migration guide, you can effectively convert the Java code to JavaScript, ensuring functionality and performance are maintained during the process.

34. src/main/java/com/example/momcare/controllers/VideoController.java

Migration Guide from Java to JavaScript

Summary:
The provided Java code is a controller class for handling video-related requests. It uses the Spring framework for RESTful web services.

Step-by-Step Migration Plan:
1. **Setup Node.js Environment:**
   - Install Node.js and npm.
   - Create a new Node.js project.

2. **Translate Dependencies:**
   - Identify equivalent libraries in JavaScript.
   - Update project dependencies.

3. **Rewrite Controller Logic:**
   - Convert Java syntax to JavaScript.
   - Implement equivalent functionality in JavaScript.

4. **Test and Validate:**
   - Ensure the migrated code works correctly.
   - Test API endpoints using tools like Postman.

Before/After Code Snippets:
Java Code (Before):
```java
package com.example.momcare.controllers;
// Import statements

@RestController
public class VideoController {
    // Controller logic
}
```

JavaScript Equivalent (After):
```javascript
const express = require('express');
const router = express.Router();
// Controller logic
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building RESTful APIs.
- Axios for making HTTP requests.

Best Practices and Idioms in JavaScript:
- Use async/await for handling asynchronous operations.
- Follow naming conventions like camelCase for variables and functions.
- Utilize arrow functions for concise code.

Common Pitfalls to Avoid:
- Ensure proper error handling in asynchronous operations.
- Be mindful of differences in data types between Java and JavaScript.
- Pay attention to scope and context in JavaScript.

Database/API Migration Advice:
If the VideoService class interacts with a database or API, consider the following:
- **Database Migration:**
  - Use a Node.js ORM like Sequelize for database operations.
  - Update database connection configurations in JavaScript.

- **API Migration:**
  - Rewrite API calls using Axios or Fetch in JavaScript.
  - Adjust endpoint URLs and request/response handling.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining functionality and best practices.

35. src/main/java/com/example/momcare/exception/ResourceNotFoundException.java

Migration Guide: Java to JavaScript

ResourceNotFoundException Class

Summary:
The `ResourceNotFoundException` class is a custom exception class in Java used for handling resource not found errors.

Migration Plan:
1. Create a JavaScript class `ResourceNotFoundException`.
2. Define constructor functions to mimic the Java constructors.
3. Update the error handling logic to match JavaScript conventions.

Before (Java):
```java
package com.example.momcare.exception;

public class ResourceNotFoundException extends Exception {
    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }

    public ResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

After (JavaScript):
```javascript
class ResourceNotFoundException extends Error {
```

```
    constructor(message, cause) {
        super(message);
        this.name = "ResourceNotFoundException";
        if (cause) {
            this.cause = cause;
        }
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries or frameworks needed for this migration.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes.
- Inherit from `Error` object for custom error classes.

Common Pitfalls to Avoid:
- Forgetting to set the `name` property of the custom error class.

By following this migration guide, you can successfully convert the `ResourceNotFoundException` class from Java to JavaScript.

36. src/main/java/com/example/momcare/handler/NotificationHandler.java

Java to JavaScript Migration Guide

Summary
The given Java code represents a `NotificationHandler` class that handles WebSocket connections, sends messages to specific sessions, and processes incoming messages of different types (User and Notification). It interacts with various services and repositories to manage notifications for users.

Migration Plan
1. **WebSocket Handling**
   - Implement WebSocket handling in JavaScript using libraries like Socket.IO or WebSocket.
   - Replace `TextWebSocketHandler` with corresponding WebSocket handling methods.
   - Update session management and message sending logic for JavaScript.

2. **Service and Repository Integration**
   - Replace service and repository calls with API calls or JavaScript equivalents.
   - Update data handling and processing based on JavaScript conventions.

3. **JSON Handling**
   - Use `JSON.parse()` and `JSON.stringify()` for JSON serialization/deserialization.
   - Modify JSON-related methods to adhere to JavaScript standards.

4. **Error Handling**
   - Implement error handling mechanisms in JavaScript for exceptions or errors.
   - Update error response handling based on JavaScript practices.

5. **Notification Handling**
   - Adjust notification processing logic for JavaScript environment.
   - Update notification mapping and storage based on JavaScript requirements.

Code Migration Snippets
- **WebSocket Handling**
  ```javascript
  // JavaScript equivalent for handling WebSocket connections
  class NotificationHandler {
  ```

```
    constructor(userService, notificationRepository, notificationService) {
        this.userService = userService;
        this.notificationRepository = notificationRepository;
        this.notificationService = notificationService;
        this.sessions = new Map();
        this.objectMapper = new ObjectMapper();
        this.HEARTBEAT_INTERVAL = 30 * 1000;
    }

    // Implement WebSocket connection logic
    afterConnectionEstablished(session) {
        // Store the session
        this.sessions.set(session.id, session);
        this.startHeartbeat(session);
    }

    // Implement WebSocket message sending
    sendMessageToSession(sessionId, message, receiverId) {
        const session = this.sessions.get(sessionId);
        if (session && session.isOpen) {
            session.send(message);
        } else {
            // Handle session not open
            // User and notification handling logic
        }
    }

    // Other WebSocket handling methods
}
```

- **JSON Handling**
  ```javascript
  // JavaScript equivalent for JSON parsing in handleMessage method
  isMessageUserId(message) {
      const payload = message.payload;
      if (typeof payload === 'string') {
          try {
```

```
            const userHandlerRequest = this.objectMapper.readValue(payload);
            return userHandlerRequest && userHandlerRequest.id;
        } catch (ex) {
            return false;
        }
    }
    return false;
}
```

- **Notification Handling**
  ```javascript
  // JavaScript equivalent for mapping NotificationHandlerRequest to Notification
  map(notificationRequest) {
      const notification = {
          receiverId: notificationRequest.receiverId,
          senderId: notificationRequest.senderId,
          timestamp: new Date().toISOString(),
          read: false,
          notificationType: notificationRequest.notificationType,
          targetId: notificationRequest.targetId
      };
      return notification;
  }
  ```

Recommended Libraries/Frameworks in JavaScript
- **WebSocket Handling**: Socket.IO, WebSocket API
- **JSON Handling**: JSON, axios for API calls
- **Error Handling**: try...catch, error handling libraries like Sentry
- **Notification Handling**: Notification libraries like Noty, Push.js

Best Practices and Idioms in JavaScript
- Use asynchronous programming for WebSocket operations.
- Utilize ES6 features like arrow functions, template literals, and destructuring.
- Follow modular and object-oriented design patterns for better code organization.

Common Pitfalls to Avoid

- Mixing synchronous and asynchronous code in WebSocket handling.
- Not handling errors properly in WebSocket communication.
- Overlooking differences in data types and structures between Java and JavaScript.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining performance and functionality.

37. src/main/java/com/example/momcare/handler/TutorialHandler.java

Migration Guide: Java to JavaScript

Overview
The given code is a WebSocket handler in Java using Spring framework. It logs connection events, processes incoming messages, handles errors, and closes connections. The goal is to migrate this code to JavaScript for WebSocket handling.

Summary
The TutorialHandler class implements the WebSocketHandler interface and defines methods for handling WebSocket events such as connection establishment, message handling, error handling, and connection closure.

Migration Plan
1. **Setup WebSocket Client/Server in JavaScript**
   - Use libraries like `ws` or `socket.io` for WebSocket functionality.
2. **Translate Java Syntax to JavaScript**
   - Convert class syntax to function syntax.
   - Replace annotations with equivalent functionality in JavaScript.
3. **Implement WebSocket Event Handling**
   - Map Java event handling methods to JavaScript event listeners.
4. **Update Logging Mechanism**
   - Replace logging with JavaScript console.log or other logging libraries.
5. **Handle Threading**
   - Use asynchronous functions or Promises instead of Thread.sleep.

Before/After Code Snippets

Java (Before)
```java
package com.example.momcare.handler;

import org.springframework.web.socket.CloseStatus;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketMessage;
import org.springframework.web.socket.WebSocketSession;
```

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class TutorialHandler implements WebSocketHandler {
    // Methods implementation...
}
```

JavaScript (After)
```javascript
const WebSocket = require('ws');

class TutorialHandler {
    onConnectionEstablished(session) {
        console.log(`Connection established on session: ${session.id}`);
    }

    handleMessage(session, message) {
        const tutorial = message.toString();
        console.log(`Message: ${tutorial}`);
        session.send(`Started processing tutorial: ${session.id} - ${tutorial}`);
        setTimeout(() => {
            session.send(`Completed processing tutorial: ${tutorial}`);
        }, 1000);
    }

    handleTransportError(session, error) {
        console.log(`Exception occurred: ${error.message} on session: ${session.id}`);
    }

    onConnectionClosed(session, closeStatus) {
        console.log(`Connection closed on session: ${session.id} with status: ${closeStatus}`);
    }

    supportsPartialMessages() {
        return false;
    }
```

```
}

module.exports = TutorialHandler;
```

Recommended Libraries/Frameworks in JavaScript
- **ws**: A simple to use WebSocket implementation.
- **socket.io**: Provides WebSocket support with additional features like fallback options.

Best Practices and Idioms in JavaScript
- Use `const` and `let` for variable declarations.
- Utilize arrow functions for concise syntax.
- Prefer Promises or async/await over synchronous operations.

Common Pitfalls to Avoid
- Beware of differences in threading models between Java and JavaScript.
- Ensure proper error handling as JavaScript is single-threaded.

Conclusion
By following this migration guide, you can successfully convert the given Java WebSocket handler code to JavaScript, ensuring efficient WebSocket communication in your application.

38. src/main/java/com/example/momcare/models/BabyHealthIndex.java

Migration Guide: Java to JavaScript

BabyHealthIndex Class

Summary:
The `BabyHealthIndex` class represents a model for storing health index data of a baby, including various measurements and timestamps.

Migration Plan:
1. Translate the class definition and member variables to JavaScript.
2. Update the constructor to match JavaScript syntax.
3. Consider using ES6 classes for object-oriented structure.

Before (Java):
```java
public class BabyHealthIndex {
    private Double head;
    private Double biparietal;
    private Double occipitofrontal;
    private Double abdominal;
    private Double femur;
    private String timeCreate;
    private String timeUpdate;
    private List<WarningHealth> warningHealths;

    public BabyHealthIndex(Double head, Double biparietal, Double occipitofrontal, Double abdominal, Double femur, String timeCreate, String timeUpdate) {
        this.head = head;
        this.biparietal = biparietal;
        this.occipitofrontal = occipitofrontal;
        this.abdominal = abdominal;
        this.femur = femur;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
    }
}
```

```

### After (JavaScript):
```javascript
class BabyHealthIndex {
    constructor(head, biparietal, occipitofrontal, abdominal, femur, timeCreate, timeUpdate) {
        this.head = head;
        this.biparietal = biparietal;
        this.occipitofrontal = occipitofrontal;
        this.abdominal = abdominal;
        this.femur = femur;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
        this.warningHealths = [];
    }
}
```

Recommended Libraries/Frameworks:
- No additional libraries are required for this specific class migration.

Best Practices in JavaScript:
- Use ES6 classes for defining object-oriented structures.
- Use `const` or `let` for variable declaration instead of `var`.

Common Pitfalls to Avoid:
- JavaScript does not have strict typing like Java, so be mindful of data types when migrating.

Additional Notes:
- For handling lists like `warningHealths`, consider using arrays in JavaScript.
- If `WarningHealth` class exists, ensure it is migrated as well.

This migration guide provides a structured approach to converting the provided Java code to JavaScript while highlighting key considerations and best practices. Remember to adapt the migration plan based on the specific requirements and dependencies of your project.
```

39. src/main/java/com/example/momcare/models/Category.java

Migration Guide: Java to JavaScript

Category Class Migration

Summary:
The Category class defines a model for a category with properties like title, content, and thumbnail.

Migration Plan:
1. Create a JavaScript class with properties like title, content, and thumbnail.
2. Use ES6 class syntax to define the class.
3. Implement getter and setter methods for each property.

Before (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Category {
    private String title;
    private String content;
    private String thumbnail;
}
```

After (JavaScript):
```javascript
class Category {
    constructor() {
        this.title = '';
        this.content = '';
        this.thumbnail = '';
```

```
    }

    getTitle() {
        return this.title;
    }

    setTitle(title) {
        this.title = title;
    }

    getContent() {
        return this.content;
    }

    setContent(content) {
        this.content = content;
    }

    getThumbnail() {
        return this.thumbnail;
    }

    setThumbnail(thumbnail) {
        this.thumbnail = thumbnail;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor for initializing class properties.

Common Pitfalls to Avoid:
- Make sure to initialize properties in the constructor to avoid undefined errors.

This migration guide provides a step-by-step plan for converting the given Java code to JavaScript. Follow the outlined steps and recommendations to ensure a smooth migration process.

40. src/main/java/com/example/momcare/models/Diary.java

Migration Guide: Java to JavaScript

Summary
The provided Java code defines a `Diary` class with various properties such as id, idUser, title, content, thumbnail, reaction, timeCreate, and timeUpdate. The class is annotated with Lombok annotations for getter, setter, and toString methods, as well as with annotations for MongoDB document mapping.

Migration Plan
1. **Remove Java-specific annotations:** Replace Java-specific annotations with equivalent JavaScript code.
2. **Convert class structure:** Convert the class structure to JavaScript class syntax.
3. **Handle property types:** Convert property types and handle list data structure.
4. **Integrate with JavaScript frameworks:** Utilize JavaScript libraries or frameworks for MongoDB interactions if needed.

Migration Steps
1. Remove Java-specific annotations and import statements.
2. Convert class structure to JavaScript class syntax.
3. Update property types and handle lists.
4. Consider using a JavaScript MongoDB library for database interactions.

Before Migration (Java)
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;
@Getter
@Setter
@ToString
```

```
@Document(collection = "Diary")
public class Diary {
    @Id
    private String id;
    private String idUser;
    private String title;
    private String content;
    private List<String> thumbnail;
    private int reaction;
    private String timeCreate;
    private String timeUpdate;

    // Constructors
}
```

After Migration (JavaScript)
```javascript
class Diary {
    constructor(id, idUser, title, content, thumbnail, reaction, timeCreate, timeUpdate) {
        this.id = id;
        this.idUser = idUser;
        this.title = title;
        this.content = content;
        this.thumbnail = thumbnail;
        this.reaction = reaction;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
    }

    // Getters and setters can be added if needed
}
```

Recommended Libraries/Frameworks in JavaScript
- **Mongoose:** A popular MongoDB library for Node.js that provides a schema-based solution for modeling data.

Best Practices and Idioms in JavaScript
- Use ES6 class syntax for defining classes.
- Utilize arrow functions for concise and readable code.
- Follow consistent naming conventions (camelCase for variables and functions).

Common Pitfalls to Avoid
- Be mindful of asynchronous operations in JavaScript and handle them appropriately.
- Ensure proper error handling when interacting with MongoDB in JavaScript.

Database/API Migration Advice
- Use Mongoose for MongoDB database interactions in Node.js.
- Update API endpoints and request handling logic to align with JavaScript frameworks like Express.js.

By following this migration guide, the Java code for the `Diary` class can be successfully converted to JavaScript, maintaining functionality and readability.

41. src/main/java/com/example/momcare/models/HandBook.java

Migration Guide: Java to JavaScript

HandBook Model

Summary:
The code defines a Java model class `HandBook` with various fields annotated with Lombok annotations and Spring Data MongoDB annotations.

Migration Plan:
1. Create a JavaScript class `HandBook` to represent the model.
2. Use equivalent libraries or frameworks in JavaScript.
3. Update annotations and data types accordingly.
4. Handle asynchronous operations if required.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.DBRef;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.Date;
import java.util.List;
@Getter
@Setter
@ToString
@Document(collection = "HandBook")
public class HandBook {
    @Id
    private String id;
    private List<String> category;
    private String title;
```

```
    private String content;
    private String thumbnail;
    private String time;
}
```

After Migration (JavaScript):
```javascript
class HandBook {
    constructor() {
        this.id = "";
        this.category = [];
        this.title = "";
        this.content = "";
        this.thumbnail = "";
        this.time = "";
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose for MongoDB interactions.
- Express.js for building RESTful APIs.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes.
- Follow async/await pattern for handling asynchronous operations.

Common Pitfalls to Avoid:
- Be mindful of data type differences between Java and JavaScript.
- Understand the event-driven nature of JavaScript compared to synchronous Java.

This migration guide provides a starting point for converting the given Java code to JavaScript. Remember to adapt the migration plan based on the specific requirements and environment of your project.

42. src/main/java/com/example/momcare/models/HandBookCategory.java

Migration Guide: Java to JavaScript

HandBookCategory Model

Summary:
The `HandBookCategory` model represents a category in a handbook with various fields like `id`, `name`, `collection`, `content`, and `thumbnail`.

Migration Plan:
1. Convert the Java model class to a JavaScript class.
2. Replace annotations with equivalent functionality in JavaScript.
3. Update the class structure to align with JavaScript syntax.
4. Ensure proper handling of data types and properties.

Before (Java):
```java
package com.example.momcare.models;

import jakarta.validation.constraints.NotBlank;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.DBRef;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;

@Getter
@Setter
@ToString
@Document(collection = "HandBookCategory")
public class HandBookCategory {
    @Id
    private String id;
```

```
    @NotBlank
    private String name;
    private List<String> collection;
    private String content;
    private String thumbnail;
}
```

After (JavaScript):
```javascript
class HandBookCategory {
    constructor(id, name, collection, content, thumbnail) {
        this.id = id;
        this.name = name;
        this.collection = collection || [];
        this.content = content || '';
        this.thumbnail = thumbnail || '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose for MongoDB integration
- Express.js for backend development

Best Practices in JavaScript:
- Utilize ES6 features like classes and destructuring
- Follow consistent naming conventions

Common Pitfalls to Avoid:
- Ensure proper handling of asynchronous operations
- Be mindful of data type conversions

Additional Notes:
- For MongoDB integration, consider using Mongoose schemas in JavaScript.
- Update any service or controller classes that interact with this model to use the JavaScript equivalent.

43. src/main/java/com/example/momcare/models/HandBookCollection.java

Java to JavaScript Migration Guide

HandBookCollection Model

Summary:
The given Java code defines a `HandBookCollection` model class with various annotations for defining document structure in a MongoDB collection.

Migration Plan:
1. Create a JavaScript class with similar properties.
2. Use JavaScript equivalent of annotations for validation and mapping.
3. Handle getters and setters manually.

Before Migration (Java):
```java
package com.example.momcare.models;

import jakarta.validation.constraints.NotBlank;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
@Getter
@Setter
@ToString
@Document(collection = "HandBookCollection")
public class HandBookCollection {
    @Id
    private String id;
    @NotBlank
    private String name;
    private String content;
    private String thumbnail;
}
```

```

### After Migration (JavaScript):
```javascript
class HandBookCollection {
    constructor() {
        this.id = '';
        this.name = '';
        this.content = '';
        this.thumbnail = '';
    }
    // Add validation logic for 'name' property
}
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose for MongoDB interaction
- Joi for validation

Best Practices in JavaScript:
- Use ES6 classes for defining models
- Use const and let for variable declarations

Common Pitfalls to Avoid:
- Ensure proper asynchronous handling in JavaScript, especially when dealing with databases.

Additional Notes:
- For interacting with MongoDB in JavaScript, consider using Node.js with Express framework.
```

44. src/main/java/com/example/momcare/models/ImageMedia.java

Migration Guide: Java to JavaScript for ImageMedia class

Summary:
The Java code defines a class `ImageMedia` that extends `Media` class and includes properties for width, height, and format of an image media.

Migration Plan:
1. Create a JavaScript class `ImageMedia` that extends a base class `Media`.
2. Convert the properties to JavaScript syntax.
3. Implement getter and setter methods for the properties.
4. Update any references to the class in other parts of the code.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class ImageMedia extends Media {
    private double width;
    private double height;
    private String format;
}
```

After Migration (JavaScript):
```javascript
class ImageMedia extends Media {
    constructor() {
        super();
        this.width = 0;
        this.height = 0;
        this.format = '';
```

```
    }

    getWidth() {
        return this.width;
    }

    setWidth(width) {
        this.width = width;
    }

    getHeight() {
        return this.height;
    }

    setHeight(height) {
        this.height = height;
    }

    getFormat() {
        return this.format;
    }

    setFormat(format) {
        this.format = format;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- TypeScript for static typing and class-based programming.
- Babel for transpiling modern JavaScript features for broader browser compatibility.

Best Practices and Idioms in JavaScript:
- Use ES6 classes for defining classes.
- Avoid using getter/setter syntax unless necessary for encapsulation.

Common Pitfalls to Avoid:
- JavaScript does not have native support for access modifiers like Java, so encapsulation may

need to be handled differently.
- Ensure compatibility with asynchronous nature of JavaScript when dealing with APIs or databases.

This guide provides a structured approach to migrating the `ImageMedia` class from Java to JavaScript, considering syntax differences, best practices, and potential pitfalls.

45. src/main/java/com/example/momcare/models/Media.java

Java to JavaScript Migration Guide for Media Class

Summary
The Java code represents a `Media` class with properties like `url`, `ratio`, `description`, `uploadDate`, and `type`. It uses Lombok annotations for getter, setter, and constructors.

Migration Plan
1. **Convert Class Structure:**
   - Create a JavaScript class named `Media`.
   - Define properties `url`, `ratio`, `description`, `uploadDate`, and `type`.
2. **Handle Getters and Setters:**
   - Implement getter and setter methods for each property.
3. **Handle Constructors:**
   - Implement constructors for initializing the `Media` object.

Before/After Snippets
Before (Java)
```java
public class Media {
    private String url;
    private String ratio;
    private String description;
    private String uploadDate;
    private MediaType type;
}
```

After (JavaScript)
```javascript
class Media {
    constructor() {
        this.url = "";
        this.ratio = "";
        this.description = "";
        this.uploadDate = "";
        this.type = null;
```

```
    }

    // Getters and Setters
    getUrl() {
        return this.url;
    }

    setUrl(url) {
        this.url = url;
    }

    // Repeat for other properties
}
```

Recommended Libraries/Frameworks
- No specific libraries are required for this migration as it involves basic class and property conversion.

Best Practices in JavaScript
- Use `class` syntax for defining classes in JavaScript.
- Use concise arrow functions for methods where applicable.
- Prefer `const` and `let` for variable declarations over `var`.

Common Pitfalls to Avoid
- Ensure proper handling of data types as JavaScript is loosely typed compared to Java.
- Be mindful of asynchronous behavior in JavaScript if any operations are asynchronous.

This guide provides a structured approach to migrating the `Media` class from Java to JavaScript. Make sure to test thoroughly after migration to ensure correctness and functionality.

46. src/main/java/com/example/momcare/models/MediaType.java

Migration Guide: Java to JavaScript

Section: MediaType Enum

Summary:
The code defines an enum `MediaType` with two enum constants `IMAGE` and `VIDEO`.

Migration Plan:
1. Define a JavaScript object to represent the enum.
2. Convert the enum constants to properties of the object.
3. Update any code that references the enum constants.

Before:
```java
package com.example.momcare.models;

public enum MediaType {
    IMAGE(0),
    VIDEO(1);

    MediaType(int i) {
    }
}
```

After:
```javascript
const MediaType = {
    IMAGE: 0,
    VIDEO: 1
};
```

Recommended Libraries/Frameworks:
- No specific libraries/frameworks needed for this migration.

Best Practices and Idioms:
- Use object literals to represent enums in JavaScript.

Common Pitfalls:
- Forgetting to update code that references the enum constants after migration.

47. src/main/java/com/example/momcare/models/Menu.java

Migration Guide: Java to JavaScript

Menu Model

Summary:
The code defines a model class `Menu` with fields like `id`, `title`, `content`, `category`, `thumbnail`, and `rate`.

Migration Plan:
1. Create a JavaScript class `Menu` with equivalent fields.
2. Use ES6 class syntax for defining the class.
3. Replace annotations like `@Getter`, `@Setter`, `@ToString` with equivalent methods in JavaScript.

Before (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;

@Getter
@Setter
@ToString
@Document(collection = "Menu")
public class Menu {
    @Id
    private String id;
    private String title;
    private List<String> content;
    private String category;
```

```
    private String thumbnail;
    private List<Integer> rate;
}
```

After (JavaScript):
```javascript
class Menu {
    constructor() {
        this.id = "";
        this.title = "";
        this.content = [];
        this.category = "";
        this.thumbnail = "";
        this.rate = [];
    }

    // Getter methods
    getId() {
        return this.id;
    }

    getTitle() {
        return this.title;
    }

    // Setter methods
    setId(id) {
        this.id = id;
    }

    setTitle(title) {
        this.title = title;
    }

    // Other getter/setter methods for content, category, thumbnail, rate
}
```

Recommended Libraries/Frameworks in JavaScript:
- For object mapping and data manipulation: `lodash`, `underscore`
- For defining classes and inheritance: `ES6 classes`

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use arrow functions for concise method definitions.
- Prefer destructuring assignment for handling object properties.

Common Pitfalls to Avoid During Migration:
- Handling asynchronous operations correctly if any.
- Ensuring compatibility of data types between Java and JavaScript.

Conclusion:
The migration from Java to JavaScript for the `Menu` model involves creating a JavaScript class with equivalent fields and methods while following JavaScript best practices and idioms. Ensure proper testing after migration to validate the functionality.

48. src/main/java/com/example/momcare/models/MenuCategory.java

Migration Guide: Java to JavaScript

MenuCategory Model

Summary:
The given Java code defines a `MenuCategory` class with attributes like `id`, `name`, `thumbnail`, and `content`. It uses Lombok annotations for getter, setter, and toString methods, and Spring Data annotations for MongoDB mapping.

Migration Plan:
1. **Remove Lombok Annotations**: In JavaScript, getters and setters are defined explicitly, and `toString` functionality is achieved using custom methods.
2. **Replace Spring Data Annotations**: JavaScript doesn't use annotations like Spring Data, so MongoDB mapping needs to be handled differently.
3. **Convert Class Structure**: Define the class structure in JavaScript with required attributes.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Getter
@Setter
@ToString
@Document(collection = "MenuCategory")
public class MenuCategory {
    @Id
    private String id;
    private String name;
    private String thumbnail;
    private String content;
```

}
```

### After Migration (JavaScript):
```javascript
class MenuCategory {
    constructor(id, name, thumbnail, content) {
        this.id = id;
        this.name = name;
        this.thumbnail = thumbnail;
        this.content = content;
    }

    // Add getter and setter methods if needed
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Mongoose**: For MongoDB object modeling.
- **Express.js**: For building APIs and handling routes.

Best Practices in JavaScript:
- Use ES6 class syntax for defining classes.
- Avoid using global variables and follow module-based architecture.

Common Pitfalls to Avoid:
- Handling asynchronous operations in JavaScript can be different from Java, so understanding promises and async/await is crucial.
- JavaScript is dynamically typed, so ensure proper data validation.

Database Migration Advice:
- Use Mongoose or native MongoDB Node.js driver for database interactions.
- Define a connection to MongoDB using appropriate libraries.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and best practices.
```

49. src/main/java/com/example/momcare/models/MomHealthIndex.java

Java to JavaScript Migration Guide

MomHealthIndex Class

Summary
The `MomHealthIndex` class represents health index data for a mom, including height, weight, and various health indicators.

Migration Plan
1. Convert the class definition to a JavaScript class.
2. Replace the `@Getter` and `@Setter` annotations with equivalent getters and setters in JavaScript.
3. Update the constructor to match JavaScript syntax.

Before Migration (Java)
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;

import java.util.List;

@Getter
@Setter
public class MomHealthIndex {
    private Double height;
    private Double weight;
    private Double HATT;
    private Double HATTr;
    private Double GIHungry;
    private Double GIFull1h;
    private Double GIFull2h;
    private Double GIFull3h;
    private String timeCreate;
```

```
    private String timeUpdate;
    private List<WarningHealth> warningHealths;

    public MomHealthIndex(Double height, Double weight, Double HATT, Double HATTr, Double GIHungry, Double GIFull1h, Double GIFull2h, Double GIFull3h, String timeCreate, String timeUpdate) {
        this.height = height;
        this.weight = weight;
        this.HATT = HATT;
        this.HATTr = HATTr;
        this.GIHungry = GIHungry;
        this.GIFull1h = GIFull1h;
        this.GIFull2h = GIFull2h;
        this.GIFull3h = GIFull3h;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
    }
}
```

After Migration (JavaScript)
```javascript
class MomHealthIndex {
    constructor(height, weight, HATT, HATTr, GIHungry, GIFull1h, GIFull2h, GIFull3h, timeCreate, timeUpdate) {
        this.height = height;
        this.weight = weight;
        this.HATT = HATT;
        this.HATTr = HATTr;
        this.GIHungry = GIHungry;
        this.GIFull1h = GIFull1h;
        this.GIFull2h = GIFull2h;
        this.GIFull3h = GIFull3h;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
    }
}
```

Recommended Libraries/Frameworks
- No specific library or framework is required for this migration.

Best Practices and Idioms
- Use ES6 class syntax for defining classes in JavaScript.
- Use `constructor` for initializing class properties.

Common Pitfalls
- Remember to use proper variable scoping in JavaScript to avoid unexpected behavior.

Conclusion
By following the outlined migration plan and best practices, you can successfully convert the `MomHealthIndex` class from Java to JavaScript.

50. src/main/java/com/example/momcare/models/Music.java

Migration Guide: Java to JavaScript

com.example.momcare.models.Music

Summary:
The given Java code defines a model class `Music` that represents a music entity with various attributes like id, title, link, content, thumbnail, and category.

Migration Plan:
1. **Remove Lombok Annotations:** Since JavaScript does not have the same level of support for annotations like Lombok in Java, we will need to manually define getters and setters for the class properties.
2. **Update Annotations:** Replace `@Document` annotation with an equivalent in JavaScript (if needed).
3. **Convert List to Array:** Convert the `List<String> category` to a JavaScript array.
4. **Handle Id Field:** Decide on how to handle the `@Id` annotation and the `id` field in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;

@Getter
@Setter
@Document(collection = "Music")
public class Music {
    @Id
    private String id;
    private String title;
```

```
    private String link;
    private String content;
    private String thumbnail;
    private List<String> category;
}
```

After Migration (JavaScript):
```javascript
class Music {
    constructor() {
        this.id = null;
        this.title = '';
        this.link = '';
        this.content = '';
        this.thumbnail = '';
        this.category = [];
    }

    // Getters and Setters
    getId() {
        return this.id;
    }

    setId(id) {
        this.id = id;
    }

    // Repeat the above pattern for other properties
}
```

Recommended Libraries/Frameworks in JavaScript:
- **No specific library required for this migration.**

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor functions for initializing object properties.

Common Pitfalls to Avoid:
- Make sure to handle asynchronous operations properly if any database operations are involved.
- Ensure proper error handling for any API calls made from JavaScript.

By following this migration guide, you should be able to successfully convert the provided Java code to JavaScript while ensuring the functionality and structure are maintained.

51. src/main/java/com/example/momcare/models/MusicCategory.java

Migration Guide: Java to JavaScript Conversion

MusicCategory Model

Summary:
The Java code defines a model class `MusicCategory` using annotations from the Lombok and Spring Data MongoDB libraries.

Migration Plan:
1. Define an equivalent class in JavaScript using ES6 class syntax.
2. Use getters and setters methods for properties.
3. Simulate the `@Document` annotation for MongoDB collection mapping.
4. Handle the `@Id` annotation for the identifier field.

Code Snippets:
Java (Before):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Getter
@Setter
@Document(collection = "MusicCategory")
public class MusicCategory {
    @Id
    private String id;
    private String name;
    private String thumbnail;
    private String content;
}
```

JavaScript (After):
```javascript
class MusicCategory {
    constructor() {
        this.id = '';
        this.name = '';
        this.thumbnail = '';
        this.content = '';
    }
}

// Simulate @Document annotation
MusicCategory.collectionName = 'MusicCategory';

// Simulate @Id annotation
Object.defineProperty(MusicCategory.prototype, 'id', {
    writable: true,
    configurable: true,
    value: ''
});
```

Recommended Libraries/Frameworks:
- Mongoose for MongoDB schema modeling in Node.js.
- ES6 class syntax for defining models in JavaScript.

Best Practices in JavaScript:
- Use ES6 class syntax for defining classes.
- Define properties within the constructor for initialization.

Common Pitfalls to Avoid:
- Lack of understanding of ES6 class syntax.
- Not handling property initialization properly.

Conclusion:
By following this migration guide, you can successfully convert the Java code for the `MusicCategory` model to JavaScript, ensuring proper handling of annotations and class properties. Remember to utilize recommended libraries and frameworks for seamless

integration with MongoDB in a JavaScript environment.

52. src/main/java/com/example/momcare/models/Notification.java

Migration Guide: Java to JavaScript

Notification Class

Summary:
The `Notification` class represents a notification entity with various attributes like `id`, `senderId`, `receiverId`, etc. It also has a method `markAsRead()` to mark the notification as read.

Migration Plan:
1. Create a JavaScript class `Notification` with equivalent properties.
2. Implement the `markAsRead()` method in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.mongodb.core.mapping.Document;

@Setter
@Getter
@Document(collection = "Notification")
public class Notification {
    private String id;
    private String senderId;
    private String receiverId;
    private String targetId;
    private String timestamp;
    private boolean isRead;
    private NotificationType notificationType;
    public void markAsRead(){
        this.setRead(true);
    }
}
```

```

### After Migration (JavaScript):
```javascript
class Notification {
    constructor() {
        this.id = '';
        this.senderId = '';
        this.receiverId = '';
        this.targetId = '';
        this.timestamp = '';
        this.isRead = false;
        this.notificationType = null;
    }

    markAsRead() {
        this.isRead = true;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor function for initializing object properties.

Common Pitfalls to Avoid:
- JavaScript is loosely typed, ensure proper handling of variable types.
- Be mindful of asynchronous operations if dealing with API calls.

Conclusion:
By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining the functionality and structure of the `Notification` class.
```

53. src/main/java/com/example/momcare/models/NotificationType.java

Migration Guide: Java to JavaScript

NotificationType Enum Migration

Summary:
The code defines an enum class called `NotificationType` with four constants representing different types of notifications.

Migration Plan:
1. Create a JavaScript file to represent the enum.
2. Use JavaScript object to mimic enum behavior.
3. Update the references to the enum constants accordingly.

Before Migration (Java):
```java
package com.example.momcare.models;

public enum NotificationType {
    REACTION, COMMENT, FOLLOW , OTHER
}
```

After Migration (JavaScript):
```javascript
// notificationType.js
const NotificationType = {
    REACTION: 'REACTION',
    COMMENT: 'COMMENT',
    FOLLOW: 'FOLLOW',
    OTHER: 'OTHER'
};
export default NotificationType;
```

Recommended Libraries/Frameworks:
- No specific library or framework is needed for this migration.

Best Practices in JavaScript:
- Use object literals to represent enums in JavaScript.
- Use `const` to define constants to prevent accidental reassignment.

Common Pitfalls to Avoid:
- Ensure that all references to the enum constants are updated to match the new JavaScript representation.

Additional Notes:
- This migration only involves converting an enum from Java to JavaScript, which is a straightforward process.

54. src/main/java/com/example/momcare/models/PeriodicCheck.java

Migration Guide: Java to JavaScript

com.example.momcare.models.PeriodicCheck

Summary:
The given code defines a Java class `PeriodicCheck` with fields representing information related to periodic checks in a momcare application.

Migration Plan:
1. **Class Definition:**
   - Translate the class definition to a JavaScript class.
   - Use ES6 class syntax to define the class.
2. **Annotations:**
   - Replace `@Getter` and `@Setter` annotations with getter and setter methods in JavaScript.
   - Remove `@Document` annotation and collection attribute as it is specific to MongoDB in Spring Data.
3. **Data Types:**
   - Update data types accordingly in JavaScript.
4. **Id Annotation:**
   - Remove `@Id` annotation as it is specific to Spring Data.
5. **Export Class:**
   - Export the JavaScript class to be accessible in other modules.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;
@Getter
@Setter
@Document(collection = "PeriodicCheck")
```

```
public class PeriodicCheck {
    @Id
    private String id;
    private int weekFrom;
    private int weekEnd;
    private String title;
    private String obligatory;
    private String advice;
    private String note;
}
```

After Migration (JavaScript):
```javascript
class PeriodicCheck {
    constructor(id, weekFrom, weekEnd, title, obligatory, advice, note) {
        this.id = id;
        this.weekFrom = weekFrom;
        this.weekEnd = weekEnd;
        this.title = title;
        this.obligatory = obligatory;
        this.advice = advice;
        this.note = note;
    }

    // Getter methods
    getId() {
        return this.id;
    }

    // Setter methods
    setId(id) {
        this.id = id;
    }
}

module.exports = PeriodicCheck;
```

Recommended Libraries/Frameworks in JavaScript:
- **None required for this migration.**

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor for initializing class properties.
- Use getter and setter methods for data encapsulation.

Common Pitfalls to Avoid During Migration:
- Ensure that data types are correctly mapped from Java to JavaScript.
- Check for any specific annotations or dependencies that are Java-specific and replace them accordingly.

This guide provides a comprehensive approach to migrating the given Java class `PeriodicCheck` to JavaScript. Follow the outlined steps and recommendations to successfully convert the code while maintaining functionality and best practices in the target language.

55. src/main/java/com/example/momcare/models/Reaction.java

Migration Guide: Java to JavaScript for Reaction Enum

Summary:
The given code snippet defines an enum `Reaction` with different emotional reactions like LIKE, LOVE, CARE, etc., each with a corresponding integer value.

Migration Plan:
1. **Define Enum in JavaScript:** JavaScript does not have native support for enums like Java. We can simulate enums using objects or constants.
2. **Update Constructor:** Since the Java enum has an integer parameter in the constructor, we need to handle this in JavaScript.
3. **Update Usages:** Update any code that references the Java enum to use the equivalent JavaScript implementation.

Before Migration (Java):
```java
package com.example.momcare.models;

public enum Reaction {
    LIKE(1),
    LOVE(2),
    CARE(3),
    SMILE(4),
    WOW(5),
    CRY(6),
    ANGRY(7);

    Reaction(int i) {
    }
}
```

After Migration (JavaScript):
```javascript
const Reaction = {
    LIKE: 1,
```

```
    LOVE: 2,
    CARE: 3,
    SMILE: 4,
    WOW: 5,
    CRY: 6,
    ANGRY: 7
};
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries are needed for migrating enums from Java to JavaScript.

Best Practices and Idioms in JavaScript:
- Use object literals or constants to simulate enums in JavaScript.
- Prefer using the dot notation to access enum values.

Common Pitfalls to Avoid:
- Forgetting to add the equivalent values in JavaScript.
- Not updating all the places where the enum is used.

This migration guide provides a clear and structured approach to converting the given Java enum to JavaScript. Ensure to test thoroughly after migration to guarantee correctness.

56. src/main/java/com/example/momcare/models/Role.java

Migration Guide: Java Enum to JavaScript

Summary:
The given Java code defines an enum `Role` with two values: `USER` and `ADMIN`.

Migration Plan:
1. Convert the Java enum to a JavaScript object or a constant map.
2. Update any references to the enum in the codebase.

Before/After Code Snippets:
Java Enum (Before):
```java
package com.example.momcare.models;

public enum Role {
    USER,
    ADMIN,
}
```

JavaScript Object (After):
```javascript
const Role = {
    USER: 'USER',
    ADMIN: 'ADMIN',
};
```

Recommended Libraries or Frameworks in JavaScript:
- No specific libraries or frameworks are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use object literals to mimic enums in JavaScript.
- Use uppercase for enum values to indicate constants.

Common Pitfalls to Avoid:

- Forgetting to update all references to the enum values in the codebase.

This guide outlines the steps needed to migrate a Java enum to a JavaScript object or constant map. Remember to update all references to the enum values throughout the codebase.

57. src/main/java/com/example/momcare/models/SocialComment.java

Migration Guide: Java to JavaScript for SocialComment Model

Summary:
The Java code provided defines a `SocialComment` class with various properties related to a social media comment. It includes fields such as `userId`, `postId`, `commentId`, `imageUrl`, `reactions`, `replies`, `description`, and `time`.

Migration Plan:
1. **Properties Mapping:**
   - Map Java properties to JavaScript object properties.
2. **Constructor Conversion:**
   - Convert Java constructor to JavaScript constructor function.
3. **Initialization of Collections:**
   - Initialize `reactions` as an empty object `{}` and `replies` as an empty array `[]` in the constructor.

Before/After Code Snippets:
Java (Before):
```java
@Getter
@Setter
@Document(collection = "SocialComment")
public class SocialComment {
    // Properties omitted for brevity

    public SocialComment(String userId, String postId, String commentId,String imageUrl, String description, String time) {
        // Constructor code omitted for brevity
    }
}
```

JavaScript (After):
```javascript
class SocialComment {
    constructor(userId, postId, commentId, imageUrl, description, time) {
```

```
        this.userId = userId;
        this.postId = postId;
        this.commentId = commentId;
        this.imageUrl = imageUrl;
        this.reactions = {};
        this.replies = [];
        this.description = description;
        this.time = time;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this conversion as it involves basic object-oriented programming concepts in JavaScript.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Prefer object literals `{}` for representing maps in JavaScript.
- Use array literals `[]` for representing lists or collections.

Common Pitfalls to Avoid:
- Ensure that all properties are properly mapped from Java to JavaScript.
- Be mindful of data types, as JavaScript is dynamically typed compared to Java.

This migration guide provides a structured approach to converting the given Java code for the `SocialComment` class to equivalent JavaScript code. By following the outlined steps and recommendations, the migration process can be smooth and error-free.

58. src/main/java/com/example/momcare/models/SocialPost.java

Java to JavaScript Migration Guide for SocialPost Class

Summary:
The Java code provided defines a `SocialPost` class with fields representing attributes of a social media post, such as description, user ID, reactions, comments, shares, media content, and time.

Migration Plan:
1. **Fields and Constructor:**
   - Convert fields and constructor to JavaScript class properties and constructor.
   - Initialize collections as empty arrays/objects.

2. **Annotations:**
   - Remove annotations specific to Java like `@Getter`, `@Setter`, and `@Document`.

3. **Recommended Libraries:**
   - No specific libraries required for this conversion.

4. **Best Practices:**
   - Use ES6 class syntax for defining classes in JavaScript.
   - Use destructuring assignment for object initialization.

5. **Pitfalls to Avoid:**
   - Ensure compatibility with the target JavaScript environment (browser, Node.js, etc.).
   - Handle asynchronous behavior if required.

Migration Steps:

Before Migration (Java Code):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.mongodb.core.mapping.Document;
```

```
import java.util.*;

@Getter
@Setter
@Document(collection = "SocialPost")
public class SocialPost {
    private String id;
    private String description;
    private String userId;
    private Map<String, SocialReaction> reactions;
    private Set<String> comments;
    private Set<String> share;
    private List<Media> media;
    private String time;

    public SocialPost() {
    }

    public SocialPost(String description, String userId, List<Media> media, String time) {
        this.description = description;
        this.userId = userId;
        this.media = Objects.requireNonNullElse(media, Collections.emptyList());
        this.time = time;
        this.reactions = Collections.emptyMap();
        this.comments = Collections.emptySet();
        this.share = Collections.emptySet();
    }
}
```

After Migration (JavaScript Code):
```javascript
class SocialPost {
    constructor(description, userId, media, time) {
        this.description = description;
        this.userId = userId;
        this.media = media || [];
        this.time = time;
```

```
        this.reactions = {};
        this.comments = new Set();
        this.share = new Set();
    }
}
```

Conclusion:
By following this migration guide, you can successfully convert the provided Java `SocialPost` class to JavaScript. Remember to test the functionality thoroughly after migration.

59. src/main/java/com/example/momcare/models/SocialReaction.java

Migration Guide: Java to JavaScript

SocialReaction Model

Summary:
The Java code defines a `SocialReaction` class with time and reaction properties. It uses Lombok annotations for getters and setters.

Migration Plan:
1. Create a JavaScript class `SocialReaction` with `time` and `reaction` properties.
2. Implement getter and setter methods manually as JavaScript does not have built-in annotations like Lombok.
3. Use JavaScript's `Date` object to handle time.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

import java.time.LocalDateTime;
@Getter
@Setter
public class SocialReaction {

    private String time;
    private Reaction reaction;

    public SocialReaction() {
    }

    public SocialReaction(Reaction reaction) {
        this.time = LocalDateTime.now().toString();
        this.reaction = reaction;
    }
```

```
    public SocialReaction(Reaction reaction, String time) {
        this.time = time;
        this.reaction = reaction;
    }
}
```

After Migration (JavaScript):
```javascript
class SocialReaction {
    constructor(reaction, time = new Date().toString()) {
        this.time = time;
        this.reaction = reaction;
    }

    getTime() {
        return this.time;
    }

    setTime(time) {
        this.time = time;
    }

    getReaction() {
        return this.reaction;
    }

    setReaction(reaction) {
        this.reaction = reaction;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:

- Use ES6 classes for defining JavaScript classes.
- Follow consistent naming conventions for methods and properties.

Common Pitfalls to Avoid:
- Be mindful of the differences between Java's `LocalDateTime` and JavaScript's `Date` handling.

Conclusion:
By following this migration guide, you can successfully convert the given Java code for the `SocialReaction` model to JavaScript. Remember to handle date/time differences and implement getter/setter methods manually in JavaScript.

60. src/main/java/com/example/momcare/models/SocialStory.java

Migration Guide from Java to JavaScript

SocialStory Class

Summary:
The given Java code defines a class `SocialStory` with attributes `media` and `time`. It also includes constructors for creating instances of `SocialStory`.

Migration Plan:
1. Create a JavaScript class `SocialStory` with similar attributes and constructors.
2. Update the syntax as per JavaScript standards.
3. Handle Date formatting differences between Java and JavaScript.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

import java.time.LocalDate;

@Getter
@Setter
public class SocialStory{
    private Media media;
    private String time;

    public SocialStory() {
    }

    public SocialStory(Media media, String time) {
        this.media = media;
        this.time = time;
    }
```

```
    public SocialStory(Media media) {
        this.media = media;
        this.time = LocalDate.now().toString();
    }
}
```

After Migration (JavaScript):
```javascript
class SocialStory {
    constructor(media, time) {
        this.media = media;
        this.time = time || new Date().toISOString();
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Moment.js for handling date and time formatting.
- Babel for transpiling ES6 code to ES5 for wider browser compatibility.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use `const` and `let` for variable declarations instead of `var`.

Common Pitfalls to Avoid:
- JavaScript dates are handled differently than Java, so ensure proper conversion and formatting.

Overall Migration Steps:
1. Identify all Java classes and their functionality.
2. Create corresponding JavaScript classes with similar attributes and methods.
3. Modify the syntax and handling of data types where necessary.
4. Test thoroughly to ensure functionality is maintained post-migration.

By following this migration guide, you can effectively convert Java code to JavaScript while

ensuring the maintainability and functionality of the application.

61. src/main/java/com/example/momcare/models/StandardsIndex.java

Migration Guide: Java to JavaScript for StandardsIndex Class

Summary:
The provided Java code defines a `StandardsIndex` class with `min` and `max` properties along with constructors to initialize these properties.

Migration Plan:
1. Create a JavaScript class named `StandardsIndex`.
2. Define `min` and `max` properties in the JavaScript class.
3. Implement constructors with appropriate parameters.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class StandardsIndex {
    private Double min;
    private Double max;

    public StandardsIndex(Double max) {
        this.max = max;
    }

    public StandardsIndex(Double min, Double max) {
        this.min = min;
        this.max = max;
    }
}
```

After Migration (JavaScript):

```javascript
class StandardsIndex {
    constructor(max) {
        this.max = max;
    }

    constructor(min, max) {
        this.min = min;
        this.max = max;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes in JavaScript.
- Prefer `constructor` function for class initialization.

Common Pitfalls to Avoid:
- JavaScript does not support method overloading like Java, so ensure unique method names.

By following this migration guide, you can successfully convert the provided Java `StandardsIndex` class to JavaScript.

62. src/main/java/com/example/momcare/models/Tracking.java

Migration Guide: Java to JavaScript

File: Tracking.java

Summary:
The `Tracking` class represents a model for tracking pregnancy information with various fields like id, week, thumbnails, baby, mom, and advice.

Migration Plan:
1. Create a JavaScript class `Tracking` to mirror the Java class.
2. Use ES6 class syntax for defining the class.
3. Convert annotations like `@Id`, `@Document`, `@Getter`, `@Setter`, and `@ToString` to equivalent JavaScript code.
4. Ensure proper handling of data types like `List<String>`.

Before (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;

@Getter
@Setter
@ToString
@Document(collection = "TrackingPregnancy")
public class Tracking {
    @Id
    private String id;
    private int week;
    private List<String> thumbnails;
```

```
    private String baby;
    private String mom;
    private String advice;

}
```

After (JavaScript):
```javascript
class Tracking {
    constructor() {
        this.id = '';
        this.week = 0;
        this.thumbnails = [];
        this.baby = '';
        this.mom = '';
        this.advice = '';
    }
}

// Equivalent to @Document(collection = "TrackingPregnancy")
Tracking.collection = "TrackingPregnancy";
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose for MongoDB integration.
- Class syntax for defining models.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use camelCase for variable and function names.

Common Pitfalls to Avoid:
- Ensure proper handling of asynchronous operations when dealing with databases in JavaScript.

This comprehensive migration guide provides a step-by-step plan to convert the given Java code to JavaScript, along with best practices and recommendations for a successful migration process.

63. src/main/java/com/example/momcare/models/User.java

Migration Guide: Java to JavaScript

User Class Migration

Summary:
The `User` class represents a user entity with various attributes like username, password, email, etc.

Migration Plan:
1. Convert Java class to JavaScript class.
2. Update annotations with JavaScript equivalents.
3. Replace Java data types with JavaScript data types.
4. Add necessary imports for JavaScript.

Before Migration (Java):
```java
package com.example.momcare.models;

import com.example.momcare.payload.response.NotificationResponse;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;
import java.util.Set;

@Getter
@Setter
@ToString
@Document(collection = "Account")
public class User   {
```

```
    @Id
    private String id;
    @NotBlank
    @Size(max = 20)
    private String userName;
    @NotBlank
    @Size(max = 120)
    private String passWord;
    @NotBlank
    @Size(max = 50)
    @Email
    private String email;
    private String avtUrl;
    private String nameDisplay;
    private Set<String> follower;
    private Set<String> following;
    private Set<String> shared;
    private String datePregnant;
    private Boolean premium;
    private Role roles;
    private List<MomHealthIndex> momIndex;
    private List<BabyHealthIndex> babyIndex;
    private Boolean enabled;
    private String token;
    private String otp;
    private String passwordToken;
    private String sessionId;
    private List<NotificationResponse> notificationsMissed;
}
```

After Migration (JavaScript):
```javascript
class User {
    constructor() {
        this.id = '';
        this.userName = '';
        this.passWord = '';
```

```
        this.email = '';
        this.avtUrl = '';
        this.nameDisplay = '';
        this.follower = new Set();
        this.following = new Set();
        this.shared = new Set();
        this.datePregnant = '';
        this.premium = false;
        this.roles = null;
        this.momIndex = [];
        this.babyIndex = [];
        this.enabled = false;
        this.token = '';
        this.otp = '';
        this.passwordToken = '';
        this.sessionId = '';
        this.notificationsMissed = [];
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for backend server.
- Mongoose for MongoDB object modeling.
- Joi for input validation.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declarations.
- Follow ES6 syntax and features like arrow functions and template literals.

Common Pitfalls to Avoid:
- JavaScript is loosely typed compared to Java, so pay attention to data types.
- Ensure proper error handling and input validation in JavaScript.

Conclusion:
By following this migration guide, you can successfully convert the provided Java code to JavaScript while adhering to best practices and utilizing recommended libraries.

64. src/main/java/com/example/momcare/models/UserStory.java

Java to JavaScript Migration Guide

UserStory Class Migration

Summary:
The Java `UserStory` class represents a model for user stories with various fields like id, userId, and socialStories. It includes constructors to create instances of UserStory.

Migration Plan:
1. Create a JavaScript class `UserStory` with similar properties and methods.
2. Use JavaScript Date objects for date and time operations.
3. Update the constructor to handle date formatting using JavaScript libraries like Moment.js.
4. Update the syntax for class properties and methods.

Before Migration:
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

@Setter
@Getter
@Document(collection = "UserStory")
public class UserStory {
    @Id
    private String id;
```

```
    private String userId;
    private List<SocialStory> socialStories;

    // Constructors...
}
```

After Migration:
```javascript
class UserStory {
    constructor(userId, socialStory) {
        this.userId = userId;
        this.socialStories = [];
        socialStory.time = new Date().toISOString(); // Use JavaScript Date for time
        this.socialStories.push(socialStory);
    }

    // Other constructors and methods...
}
```

Recommended Libraries/Frameworks:
- Moment.js for date/time formatting in JavaScript.
- Mongoose for MongoDB object modeling in Node.js.

Best Practices and Idioms:
- Use ES6 class syntax for defining classes in JavaScript.
- Follow naming conventions for properties and methods.

Common Pitfalls to Avoid:
- Be mindful of asynchronous operations in JavaScript, especially when dealing with APIs or databases.

This is just a guide for migrating one class from Java to JavaScript. Repeat the same process for other classes and dependencies in the codebase.

65. src/main/java/com/example/momcare/models/Video.java

Migration Guide: Java to JavaScript

Video Class Migration

Summary:
The Java code snippet provided defines a `Video` class with various attributes like `id`, `title`, `link`, `content`, `thumbnail`, and `category`. It also utilizes Lombok annotations for getter and setter methods and MongoDB annotations for document mapping.

Migration Plan:
1. **Remove Java specific annotations**: Replace `@Getter`, `@Setter`, `@Document`, and `@Id` with equivalent JavaScript syntax.
2. **Convert class attributes**: Update the class attributes to JavaScript syntax.
3. **Handle List data type**: Convert `List<String>` to an appropriate data structure in JavaScript.
4. **Optional: Use a library for schema validation**: Consider using a library like Joi for schema validation in JavaScript.

Code Snippets:
Before:
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;
@Getter
@Setter
@Document(collection = "Video")
public class Video {
    @Id
    private String id;
    private String title;
```

```
    private String link;
    private String content;
    private String thumbnail;
    private List<String> category;
}
```

After:
```javascript
class Video {
    constructor() {
        this.id = '';
        this.title = '';
        this.link = '';
        this.content = '';
        this.thumbnail = '';
        this.category = [];
    }
}
```

Recommended Libraries/Frameworks:
- **Mongoose**: For MongoDB object modeling in Node.js.
- **Joi**: For schema validation in JavaScript.

Best Practices and Idioms:
- Use ES6 class syntax for defining classes.
- Utilize spread syntax for handling array data.

Common Pitfalls:
- Ensure proper handling of asynchronous operations in JavaScript.
- Pay attention to data types and conversions between Java and JavaScript.

Overall Migration Advice:
- Understand the differences in syntax and paradigms between Java and JavaScript.
- Leverage JavaScript frameworks like Node.js for server-side operations.
- Consider using TypeScript for type safety in JavaScript code.

66. src/main/java/com/example/momcare/models/VideoCategory.java

Migration Guide: Java to JavaScript

VideoCategory Model

Summary:
The given Java code defines a model class `VideoCategory` using the Lombok library annotations for generating getters and setters, and Spring Data annotations for MongoDB integration.

Migration Plan:
1. **Remove Java-specific annotations**: Replace `@Getter`, `@Setter`, `@Document`, and `@Id` annotations with JavaScript equivalents.
2. **Convert class to JavaScript format**: Define the `VideoCategory` class in JavaScript with properties `id`, `name`, `thumbnail`, and `content`.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Getter
@Setter
@Document(collection = "VideoCategory")
public class VideoCategory {
    @Id
    private String id;
    private String name;
    private String thumbnail;
    private String content;
}
```

After Migration (JavaScript):
```javascript
class VideoCategory {
    constructor(id, name, thumbnail, content) {
        this.id = id;
        this.name = name;
        this.thumbnail = thumbnail;
        this.content = content;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **No specific library required** for defining model classes in JavaScript.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor functions for initializing class properties.

Common Pitfalls to Avoid:
- Ensure proper handling of asynchronous operations if integrating with MongoDB in JavaScript.

This is a basic migration guide for converting the given Java code for the `VideoCategory` model to JavaScript. Further steps may be required based on the overall application architecture and requirements.

67. src/main/java/com/example/momcare/models/VideoMedia.java

Migration Guide: Java to JavaScript

VideoMedia Class Migration

Summary:
The `VideoMedia` class is a model class that extends the `Media` class and includes properties for duration and resolution.

Migration Plan:
1. Create a JavaScript class `VideoMedia` that extends a base class `Media`.
2. Define properties `duration` and `resolution`.
3. Implement getters and setters for the properties.

Before (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class VideoMedia extends Media{
    private double duration;
    private String resolution;
}
```

After (JavaScript):
```javascript
class VideoMedia extends Media {
    constructor() {
        super();
        this.duration = 0.0;
        this.resolution = '';
    }
```

```
    getDuration() {
        return this.duration;
    }

    setDuration(duration) {
        this.duration = duration;
    }

    getResolution() {
        return this.resolution;
    }

    setResolution(resolution) {
        this.resolution = resolution;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks needed for this migration.

Best Practices in JavaScript:
- Use ES6 class syntax for defining classes.
- Use getter and setter methods for encapsulation.
- Use `super()` to call the constructor of the base class.

Common Pitfalls to Avoid:
- Ensure proper data type handling in JavaScript as it is weakly typed compared to Java.
- Handle asynchronous operations if needed in JavaScript.

Overall Recommendations:
- Follow a step-by-step approach when migrating code.
- Test thoroughly after migration to ensure the functionality remains intact.
- Make use of modern JavaScript features to improve code readability and maintainability.

68. src/main/java/com/example/momcare/models/WarningHealth.java

Migration Guide from Java to JavaScript

WarningHealth Class Migration

Summary:
The code defines a Java class `WarningHealth` with attributes `name`, `type`, and `level`, along with Lombok annotations `@Getter` and `@Setter` for automatic getter and setter generation.

Migration Plan:
1. Create a JavaScript class `WarningHealth` with properties `name`, `type`, and `level`.
2. Implement getter and setter methods manually in JavaScript.
3. Remove Lombok annotations as they are not applicable in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.models;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class WarningHealth {
    private String name;
    private String type;
    private int level;
}
```

After Migration (JavaScript):
```javascript
class WarningHealth {
    constructor() {
        this.name = '';
        this.type = '';
        this.level = 0;
```

```
    }

    getName() {
        return this.name;
    }

    setName(name) {
        this.name = name;
    }

    getType() {
        return this.type;
    }

    setType(type) {
        this.type = type;
    }

    getLevel() {
        return this.level;
    }

    setLevel(level) {
        this.level = level;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Implement getter and setter methods for proper encapsulation.

Common Pitfalls to Avoid:
- Remember that JavaScript does not have built-in support for getters and setters like Java, so manual implementation is necessary.

By following this migration guide, you can successfully convert the given Java code to JavaScript. Remember to test thoroughly to ensure the functionality remains intact after the migration.

69. src/main/java/com/example/momcare/payload/request/AddUserFollowerRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.AddUserFollowerRequest

Summary:
The provided Java code defines a class `AddUserFollowerRequest` with two private string fields `idUser` and `idFollowingUser`, along with getter and setter methods generated by Lombok annotations.

Migration Plan:
1. Create a JavaScript class with equivalent properties and methods.
2. Implement getter and setter functions manually as JavaScript does not have built-in support like Lombok in Java.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class AddUserFollowerRequest {
    private String idUser;
    private String idFollowingUser;
}
```

After Migration (JavaScript):
```javascript
class AddUserFollowerRequest {
    constructor() {
        this.idUser = "";
        this.idFollowingUser = "";
```

```
    }

    getIdUser() {
        return this.idUser;
    }

    setIdUser(idUser) {
        this.idUser = idUser;
    }

    getIdFollowingUser() {
        return this.idFollowingUser;
    }

    setIdFollowingUser(idFollowingUser) {
        this.idFollowingUser = idFollowingUser;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific library is required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes in JavaScript.
- Follow consistent naming conventions similar to Java.

Common Pitfalls to Avoid:
- Be mindful of differences in how getters and setters are implemented in JavaScript compared to Java.
- Ensure proper initialization of class properties within the constructor.

Overall Notes:
- This migration involves straightforward translation of a simple Java class to JavaScript without any external dependencies.

70. src/main/java/com/example/momcare/payload/request/BabyHealthIndexRequest.java

Migration Guide: Java to JavaScript

File: BabyHealthIndexRequest.java

Summary:
The `BabyHealthIndexRequest` class is a Java POJO (Plain Old Java Object) representing a request payload for baby health index data. It contains fields such as `userID`, `index`, and various body measurements.

Migration Plan:
1. Create a JavaScript class with equivalent properties.
2. Use ES6 class syntax for defining the class.
3. Replace Lombok annotations with manual getter and setter methods.
4. Update data types to match JavaScript equivalents.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class BabyHealthIndexRequest {
    private String userID;
    private int index;
    private Double head;
    private Double biparietal;
    private Double occipitofrontal;
    private Double abdominal;
    private Double femur;
}
```

After Migration (JavaScript):
```javascript
class BabyHealthIndexRequest {
    constructor() {
        this.userID = '';
        this.index = 0;
        this.head = 0.0;
        this.biparietal = 0.0;
        this.occipitofrontal = 0.0;
        this.abdominal = 0.0;
        this.femur = 0.0;
    }

    getUserId() {
        return this.userID;
    }

    setUserId(userId) {
        this.userID = userId;
    }

    // Define getter and setter methods for other properties
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Follow consistent naming conventions (camelCase for variables and functions).

Common Pitfalls to Avoid:
- Ensure data types are correctly mapped between Java and JavaScript.

Overall Migration Advice:
- Keep the code modular and follow JavaScript best practices.
- Test thoroughly after migration to ensure functionality is retained.

71. src/main/java/com/example/momcare/payload/request/ChangePasswordRequest.java

Java to JavaScript Migration Guide for ChangePasswordRequest Class

Summary:
The Java code defines a `ChangePasswordRequest` class with three string fields: `id`, `oldPassword`, and `newPassword`. The class uses Lombok annotations `@Getter` and `@Setter` to automatically generate getter and setter methods for the fields.

Migration Plan:
1. **Remove Lombok Annotations:** Since JavaScript does not have an equivalent to Lombok, manually create getter and setter methods for the fields.
2. **Convert Class Definition:** Translate the class definition syntax from Java to JavaScript.
3. **Update Data Types:** JavaScript does not require explicit data types for variables, so remove them from the field declarations.

Before/After Code Snippets:
Java Code:
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class ChangePasswordRequest {
    private String id;
    private String oldPassword;
    private String newPassword;
}
```

JavaScript Equivalent:
```javascript
class ChangePasswordRequest {
    constructor() {
```

```
        this.id = "";
        this.oldPassword = "";
        this.newPassword = "";
    }

    getId() {
        return this.id;
    }

    setId(id) {
        this.id = id;
    }

    getOldPassword() {
        return this.oldPassword;
    }

    setOldPassword(oldPassword) {
        this.oldPassword = oldPassword;
    }

    getNewPassword() {
        return this.newPassword;
    }

    setNewPassword(newPassword) {
        this.newPassword = newPassword;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this conversion.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Follow consistent naming conventions for methods and variables.

Common Pitfalls to Avoid:
- Remember to initialize the fields in the constructor to avoid `undefined` values.
- Ensure proper scoping of variables and methods within the class.

By following this migration guide, you can successfully convert the `ChangePasswordRequest` class from Java to JavaScript.

72. src/main/java/com/example/momcare/payload/request/CreatePasswordRequest.java

Migration Guide: Java to JavaScript

CreatePasswordRequest Class

Summary:
The `CreatePasswordRequest` class is a simple Java class with three private fields `userName`, `token`, and `newPassword`, along with getter and setter methods generated using Lombok annotations.

Migration Plan:
1. Create a JavaScript class `CreatePasswordRequest` with three properties `userName`, `token`, and `newPassword`.
2. Implement getter and setter methods manually in JavaScript as there is no direct equivalent of Lombok in JavaScript.
3. Ensure data encapsulation by keeping the properties private and accessing them through getter and setter methods.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class CreatePasswordRequest {
    private String userName;
    private String token;
    private String newPassword;
}
```

After Migration (JavaScript):
```javascript
```

```
class CreatePasswordRequest {
    constructor() {
        this._userName = '';
        this._token = '';
        this._newPassword = '';
    }

    get userName() {
        return this._userName;
    }

    set userName(userName) {
        this._userName = userName;
    }

    get token() {
        return this._token;
    }

    set token(token) {
        this._token = token;
    }

    get newPassword() {
        return this._newPassword;
    }

    set newPassword(newPassword) {
        this._newPassword = newPassword;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- None required for this specific migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.

- Manually implement getter and setter methods for data encapsulation.

Common Pitfalls to Avoid:
- Forgetting to use `_` prefix for private properties in JavaScript.

Overall Migration Advice:
- Keep the code modular and follow JavaScript best practices.
- Test thoroughly after migration to ensure functionality remains intact.

73. src/main/java/com/example/momcare/payload/request/DiaryRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.DiaryRequest

Summary:
The code defines a Java class `DiaryRequest` with various fields representing attributes of a diary entry.

Migration Plan:
1. Create a JavaScript class `DiaryRequest` with equivalent fields.
2. Use ES6 class syntax for defining the class.
3. Replace `@Getter`, `@Setter`, `@AllArgsConstructor`, `@NoArgsConstructor` annotations with equivalent JavaScript syntax.
4. Update data types as per JavaScript conventions.
5. Make necessary adjustments for handling arrays in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.List;
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class DiaryRequest {
    private String id;
    private String idUser;
    private String title;
    private String content;
```

```
    private List<String> thumbnail;
    private int reaction;
    private String timeCreate;
    private String timeUpdate;
}
```

After Migration (JavaScript):
```javascript
class DiaryRequest {
    constructor(id, idUser, title, content, thumbnail, reaction, timeCreate, timeUpdate) {
        this.id = id;
        this.idUser = idUser;
        this.title = title;
        this.content = content;
        this.thumbnail = thumbnail;
        this.reaction = reaction;
        this.timeCreate = timeCreate;
        this.timeUpdate = timeUpdate;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries/frameworks are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use `constructor` for initializing class properties.
- Follow camelCase naming convention for variables.

Common Pitfalls to Avoid:
- JavaScript does not have strict types like Java, so be cautious with data type conversions.
- Arrays and objects behave differently in JavaScript compared to Java, so handle them accordingly.

Database/API Migration Advice:
If the class is used for interacting with a database or API, you may need to update the

corresponding database queries or API endpoints to work with JavaScript.

By following this migration guide, you can effectively convert the provided Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

74. src/main/java/com/example/momcare/payload/request/MomHealthIndexRequest.java

Migration Guide: Java to JavaScript

MomHealthIndexRequest Class

Summary:
The `MomHealthIndexRequest` class is a Java class used for storing health index data for a mom, including user ID, index, height, weight, and various other health-related parameters.

Migration Plan:
1. Create a JavaScript class with equivalent properties.
2. Use ES6 class syntax for defining the class.
3. Remove annotations like `@Getter` and `@Setter` as they are not needed in JavaScript.

Before (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class MomHealthIndexRequest {
    private String userID;
    private int index;
    private Double height;
    private Double weight;
    private Double HATT;
    private Double HATTr;
    private Double GIHungry;
    private Double GIFull1h;
    private Double GIFull2h;
    private Double GIFull3h;
}
```

After (JavaScript):
```javascript
class MomHealthIndexRequest {
    constructor() {
        this.userID = '';
        this.index = 0;
        this.height = null;
        this.weight = null;
        this.HATT = null;
        this.HATTr = null;
        this.GIHungry = null;
        this.GIFull1h = null;
        this.GIFull2h = null;
        this.GIFull3h = null;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use `null` for initializing properties with no values.

Common Pitfalls to Avoid:
- JavaScript does not have the concept of annotations like `@Getter` and `@Setter`, so ensure to remove them.

Conclusion:
By following this migration guide, you can successfully convert the given Java code for `MomHealthIndexRequest` class to JavaScript. Remember to adhere to JavaScript best practices and idioms throughout the migration process.

75. src/main/java/com/example/momcare/payload/request/NotificationHandlerRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.NotificationHandlerRequest

Summary:
The given Java code defines a class `NotificationHandlerRequest` that represents a request payload for handling notifications in a momcare application. It contains fields for senderId, receiverId, targetId, and notificationType.

Migration Plan:
1. Create a JavaScript class with the same properties.
2. Implement getters and setters if needed.
3. Update any references to NotificationType enum.

Before (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.NotificationType;
import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class NotificationHandlerRequest {
    private String senderId;
    private String receiverId;
    private String targetId;
    private NotificationType notificationType;
}
```

After (JavaScript):
```javascript
class NotificationHandlerRequest {
```

```
    constructor() {
        this.senderId = '';
        this.receiverId = '';
        this.targetId = '';
        this.notificationType = ''; // Update according to JavaScript equivalent
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- None required for this specific migration.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes.
- Prefer `constructor` for initializing properties.

Common Pitfalls to Avoid:
- Ensure enum values are correctly mapped or handled in JavaScript.

Overall Recommendations:
- Utilize ES6 features like classes and arrow functions.
- Pay attention to data types and conversions between Java and JavaScript.
- Test thoroughly after migration to ensure functionality is maintained.

This guide provides a structured approach to converting the given Java code to JavaScript. Following these steps will help in a smooth and successful migration process.

76. src/main/java/com/example/momcare/payload/request/NotificationRequest.java

Java to JavaScript Migration Guide

NotificationRequest Class

Summary:
The Java code defines a `NotificationRequest` class with attributes for senderId, receiverId, targetId, and notificationType.

Migration Plan:
1. Create a JavaScript class with the same attributes and use ES6 class syntax.
2. Utilize getters and setters methods for encapsulation.
3. Use default parameter values for constructor initialization.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.NotificationType;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class NotificationRequest {
    private String senderId;
    private String receiverId;
    private String targetId;
    private NotificationType notificationType;
}
```

After Migration (JavaScript):
```javascript
class NotificationRequest {
    constructor(senderId, receiverId, targetId, notificationType) {
        this.senderId = senderId;
        this.receiverId = receiverId;
        this.targetId = targetId;
        this.notificationType = notificationType;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No additional libraries are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Utilize default parameter values for constructor initialization.

Common Pitfalls to Avoid:
- Ensure proper scoping and access control of class attributes in JavaScript.

By following this migration guide, you can effectively convert the provided Java code to JavaScript. Ensure to test the migrated code thoroughly to validate its functionality.

77. src/main/java/com/example/momcare/payload/request/OPTRequest.java

Migration Guide: Java to JavaScript

OPTRequest Class

Summary:
The `OPTRequest` class is a simple Java class with `userName` and `otp` fields along with Lombok annotations for getters and setters.

Migration Plan:
1. Create a JavaScript class with `userName` and `otp` properties.
2. Implement getter and setter methods manually in JavaScript.
3. Update any references to the class to use the new JavaScript implementation.

Before (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class OPTRequest {
    private String userName;
    private String otp;
}
```

After (JavaScript):
```javascript
class OPTRequest {
    constructor() {
        this.userName = '';
        this.otp = '';
    }
```

```
    getUserName() {
        return this.userName;
    }

    setUserName(userName) {
        this.userName = userName;
    }

    getOtp() {
        return this.otp;
    }

    setOtp(otp) {
        this.otp = otp;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Explicitly define getter and setter methods for properties.

Common Pitfalls to Avoid:
- Ensure that all references to getters and setters are updated after migration.

Database/API Usage:
If the `OPTRequest` class is used in conjunction with a database or API, the migration would involve updating the data access and API communication code to align with JavaScript syntax and standards. Libraries like Axios can be used for API calls in JavaScript.

78. src/main/java/com/example/momcare/payload/request/ShareResquest.java

Migration Guide: Java to JavaScript

`ShareResquest.java`

Summary:
The `ShareResquest` class is a Java bean class that defines the structure of a request payload for sharing a post.

Migration Plan:
1. **Create a JavaScript class `ShareResquest` with equivalent properties.**
2. **Remove the Lombok annotations `@Setter` and `@Getter` as JavaScript does not have direct equivalents.**
3. **Update the property declarations to follow JavaScript conventions.**

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class ShareResquest {
    private String idPost;
    private String idUser;
}
```

After Migration (JavaScript):
```javascript
class ShareResquest {
    constructor() {
        this.idPost = '';
        this.idUser = '';
```

```
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries or frameworks are needed for this migration.

Best Practices in JavaScript:
- Use ES6 class syntax for defining classes.
- Utilize constructor functions for initializing class properties.

Common Pitfalls to Avoid:
- Be mindful of the differences in syntax and conventions between Java and JavaScript.
- Ensure proper data types are used when declaring properties in JavaScript.

Overall, migrating the `ShareResquest` class from Java to JavaScript involves creating a JavaScript class with equivalent properties and adhering to JavaScript conventions for class definitions. Remember to remove any Java-specific annotations and update the syntax accordingly.

79. src/main/java/com/example/momcare/payload/request/SocialCommentDeleteRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.SocialCommentDeleteRequest

Summary:
The given Java code defines a payload request class `SocialCommentDeleteRequest` with three string properties `postId`, `id`, and `commentId`.

Migration Plan:
1. Create a JavaScript class `SocialCommentDeleteRequest` with properties `postId`, `id`, and `commentId`.
2. Use ES6 class syntax to define the class.
3. Consider using TypeScript for type safety.

Before (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class SocialCommentDeleteRequest {
    private String postId;
    private String id;
    private String commentId;
}
```

After (JavaScript):
```javascript
class SocialCommentDeleteRequest {
    constructor() {
        this.postId = "";
```

```
        this.id = "";
        this.commentId = "";
    }
}
```

Recommended Libraries/Frameworks:
- TypeScript for type safety
- Babel for transpiling ES6 code
- ESLint for code linting

Best Practices in JavaScript:
- Use `class` syntax for defining classes.
- Prefer `const` and `let` over `var` for variable declarations.
- Use arrow functions for concise syntax.

Common Pitfalls to Avoid:
- JavaScript is dynamically typed, so ensure proper type checking.
- Be mindful of asynchronous operations in JavaScript.

Overall, migrating the given Java code to JavaScript involves creating a JavaScript class with similar properties and utilizing ES6 features for a more modern and efficient codebase.

80. src/main/java/com/example/momcare/payload/request/SocialCommentNewRequest.java

Java to JavaScript Migration Guide

com.example.momcare.payload.request.SocialCommentNewRequest

Summary:
The given Java code defines a request payload class `SocialCommentNewRequest` with attributes related to a social media comment.

Migration Plan:
1. **Create a JavaScript class with equivalent properties.**
2. **Update the getter and setter methods to JavaScript conventions.**
3. **Consider using ES6 class syntax for defining the class.**
4. **Ensure compatibility with JavaScript data types.**

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class SocialCommentNewRequest {
    private String postId;
    private String userId;
    private String commentId;
    private String imageUrl;
    private String description;
}
```

After Migration (JavaScript):
```javascript
class SocialCommentNewRequest {
```

```
    constructor() {
        this.postId = '';
        this.userId = '';
        this.commentId = '';
        this.imageUrl = '';
        this.description = '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **No specific libraries are required for this migration.**

Best Practices and Idioms in JavaScript:
- **Use ES6 class syntax for defining classes.**
- **Follow camelCase naming conventions for variables and properties.**

Common Pitfalls to Avoid:
- **Ensure that data types are handled correctly during migration.**
- **Be mindful of case sensitivity in property names.**

By following this migration guide, you can effectively convert the provided Java code to JavaScript while maintaining functionality and best practices.

81. src/main/java/com/example/momcare/payload/request/SocialCommentUpdateRequest.java

Migration Guide: Java to JavaScript

File: SocialCommentUpdateRequest.java

Summary:
The code defines a Java class `SocialCommentUpdateRequest` which represents a request payload for updating a social comment. It contains fields such as postId, id, userId, description, imageUrl, reaction, and userIdReaction.

Migration Plan:
1. Create a JavaScript class with the same name and fields.
2. Use ES6 class syntax to define the class.
3. Update the getter and setter methods to use JavaScript conventions.
4. Make sure to handle the `SocialReaction` type appropriately in JavaScript.

Code Snippets:
Before (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.SocialReaction;
import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class SocialCommentUpdateRequest {
    private String postId;
    private String id;
    private String userId;
    private String description;
    private String imageUrl;
    private SocialReaction reaction;
    private String userIdReaction;
}
```

```

#### After (JavaScript):
```javascript
class SocialCommentUpdateRequest {
    constructor() {
        this.postId = '';
        this.id = '';
        this.userId = '';
        this.description = '';
        this.imageUrl = '';
        this.reaction = {}; // Handle SocialReaction appropriately
        this.userIdReaction = '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Avoid direct manipulation of class properties.

Common Pitfalls to Avoid:
- Ensure proper handling of object types like `SocialReaction`.

General Notes:
- This migration involves straightforward conversion of a Java class to a JavaScript class without any significant complexities.
```

82. src/main/java/com/example/momcare/payload/request/SocialPostNewRequest.java

Migration Guide: Java to JavaScript

SocialPostNewRequest Class Migration

Summary:
The Java class `SocialPostNewRequest` is a payload request class used for social media post creation. It contains fields for post description, user information, and media attachments.

Migration Plan:
1. Convert the class to a JavaScript object.
2. Use ES6 class syntax to define the object structure.
3. Update getter and setter methods to JavaScript conventions.
4. Replace Java List with JavaScript arrays.
5. Consider using ES6 modules for better code organization.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.Media;
import lombok.Getter;
import lombok.Setter;

import java.util.List;
import java.util.Set;
@Getter
@Setter
public class SocialPostNewRequest
{
    private String description;
    private String userId;
    private String userName;
    private String displayName;
    private String avtUrl;
    private List<Media> media;
```

}
```

### After Migration (JavaScript):
```javascript
class SocialPostNewRequest {
    constructor() {
        this.description = '';
        this.userId = '';
        this.userName = '';
        this.displayName = '';
        this.avtUrl = '';
        this.media = [];
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries or frameworks are required for this migration, as it involves basic object structure conversion.

Best Practices and Idioms:
- Utilize ES6 class syntax for object-oriented programming.
- Use default parameter values for initializing properties in the constructor.
- Follow JavaScript naming conventions for variables and functions.

Common Pitfalls to Avoid:
- Be mindful of data type differences between Java and JavaScript (e.g., Java String vs. JavaScript string).
- Ensure proper handling of asynchronous operations if interacting with APIs in JavaScript.

Overall Migration Considerations:
- Pay attention to data types and object structures during the conversion process.
- Test the migrated code thoroughly to ensure functionality remains intact.
- Refactor any Java-specific libraries or dependencies to their JavaScript equivalents.
```

83. src/main/java/com/example/momcare/payload/request/SocialPostUpdateResquest.java

Migration Guide from Java to JavaScript

SocialPostUpdateRequest Class

Summary:
The provided code defines a Java class `SocialPostUpdateRequest` which represents a request payload for updating a social media post. It includes fields for id, description, media (list of Media objects), and reactions (map of String to SocialReaction objects).

Migration Plan:
1. Create a JavaScript class `SocialPostUpdateRequest`.
2. Define properties for id, description, media (array of Media objects), and reaction (object with keys as strings and values as SocialReaction objects).
3. Update getter and setter methods to use JavaScript syntax.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.Media;
import com.example.momcare.models.SocialReaction;
import lombok.Getter;
import lombok.Setter;

import java.util.List;
import java.util.Map;
import java.util.Set;
@Getter
@Setter
public class SocialPostUpdateResquest {
    private String id;
    private String description;
    private List<Media> media;
    private Map<String, SocialReaction> reaction;
}
```

```

### After Migration (JavaScript):
```javascript
class SocialPostUpdateRequest {
    constructor() {
        this.id = '';
        this.description = '';
        this.media = [];
        this.reaction = {};
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration, as it involves basic JavaScript syntax.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use array literals `[]` for defining arrays and object literals `{}` for defining objects.

Common Pitfalls to Avoid:
- Pay attention to the differences in syntax between Java and JavaScript, especially in defining classes and properties.

Overall Migration Considerations:
- This migration involves mainly syntactical changes from Java to JavaScript.
- Ensure all dependencies are resolved in the JavaScript environment.
- Test thoroughly to validate the behavior of the migrated code.
```

84. src/main/java/com/example/momcare/payload/request/SocialReactionDeleteRequest.java

Migration Guide: Java to JavaScript

SocialReactionDeleteRequest Class

Summary:
The code defines a Java class `SocialReactionDeleteRequest` with two private String fields `postId` and `id`, along with getters and setters generated using Lombok annotations.

Migration Plan:
1. **Create a JavaScript class with equivalent fields and methods.**
2. **Use ES6 class syntax for defining the class.**
3. **Manually create getter and setter methods for the fields.**

Code Snippets:
Before:
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class SocialReactionDeleteRequest {
    private String postId;
    private String id;
}
```

After:
```javascript
class SocialReactionDeleteRequest {
    constructor() {
        this.postId = '';
        this.id = '';
```

```
    }

    getPostId() {
        return this.postId;
    }

    setPostId(postId) {
        this.postId = postId;
    }

    getId() {
        return this.id;
    }

    setId(id) {
        this.id = id;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for class definitions.
- Manually implement getter and setter methods as needed.

Common Pitfalls to Avoid:
- Not initializing variables in the constructor can lead to unexpected behavior.

By following this migration guide, you can successfully convert the given Java code to JavaScript.

85. src/main/java/com/example/momcare/payload/request/SocialReactionNewRequest.java

Migration Guide: Java to JavaScript

File: SocialReactionNewRequest.java

Summary:
The `SocialReactionNewRequest` class in Java represents a payload request for social reactions, containing information such as postId, userId, and reaction.

Migration Plan:
1. Create a new JavaScript class `SocialReactionNewRequest`.
2. Define properties `postId`, `userId`, and `reaction` in the JavaScript class.
3. Implement getter and setter methods for the properties.
4. Consider using JavaScript naming conventions for properties and methods.

Before (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.Reaction;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class SocialReactionNewRequest {
    private String postId;
    private String userId;
    private Reaction reaction;
}
```

After (JavaScript):
```javascript
class SocialReactionNewRequest {
    constructor() {
```

```
        this.postId = '';
        this.userId = '';
        this.reaction = null;
    }

    getPostId() {
        return this.postId;
    }

    setPostId(postId) {
        this.postId = postId;
    }

    getUserId() {
        return this.userId;
    }

    setUserId(userId) {
        this.userId = userId;
    }

    getReaction() {
        return this.reaction;
    }

    setReaction(reaction) {
        this.reaction = reaction;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries/frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes in JavaScript.
- Use constructor function for initializing class properties.

Common Pitfalls to Avoid:
- Ensure proper conversion of getter and setter methods from Java to JavaScript.
- Be mindful of differences in syntax and conventions between Java and JavaScript.

This is a basic example of migrating a Java class to JavaScript. Depending on the complexity of the Java codebase, the migration process may involve additional steps and considerations.

86. src/main/java/com/example/momcare/payload/request/SocialReactionUpdateRequest.java

Java to JavaScript Migration Guide

Summary:
The given Java code defines a request payload class `SocialReactionUpdateRequest` used for updating social reactions on a post.

Migration Plan:
1. **Convert Class Definition:**
   - Create a JavaScript class with equivalent properties.
   - Replace `@Getter` and `@Setter` annotations with standard JavaScript property declarations.

2. **Update Property Types:**
   - Update property types to match JavaScript conventions.
   - Consider using TypeScript for type safety.

3. **Handle Reaction Enum:**
   - Define an equivalent enum or constant for `Reaction` in JavaScript.

4. **Update Imports:**
   - Remove Java-specific imports.
   - Replace with equivalent JavaScript imports if necessary.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.Reaction;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class SocialReactionUpdateRequest {
    private String postId;
```

```
    private String id;
    private Reaction reaction;
}
```

After Migration (JavaScript):
```javascript
class SocialReactionUpdateRequest {
    constructor() {
        this.postId = '';
        this.id = '';
        this.reaction = ReactionEnum.NONE; // Assuming ReactionEnum is defined
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **ReactJS:** For building user interfaces.
- **ExpressJS:** For backend server applications.
- **Axios:** For making HTTP requests.
- **TypeScript:** For adding static typing to JavaScript.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` for variable declarations instead of `var`.
- Follow ES6+ syntax for classes, arrow functions, and modules.
- Use Promises or async/await for asynchronous operations.

Common Pitfalls to Avoid:
- JavaScript is loosely typed, so ensure proper type handling.
- Be cautious of asynchronous operations causing unexpected behavior.
- Watch out for differences in object-oriented concepts between Java and JavaScript.

Database/API Migration Advice:
- If interacting with a database, consider using Node.js with libraries like Sequelize or MongoDB.
- For APIs, use Express.js for creating RESTful APIs and Axios for making HTTP requests.

By following this migration guide, you can successfully convert the given Java code to JavaScript while leveraging the best practices and tools available in the JavaScript ecosystem.

87. src/main/java/com/example/momcare/payload/request/SocialStoryNewRequest.java

Migration Guide: Java to JavaScript

Summary:
The given Java code defines a request payload class `SocialStoryNewRequest` with fields for description, userId, and media.

Migration Plan:
1. **Convert Class Definition:**
   - Create a JavaScript class `SocialStoryNewRequest` with properties for description, userId, and media.
2. **Replace Annotations:**
   - Remove `@Setter` and `@Getter` annotations as JavaScript does not have equivalent annotations.
3. **Update Data Types:**
   - Update data types to match JavaScript equivalents.
4. **Handle List Data Type:**
   - If `List` is used in Java, convert it to an array in JavaScript.

Before/After Code Snippets:
Java (Before):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.Media;
import lombok.Getter;
import lombok.Setter;

import java.util.List;
@Setter
@Getter
public class SocialStoryNewRequest {
    private String description;
    private String userId;
    private String media;
}
```

```

### JavaScript (After):
```javascript
class SocialStoryNewRequest {
    constructor() {
        this.description = '';
        this.userId = '';
        this.media = '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Framework:** React.js or Angular
- **Library:** Lodash for utility functions

Best Practices and Idioms in JavaScript:
- Use ES6 classes for defining classes.
- Use camelCase for variable and function names.
- Use const and let for variable declarations.

Common Pitfalls to Avoid:
- Misunderstanding asynchronous nature of JavaScript.
- Not handling null or undefined values properly.

By following this migration guide, you can successfully convert the given Java code to JavaScript. Remember to test thoroughly after migration to ensure the functionality remains intact.
```

88. src/main/java/com/example/momcare/payload/request/StandIndexRequest.java

Migration Guide: Java to JavaScript

StandIndexRequest Class Migration

Summary:
The `StandIndexRequest` class is a Java class that represents a payload request object with two String fields `datePregnant` and `dateEnd`.

Migration Plan:
1. Create a JavaScript class `StandIndexRequest` with equivalent properties.
2. Remove the `@Setter` and `@Getter` annotations as JavaScript does not have such annotations.
3. Use getter and setter methods to access and modify the properties.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class StandIndexRequest {
    private String datePregnant;
    private String dateEnd;
}
```

After Migration (JavaScript):
```javascript
class StandIndexRequest {
    constructor() {
        this.datePregnant = '';
        this.dateEnd = '';
```

```
    }

    getDatePregnant() {
        return this.datePregnant;
    }

    setDatePregnant(datePregnant) {
        this.datePregnant = datePregnant;
    }

    getDateEnd() {
        return this.dateEnd;
    }

    setDateEnd(dateEnd) {
        this.dateEnd = dateEnd;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 classes for object-oriented programming.
- Use getter and setter methods for property access.

Common Pitfalls to Avoid:
- Avoid directly accessing class properties to maintain encapsulation.

Overall Migration Advice:
- Ensure compatibility with the target environment (e.g., browser vs. Node.js).
- Refactor any Java-specific logic or libraries to JavaScript equivalents.
- Test thoroughly to ensure the functionality remains intact after migration.

89. src/main/java/com/example/momcare/payload/request/UserHandlerRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.UserHandlerRequest

Summary:
This Java code defines a simple UserHandlerRequest class with an 'id' field using Lombok annotations for getter and setter methods.

Migration Plan:
1. Create a JavaScript class named UserHandlerRequest.
2. Define a constructor with an 'id' property.
3. Implement getter and setter methods manually in JavaScript.

Before (Java):
```java
package com.example.momcare.payload.request;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class UserHandlerRequest {
    private String id;
}
```

After (JavaScript):
```javascript
class UserHandlerRequest {
    constructor(id) {
        this.id = id;
    }

    getId() {
```

```
        return this.id;
    }

    setId(id) {
        this.id = id;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific library/framework required for this simple migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Avoid using getters and setters unless necessary due to performance reasons.

Common Pitfalls to Avoid:
- Ensure that the naming conventions and access modifiers are consistent with the Java code.

This migration guide provides a basic example of converting a Java class with Lombok annotations to a JavaScript class. Additional considerations may be needed depending on the complexity of the Java codebase being migrated.

90. src/main/java/com/example/momcare/payload/request/UserLoginRequest.java

Migration Guide: Java to JavaScript

UserLoginRequest Class Migration

Summary:
The given Java code represents a `UserLoginRequest` class with fields for `username` and `password`.

Migration Plan:
1. Create a JavaScript class with equivalent properties.
2. Implement getter and setter methods in JavaScript class.
3. Add constructors for initialization.
4. Update any dependencies related to `lombok` annotations.

Before (Java):
```java
package com.example.momcare.payload.request;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserLoginRequest {
    private String username;
    private String password;
}
```

After (JavaScript):
```javascript
```

```
class UserLoginRequest {
    constructor(username, password) {
        this.username = username;
        this.password = password;
    }

    getUsername() {
        return this.username;
    }

    setUsername(username) {
        this.username = username;
    }

    getPassword() {
        return this.password;
    }

    setPassword(password) {
        this.password = password;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific library/framework needed for this migration.

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes in JavaScript.
- Use constructor functions for object initialization.

Common Pitfalls to Avoid:
- Ensure proper handling of data types as JavaScript is loosely typed.

Overall Migration Advice:
- Ensure to handle any platform-specific dependencies or configurations.
- Test thoroughly after migration to ensure functionality remains intact.

91. src/main/java/com/example/momcare/payload/request/UserRequest.java

Java to JavaScript Migration Guide

`UserRequest` Class Migration

Summary:
The `UserRequest` class is a payload request class that contains various user information fields.

Migration Plan:
1. Remove Java-specific annotations like `@Getter`, `@Setter`, `@NoArgsConstructor`, `@AllArgsConstructor`.
2. Convert Java data types to JavaScript equivalents.
3. Use ES6 classes or object literals to define the class structure.
4. Replace Java collections with JavaScript arrays or objects.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.BabyHealthIndex;
import com.example.momcare.models.MomHealthIndex;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.List;
import java.util.Set;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserRequest {
    private String id;
    private String userName;
```

```
    private String passWord;
    private String email;
    private String avtUrl;
    private String nameDisplay;
    private Set<String> follower;
    private Set<String> following;
    private Set<String> shared;
    private String datePregnant;
    private Boolean premium;
    private List<MomHealthIndex> momIndex;
    private List<BabyHealthIndex> babyIndex;
}
```

After Migration (JavaScript):
```javascript
class UserRequest {
    constructor(id, userName, passWord, email, avtUrl, nameDisplay, follower, following, shared,
datePregnant, premium, momIndex, babyIndex) {
        this.id = id;
        this.userName = userName;
        this.passWord = passWord;
        this.email = email;
        this.avtUrl = avtUrl;
        this.nameDisplay = nameDisplay;
        this.follower = new Set(follower);
        this.following = new Set(following);
        this.shared = new Set(shared);
        this.datePregnant = datePregnant;
        this.premium = premium;
        this.momIndex = momIndex;
        this.babyIndex = babyIndex;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 features like classes and spread syntax for object manipulation.
- Use `const` and `let` for variable declarations instead of `var`.
- Avoid using `Set` data structure if compatibility with older browsers is a concern.

Common Pitfalls to Avoid:
- Ensure proper handling of null values as JavaScript does not have strict type checking like Java.
- Be mindful of differences in object initialization and property access between Java and JavaScript.

Overall Migration Advice:
- Pay attention to data types and conversions when migrating from Java to JavaScript.
- Test thoroughly to ensure the behavior remains consistent after migration.
- Consider refactoring code to align with JavaScript coding standards and best practices.

92. src/main/java/com/example/momcare/payload/request/UserSignUpRequest.java

Migration Guide: Java to JavaScript

UserSignUpRequest Class

Summary:
The `UserSignUpRequest` class is a payload request class with fields for user information like username, email, name display, and password.

Migration Plan:
1. Create a JavaScript class with the same fields.
2. Use ES6 class syntax to define the class.
3. Use getter and setter methods for data encapsulation.
4. Update import statements and annotations.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class UserSignUpRequest {
    private String userName;
    private String email;
    private String nameDisplay;
    private String password;
}
```

After Migration (JavaScript):
```javascript
class UserSignUpRequest {
    constructor(userName, email, nameDisplay, password) {
        this.userName = userName;
        this.email = email;
        this.nameDisplay = nameDisplay;
        this.password = password;
    }

    getUserName() {
        return this.userName;
    }

    setUserName(userName) {
        this.userName = userName;
    }

    getEmail() {
        return this.email;
    }

    setEmail(email) {
        this.email = email;
    }

    getNameDisplay() {
        return this.nameDisplay;
    }

    setNameDisplay(nameDisplay) {
        this.nameDisplay = nameDisplay;
    }

    getPassword() {
        return this.password;
    }
```

```
    setPassword(password) {
        this.password = password;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- None required for this specific migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use getter and setter methods for data access and manipulation.

Common Pitfalls to Avoid:
- Ensure proper scoping of variables and methods in JavaScript classes.
- Handle data types and type conversions carefully during migration.

By following this migration guide, you can effectively convert the provided Java code for `UserSignUpRequest` into JavaScript. Remember to test thoroughly after migration to ensure correctness and functionality.

93. src/main/java/com/example/momcare/payload/request/UserStoryDeleteRequest.java

Migration Guide: Java to JavaScript

com.example.momcare.payload.request.UserStoryDeleteRequest

Summary:
This Java code defines a request payload object `UserStoryDeleteRequest` that contains a user ID and a list of social stories to be deleted.

Migration Plan:
1. Create a JavaScript class `UserStoryDeleteRequest`.
2. Define properties `userId` and `socialStories`.
3. Implement getter and setter methods for the properties.

Before Migration (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.SocialStory;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Setter
@Getter
public class UserStoryDeleteRequest {
    private String userId;
    private List<SocialStory> socialStories;
}
```

After Migration (JavaScript):
```javascript
class UserStoryDeleteRequest {
    constructor() {
```

```
        this.userId = '';
        this.socialStories = [];
    }

    getUserId() {
        return this.userId;
    }

    setUserId(userId) {
        this.userId = userId;
    }

    getSocialStories() {
        return this.socialStories;
    }

    setSocialStories(socialStories) {
        this.socialStories = socialStories;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Prefer using `constructor` for initializing properties.

Common Pitfalls to Avoid:
- JavaScript does not have built-in annotations like `@Getter` and `@Setter` in Java, so manual getter and setter methods need to be implemented.

Overall Migration Advice:
- Ensure compatibility with JavaScript data structures and syntax.
- Test thoroughly to validate functionality after migration.
- Consider using TypeScript for stricter type checking during migration if needed.
- Update any dependencies on Java-specific libraries to their JavaScript equivalents.

94. src/main/java/com/example/momcare/payload/request/UserStoryNewRequest.java

Migration Guide: Java to JavaScript

File: UserStoryNewRequest.java

Summary:
The `UserStoryNewRequest` class represents a request payload for creating a new user story associated with a user.

Migration Plan:
1. Create a new JavaScript class `UserStoryNewRequest` with similar properties.
2. Replace `@Getter` and `@Setter` annotations with equivalent property definitions in JavaScript.
3. Update import statements if needed.

Before (Java):
```java
package com.example.momcare.payload.request;

import com.example.momcare.models.SocialStory;
import lombok.Getter;
import lombok.Setter;

import java.util.List;
@Getter
@Setter
public class UserStoryNewRequest {
    private String userId;
    private SocialStory socialStory;
}
```

After (JavaScript):
```javascript
class UserStoryNewRequest {
    constructor() {
```

```
        this.userId = '';
        this.socialStory = null;
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries/frameworks required for this migration.

Best Practices in JavaScript:
- Use `class` syntax for defining classes in JavaScript.
- Initialize properties in the constructor for better readability.

Common Pitfalls to Avoid:
- Remember to handle null values appropriately in JavaScript.

Overall, this migration involves creating a JavaScript class equivalent to the Java class `UserStoryNewRequest` with similar properties and no additional dependencies.

95. src/main/java/com/example/momcare/payload/response/HandBookDetailResponse.java

Java to JavaScript Migration Guide

Summary:
The provided Java code defines a `HandBookDetailResponse` class in a package called `com.example.momcare.payload.response`. The class has three fields: `status`, `data` of type `HandBook`, and `message`. It also has a constructor to initialize these fields.

Migration Plan:
1. **Class Conversion**:
   - Create a JavaScript class `HandBookDetailResponse`.
   - Define properties `status`, `data`, and `message` in the constructor.

2. **Getters and Setters**:
   - Use ES6 class syntax to define getter and setter methods for the properties.

3. **Dependencies**:
   - Check for any additional dependencies used in the `HandBook` class and ensure they are also migrated.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.HandBook;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class HandBookDetailResponse {
    private String status;
    private HandBook data;
    private String message;

    public HandBookDetailResponse(String status, HandBook data, String message) {
```

```
        this.status = status;
        this.data = data;
        this.message = message;
    }
}
```

After Migration (JavaScript):
```javascript
class HandBookDetailResponse {
    constructor(status, data, message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }

    get status() {
        return this.status;
    }

    set status(status) {
        this.status = status;
    }

    get data() {
        return this.data;
    }

    set data(data) {
        this.data = data;
    }

    get message() {
        return this.message;
    }

    set message(message) {
        this.message = message;
```

```
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **No additional libraries or frameworks are needed for this migration.**

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Avoid using `getter` and `setter` method names that clash with property names.

Common Pitfalls to Avoid:
- Accidentally reusing property names for getter and setter methods.
- Not handling asynchronous code properly if any API calls are involved in the migration.

96. src/main/java/com/example/momcare/payload/response/MenuDetailResponse.java

Migration Guide: Java to JavaScript

File: MenuDetailResponse.java

Summary:
The `MenuDetailResponse` class is a response payload class in Java used to encapsulate status, data, and message for a menu detail response.

Migration Plan:
1. Create a JavaScript class with equivalent properties.
2. Use ES6 class syntax for the JavaScript class.
3. Update the constructor to match the Java constructor.
4. Implement getter and setter methods if needed.
5. Consider using TypeScript for static type checking.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.Menu;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class MenuDetailResponse {
    private String status;
    private Menu data;
    private String message;

    public MenuDetailResponse(String status, Menu data, String message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }
```

}
```

### After Migration (JavaScript):
```javascript
class MenuDetailResponse {
    constructor(status, data, message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for object-oriented programming.
- Avoid using getter and setter methods unless necessary.

Common Pitfalls to Avoid:
- Make sure to handle data types correctly as JavaScript is dynamically typed.
- Watch for potential issues with object serialization/deserialization when interacting with APIs.

Overall Notes:
- The migration of the `MenuDetailResponse` class from Java to JavaScript involves creating a class with similar properties and constructor in JavaScript. Remember to handle data types carefully and consider utilizing TypeScript for additional type safety.
```

97. src/main/java/com/example/momcare/payload/response/NotificationResponse.java

Migration Guide: Java to JavaScript

NotificationResponse Class

Summary:
The `NotificationResponse` class is a data transfer object (DTO) representing a notification response object with various properties.

Migration Plan:
1. Create a JavaScript class or object to mirror the structure of the `NotificationResponse` class.
2. Use the equivalent of Java annotations (like `@Getter`, `@Setter`, `@AllArgsConstructor`, `@NoArgsConstructor`) in JavaScript.

Before (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.NotificationType;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Setter
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class NotificationResponse {
    private String id;
    private String senderId;
    private String receiverId;
    private String timestamp;
    private boolean isRead;
    private NotificationType notificationType;
    private String senderName;
```

```
    private String senderAvt;
    private String targetId;
}
```

After (JavaScript):
```javascript
class NotificationResponse {
    constructor(id, senderId, receiverId, timestamp, isRead, notificationType, senderName,
senderAvt, targetId) {
        this.id = id;
        this.senderId = senderId;
        this.receiverId = receiverId;
        this.timestamp = timestamp;
        this.isRead = isRead;
        this.notificationType = notificationType;
        this.senderName = senderName;
        this.senderAvt = senderAvt;
        this.targetId = targetId;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Prefer explicit property assignments in the constructor.

Common Pitfalls to Avoid:
- Be mindful of JavaScript's loose typing compared to Java.
- Pay attention to differences in naming conventions between Java and JavaScript.

98. src/main/java/com/example/momcare/payload/response/Response.java

Migration Guide: Java to JavaScript

Response Class

Summary:
The `Response` class is a payload response class with status, data, and message fields.

Migration Plan:
1. Create a JavaScript class with similar fields and a constructor.
2. Use ES6 class syntax for defining the class.
3. Utilize TypeScript for static typing if needed.

Before:
```java
package com.example.momcare.payload.response;

import lombok.Getter;
import lombok.Setter;

import java.util.List;
@Getter
@Setter
public class Response {
    private String status;
    private List<?> data;
    private String message;

    public Response(String status, List<?> data, String message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }
}
```

After:
```javascript
class Response {
    constructor(status, data, message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }
}
```

Recommended Libraries/Frameworks:
- TypeScript for static typing
- Lodash for utility functions on arrays

Best Practices:
- Use ES6 class syntax for defining classes
- Use explicit data types when defining class fields

Common Pitfalls:
- Pay attention to data types as JavaScript is loosely typed
- Ensure consistent handling of data structures when migrating list objects

Database/API Usage

Migration Advice:
1. Replace Java database access libraries with JavaScript equivalents like Sequelize for SQL databases or MongoDB Node.js driver for MongoDB.
2. Utilize Express.js for building RESTful APIs in JavaScript.

By following this migration guide, you can successfully convert the provided Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

99. src/main/java/com/example/momcare/payload/response/SocialCommentResponse.java

Java to JavaScript Migration Guide

Summary:
The provided Java code defines a class `SocialCommentResponse` that represents a response payload for social comments. It contains various fields like `id`, `userId`, `userName`, etc., to store information about a social comment.

Migration Plan:
1. **Convert Java Class to JavaScript Class:**
   - Create a JavaScript class with the same name `SocialCommentResponse`.
   - Define properties corresponding to the fields in the Java class.

2. **Handle Getters and Setters:**
   - In JavaScript, you can directly access properties without explicit getter and setter methods.

3. **Handle Constructor:**
   - In JavaScript, use the `constructor` method to initialize the object properties.

4. **Datatypes Mapping:**
   - Java datatypes like `String`, `Map`, and `List` need to be mapped to JavaScript equivalents.

5. **Library/Dependencies:**
   - Consider using libraries like `lodash` or `Immutable.js` for handling collections and objects.

6. **Best Practices:**
   - Use `const` and `let` for variable declarations.
   - Follow camelCase naming convention for properties and methods.

7. **Pitfalls to Avoid:**
   - Be cautious about the differences in how Java and JavaScript handle data structures and types.

Before/After Code Snippets:
Java Code:
```java
```

```
package com.example.momcare.payload.response;

import com.example.momcare.models.SocialReaction;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.List;
import java.util.Map;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class SocialCommentResponse {
    private String id;
    private String userId;
    // Other fields...
    private String time;
}
```

JavaScript Code:
```javascript
class SocialCommentResponse {
    constructor() {
        this.id = '';
        this.userId = '';
        // Initialize other fields...
        this.time = '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **lodash:** For utility functions to manipulate objects and arrays.
- **Immutable.js:** For handling immutable data structures.

Remember to install these libraries using npm or yarn.

This guide outlines the necessary steps and considerations for migrating the provided Java code to JavaScript. Follow the migration plan carefully and leverage JavaScript best practices to ensure a successful migration.

100. src/main/java/com/example/momcare/payload/response/SocialPostResponse.java

Java to JavaScript Migration Guide

SocialPostResponse Class Migration

Summary:
The Java code defines a `SocialPostResponse` class that represents a response object for social media posts containing various attributes like id, description, user details, reactions, comments, media, etc.

Migration Plan:
1. Define a class `SocialPostResponse` in JavaScript with equivalent properties.
2. Use ES6 class syntax for defining the class.
3. Replace Java annotations with JavaScript class properties.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.Media;
import com.example.momcare.models.SocialReaction;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.List;
import java.util.Map;
import java.util.Set;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class SocialPostResponse {
    private String id;
```

```
    private String description;
    private String userId;
    private String userName;
    private String displayName;
    private String avtUrl;
    private Map<String, SocialReaction> reactions;
    private Set<String> comments;
    private Set<String> share;
    private List<Media> media;
    private String time;
}
```

After Migration (JavaScript):
```javascript
class SocialPostResponse {
    constructor() {
        this.id = '';
        this.description = '';
        this.userId = '';
        this.userName = '';
        this.displayName = '';
        this.avtUrl = '';
        this.reactions = new Map();
        this.comments = new Set();
        this.share = new Set();
        this.media = [];
        this.time = '';
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Initialize object properties in the constructor.

Common Pitfalls to Avoid:
- Be cautious with data types as JavaScript is dynamically typed.

Conclusion:
The migration of the `SocialPostResponse` class from Java to JavaScript involves defining a class with equivalent properties and using ES6 class syntax for the definition. Pay attention to data types and object initialization to ensure a smooth migration.

101. src/main/java/com/example/momcare/payload/response/SocialReactionResponse.java

Migration Guide: Java to JavaScript

SocialReactionResponse Class

Summary:
The `SocialReactionResponse` class is a payload response class that contains data related to social reactions. It includes fields for avatar URL, display name, time, and reaction.

Migration Plan:
1. Create a JavaScript class or object to represent the `SocialReactionResponse`.
2. Define properties for avatar URL, display name, time, and reaction.
3. Use ES6 class syntax or object literals to define the structure.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.Reaction;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

import java.time.LocalTime;

@Getter
@Setter
@AllArgsConstructor
public class SocialReactionResponse {
    private String avtUrl;
    private String displayName;
    private String time;
    private Reaction reaction;
}
```

After Migration (JavaScript):
```javascript
class SocialReactionResponse {
    constructor(avtUrl, displayName, time, reaction) {
        this.avtUrl = avtUrl;
        this.displayName = displayName;
        this.time = time;
        this.reaction = reaction;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use constructor functions for initializing object properties.

Common Pitfalls to Avoid:
- JavaScript does not have built-in support for annotations like Lombok in Java. Manual property assignment may be required.

Conclusion:
By following the migration plan outlined above, you can convert the Java `SocialReactionResponse` class to JavaScript successfully. Remember to adhere to best practices and avoid common pitfalls during the migration process.

102. src/main/java/com/example/momcare/payload/response/StandardsBabyIndexResponse.java

Migration Guide from Java to JavaScript

Summary:
The given Java code defines a response payload class `StandardsBabyIndexResponse` with attributes representing different standards indices related to baby growth.

Migration Plan:
1. **Create a JavaScript class `StandardsBabyIndexResponse` with the same attributes.**
2. **Use ES6 class syntax for defining classes in JavaScript.**
3. **Implement getter and setter methods for each attribute in JavaScript.**
4. **Consider using appropriate JavaScript libraries for data manipulation if needed.**

Code Migration:

Java code:
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.StandardsIndex;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class StandardsBabyIndexResponse {
    private StandardsIndex head;
    private StandardsIndex biparietal;
    private StandardsIndex occipitofrontal;
    private StandardsIndex abdominal;
    private StandardsIndex femur;
}
```

JavaScript code:
```javascript
```

```
class StandardsBabyIndexResponse {
    constructor() {
        this.head = null;
        this.biparietal = null;
        this.occipitofrontal = null;
        this.abdominal = null;
        this.femur = null;
    }

    getHead() {
        return this.head;
    }

    setHead(value) {
        this.head = value;
    }

    getBiparietal() {
        return this.biparietal;
    }

    setBiparietal(value) {
        this.biparietal = value;
    }

    getOccipitofrontal() {
        return this.occipitofrontal;
    }

    setOccipitofrontal(value) {
        this.occipitofrontal = value;
    }

    getAbdominal() {
        return this.abdominal;
    }

    setAbdominal(value) {
```

```
        this.abdominal = value;
    }

    getFemur() {
        return this.femur;
    }

    setFemur(value) {
        this.femur = value;
    }
}

```

Recommended Libraries/Frameworks in JavaScript:
- **No specific libraries are needed for this code migration.**
- **However, using a library like Lodash can simplify data manipulation tasks.**

Best Practices and Idioms in JavaScript:
- **Use ES6 class syntax for defining classes.**
- **Avoid using getter/setter methods if not necessary.**
- **Follow consistent naming conventions as per JavaScript standards.**

Common Pitfalls to Avoid:
- **JavaScript is loosely typed compared to Java, so handle datatype conversions carefully.**
- **Ensure proper initialization of variables in the constructor.**

This migration guide provides a step-by-step approach to convert the given Java code to equivalent JavaScript code for the `StandardsBabyIndexResponse` class.

103. src/main/java/com/example/momcare/payload/response/StandardsMomIndexResponse.java

Java to JavaScript Migration Guide

Summary:
The given Java code defines a response payload class `StandardsMomIndexResponse` containing multiple instances of the `StandardsIndex` class.

Migration Plan:
1. **Create a JavaScript class for StandardsMomIndexResponse**
2. **Convert class fields to properties**
3. **Update getter and setter methods**

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.StandardsIndex;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class StandardsMomIndexResponse {
    private StandardsIndex BMI;
    private StandardsIndex HATT;
    private StandardsIndex HATTr;
    private StandardsIndex GIHungry;
    private StandardsIndex GIFull1h;
    private StandardsIndex GIFull2h;
    private StandardsIndex GIFull3h;
}
```

After Migration (JavaScript):
```javascript
class StandardsMomIndexResponse {
```

```
    constructor() {
        this.BMI = null;
        this.HATT = null;
        this.HATTr = null;
        this.GIHungry = null;
        this.GIFull1h = null;
        this.GIFull2h = null;
        this.GIFull3h = null;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **React.js** for building user interfaces.
- **Express.js** for server-side applications.
- **Lodash** for utility functions.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var`.
- Embrace arrow functions for concise code.
- Follow ES6+ syntax for modern JavaScript development.

Common Pitfalls to Avoid:
- Be mindful of asynchronous operations in JavaScript.
- Watch out for type coercion and loose equality comparisons.

Overall, converting the given Java code to JavaScript involves translating class definitions, updating syntax, and adjusting to JavaScript's dynamic nature. By following the migration plan and best practices, the transition can be smooth and successful.

104. src/main/java/com/example/momcare/payload/response/TrackingWeekDetailResponse.java

Migration Guide: Java to JavaScript

TrackingWeekDetailResponse Class

Summary:
The `TrackingWeekDetailResponse` class represents a response payload object containing details related to tracking a week of a mom and baby. It includes fields like id, week number, name, baby details, mom details, advice, and a list of thumbnails.

Migration Plan:
1. Create a JavaScript class `TrackingWeekDetailResponse` with equivalent properties.
2. Update the constructor to initialize the object with the provided parameters.
3. Use ES6 class syntax for defining the class.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import lombok.Getter;
import lombok.Setter;

import java.util.List;
@Getter
@Setter
public class TrackingWeekDetailResponse {
    private String id;
    private int week;
    private String name;
    private String baby;
    private String mom;
    private String advice;
    private List<String> thumbnail;

    public TrackingWeekDetailResponse(String id, int week, String name, String baby, String mom, String advice, List<String> thumbnail) {
```

```
        this.id = id;
        this.week = week;
        this.name = name;
        this.baby = baby;
        this.mom = mom;
        this.advice = advice;
        this.thumbnail = thumbnail;
    }
}
```

After Migration (JavaScript):
```javascript
class TrackingWeekDetailResponse {
    constructor(id, week, name, baby, mom, advice, thumbnail) {
        this.id = id;
        this.week = week;
        this.name = name;
        this.baby = baby;
        this.mom = mom;
        this.advice = advice;
        this.thumbnail = thumbnail;
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries or frameworks required for this migration.

Best Practices and Idioms:
- Use ES6 class syntax for defining classes.
- Prefer `const` and `let` over `var` for variable declarations.
- Utilize destructuring assignment for object properties where applicable.

Common Pitfalls to Avoid:
- Ensure proper handling of asynchronous operations if any asynchronous code is involved in the migration.
- Be cautious of data types conversion differences between Java and JavaScript.

Additional Notes:
- This migration guide covers the conversion of the provided Java code snippet to equivalent JavaScript code for the `TrackingWeekDetailResponse` class. Further migration steps for other parts of the codebase may require separate analysis and planning.

105. src/main/java/com/example/momcare/payload/response/TrackingWeekResponse.java

Java to JavaScript Migration Guide

TrackingWeekResponse Class

Summary:
The `TrackingWeekResponse` class is a simple Java class with getter and setter methods for `id`, `week`, `name`, and `type` fields.

Migration Plan:
1. Create a JavaScript class to represent `TrackingWeekResponse`.
2. Use ES6 class syntax to define the class.
3. Implement a constructor function to initialize the class properties.
4. Use ES6 getter and setter methods for accessing and modifying class properties.

Before:
```java
package com.example.momcare.payload.response;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class TrackingWeekResponse {
    private String id;
    private int week;
    private String name;
    private String type;

    public TrackingWeekResponse(String id, int week, String name, String type) {
        this.id = id;
        this.week = week;
        this.name = name;
        this.type = type;
    }
```

}
```

### After:
```javascript
class TrackingWeekResponse {
    constructor(id, week, name, type) {
        this.id = id;
        this.week = week;
        this.name = name;
        this.type = type;
    }

    get getId() {
        return this.id;
    }

    set setId(id) {
        this.id = id;
    }

    get getWeek() {
        return this.week;
    }

    set setWeek(week) {
        this.week = week;
    }

    get getName() {
        return this.name;
    }

    set setName(name) {
        this.name = name;
    }

```
    get getType() {
        return this.type;
    }

    set setType(type) {
        this.type = type;
    }
}
```

Recommended Libraries/Frameworks:
- No specific libraries or frameworks are necessary for this migration.

Best Practices:
- Use ES6 class syntax for defining classes in JavaScript.
- Use getter and setter methods for class properties.
- Follow naming conventions for methods and properties.

Common Pitfalls:
- Ensure that the class properties are properly initialized in the constructor.
- Make sure to define getter and setter methods for each property that requires access control.

106. src/main/java/com/example/momcare/payload/response/UserLoginResponse.java

Java to JavaScript Migration Guide

UserLoginResponse Class

Summary:
The `UserLoginResponse` class in Java is a simple data transfer object (DTO) that contains information about a user's login response, such as token, id, and role.

Migration Plan:
1. Create a new JavaScript class `UserLoginResponse` with properties `token`, `id`, and `role`.
2. Use ES6 class syntax to define the class and properties.
3. Remove the annotations used in Java as JavaScript does not support annotations.
4. Use getter and setter methods to access and set the properties of the class.

Before (Java):
```java
package com.example.momcare.payload.response;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@AllArgsConstructor
@NoArgsConstructor
@Setter
@Getter
public class UserLoginResponse {
    private String token;
    private String id;
    private String role;
}
```

After (JavaScript):

```javascript
class UserLoginResponse {
    constructor() {
        this.token = '';
        this.id = '';
        this.role = '';
    }

    getToken() {
        return this.token;
    }

    setToken(token) {
        this.token = token;
    }

    getId() {
        return this.id;
    }

    setId(id) {
        this.id = id;
    }

    getRole() {
        return this.role;
    }

    setRole(role) {
        this.role = role;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are required for this migration.

Best Practices and Idioms in JavaScript:

- Use ES6 class syntax for defining classes.
- Use getter and setter methods to access and set class properties.

Common Pitfalls to Avoid:
- Remember that JavaScript does not have built-in support for annotations like Java.

Conclusion
By following this migration guide, you can successfully convert the given Java code for the `UserLoginResponse` class to JavaScript. Remember to adhere to best practices and idioms in JavaScript to ensure a smooth migration process.

107. src/main/java/com/example/momcare/payload/response/UserProfile.java

Migration Guide: Java to JavaScript

UserProfile Class Migration

Summary:
The given Java code defines a `UserProfile` class with fields such as id, username, displayName, avtUrl, followers, and share.

Migration Plan:
1. Create a JavaScript class with similar properties.
2. Use ES6 class syntax for defining the class.
3. Use JavaScript Set data structure for `followers` and `share` fields.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import java.util.Set;

@Getter
@Setter
@AllArgsConstructor
public class UserProfile {
    private String id;
    private String username;
    private String displayName;
    private String avtUrl;
    private Set<String> followers;
    private Set<String> share;
}
```

After Migration (JavaScript):
```javascript
class UserProfile {
    constructor(id, username, displayName, avtUrl, followers = new Set(), share = new Set()) {
        this.id = id;
        this.username = username;
        this.displayName = displayName;
        this.avtUrl = avtUrl;
        this.followers = followers;
        this.share = share;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are needed for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Use default parameter values for optional fields like `followers` and `share`.

Common Pitfalls to Avoid:
- Remember that JavaScript Sets are not the same as Java Sets, so behavior might differ.

Overall Migration Considerations:
- JavaScript does not have the exact equivalent of Lombok, so getters/setters need to be manually implemented.
- Pay attention to data types and conversions when migrating between Java and JavaScript.
- Consider using TypeScript for stronger typing support in JavaScript.

108. src/main/java/com/example/momcare/payload/response/UserResponse.java

Migration Guide: Java to JavaScript

UserResponse Class

Summary
The `UserResponse` class is a data class that represents a user's response with various attributes like id, username, email, etc.

Migration Plan
1. Create a JavaScript class named `UserResponse`.
2. Define the class properties similar to the Java class.
3. Implement constructors matching the Java class constructors.
4. Consider using ES6 class syntax for defining the class.
5. Use ES6 Set data structure for `follower` and `following` properties.

Before/After Code Snippets
Before (Java)
```java
public class UserResponse {
    private String id;
    private String userName;
    private String email;
    private String datePregnant;
    private Boolean premium;
    private String avtUrl;
    private Set<String> follower;
    private Set<String> following;
    private String nameDisplay;

    // Constructors
}
```

After (JavaScript)
```javascript
```

```
class UserResponse {
    constructor(id, userName, email, datePregnant, premium, avtUrl, follower = new Set(), following = new Set(), nameDisplay) {
        this.id = id;
        this.userName = userName;
        this.email = email;
        this.datePregnant = datePregnant;
        this.premium = premium;
        this.avtUrl = avtUrl;
        this.follower = follower;
        this.following = following;
        this.nameDisplay = nameDisplay;
    }
}
```

Recommended Libraries/Frameworks
- No specific libraries are required for this migration.

Best Practices and Idioms
- Use ES6 class syntax for defining classes.
- Utilize Set data structure for storing unique values in collections.

Common Pitfalls to Avoid
- Ensure proper handling of default parameter values in constructors.
- Remember that JavaScript does not have strict typing like Java, so data validation may be needed.

Conclusion
By following this migration guide, you can successfully convert the given Java code for the `UserResponse` class to JavaScript. Remember to test the migrated code thoroughly to ensure its functionality.

109. src/main/java/com/example/momcare/payload/response/UserStoryResponse.java

Migration Guide: Java to JavaScript

UserStoryResponse Class

Summary:
The `UserStoryResponse` class is a payload response class that contains user-related information along with a list of social stories.

Migration Plan:
1. Convert the class to a JavaScript object.
2. Use ES6 class syntax to define the object.
3. Update getter and setter methods to JavaScript conventions.
4. Replace `List` with an array data structure.
5. Ensure compatibility with JSON serialization.

Before Migration (Java):
```java
package com.example.momcare.payload.response;

import com.example.momcare.models.SocialStory;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.stereotype.Service;

import java.util.List;

@Setter
@Getter
@AllArgsConstructor
public class UserStoryResponse {
    private String id;
    private String userName;
    private String displayName;
    private String userId;
```

```
    private String avtUrl;
    private List<SocialStory> socialStories;
}
```

After Migration (JavaScript):
```javascript
class UserStoryResponse {
    constructor(id, userName, displayName, userId, avtUrl, socialStories) {
        this.id = id;
        this.userName = userName;
        this.displayName = displayName;
        this.userId = userId;
        this.avtUrl = avtUrl;
        this.socialStories = socialStories;
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- None required for this migration.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for object-oriented programming.
- Use camelCase for variable and method names.

Common Pitfalls to Avoid:
- Ensure that all properties are properly initialized in the constructor.
- Be mindful of data types when converting from Java to JavaScript.

Database/API Usage:
If the `UserStoryResponse` class interacts with a database or API, consider using JavaScript libraries like `axios` for API calls and `mongoose` for MongoDB interactions.

By following this migration guide, you can successfully convert the given Java code to JavaScript, ensuring compatibility and maintainability in your projects.

110. src/main/java/com/example/momcare/repository/DiaryRepository.java

Migration Guide: Java to JavaScript

com.example.momcare.repository.DiaryRepository

Summary:
The Java code defines a repository interface `DiaryRepository` for interacting with a MongoDB database. It includes methods for finding diary entries by user ID, getting a diary by ID, and searching diaries by title.

Migration Plan:
1. **Convert Interface to Class:**
   - In JavaScript, we don't have interfaces like Java, so we will create a class instead.

2. **Implement MongoDB Queries:**
   - Utilize a MongoDB Node.js driver like `mongodb` to interact with the database.

3. **Translate Query Methods:**
   - Rewrite the query methods using the driver's query syntax.

Before Migration Code (Java):
```java
// Java code here
```

After Migration Code (JavaScript):
```javascript
const { MongoClient } = require('mongodb');

class DiaryRepository {
    constructor() {
        this.client = new MongoClient('mongodb://localhost:27017');
        this.db = null;
    }

    async connect() {
```

```
        await this.client.connect();
        this.db = this.client.db('momcare');
    }

    async findAllByIdUser(idUser) {
        return await this.db.collection('diaries').find({ idUser }).toArray();
    }

    async getDiaryById(id) {
        return await this.db.collection('diaries').findOne({ _id: id });
    }

    async findByTitleLike(keyWord) {
        return await this.db.collection('diaries').find({ title: { $regex: keyWord, $options:
'i' } }).toArray();
    }
}

module.exports = DiaryRepository;
```

Recommended Libraries/Frameworks in JavaScript:
- MongoDB Node.js Driver: `mongodb`
- Express.js for building REST APIs

Best Practices and Idioms in JavaScript:
- Use `async/await` for asynchronous operations.
- Follow modular patterns with ES6 modules.

Common Pitfalls to Avoid:
- Not handling errors properly in asynchronous code.
- Incorrectly handling MongoDB connection and queries.

Database Migration Advice:
- Use the `mongodb` Node.js driver for database connectivity.
- Ensure proper error handling and connection management.
- Consider using an ORM like Mongoose for schema validation and modeling.

API Migration Advice:
- Utilize Express.js to create RESTful APIs.
- Map HTTP methods to corresponding CRUD operations.
- Implement middleware for request processing and validation.

By following this migration guide, you can successfully convert the Java code to JavaScript, maintaining functionality and best practices along the way.

111. src/main/java/com/example/momcare/repository/HandBookCategoryRepository.java

Migration Guide: Java to JavaScript

HandBookCategoryRepository

Summary:
The `HandBookCategoryRepository` is a repository interface for `HandBookCategory` model using Spring Data MongoDB. It defines a method to query and retrieve `HandBookCategory` objects based on a specific collection.

Migration Plan:
1. Create a JavaScript equivalent of the `HandBookCategory` model.
2. Implement a method to query and retrieve `HandBookCategory` objects based on a specific collection in JavaScript.

Before (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.HandBookCategory;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface HandBookCategoryRepository extends MongoRepository<HandBookCategory, String> {
    @Query("{'collection': ?0}")
    List<HandBookCategory> findByCollectionIn(String collection);
}
```

After (JavaScript):
```javascript
// MongoDB equivalent model in JavaScript
class HandBookCategory {
```

```
    constructor(name, description) {
        this.name = name;
        this.description = description;
    }
}

// MongoDB equivalent repository in JavaScript
class HandBookCategoryRepository {
    constructor(db) {
        this.collection = db.collection('handBookCategories');
    }

    async findByCollectionIn(collectionName) {
        return await this.collection.find({ collection: collectionName }).toArray();
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Node.js**: For server-side JavaScript runtime environment.
- **MongoDB Node.js Driver**: To interact with MongoDB database.

Best Practices and Idioms in JavaScript:
- Use `async/await` for asynchronous operations.
- Use ES6 classes for defining models and repositories.

Common Pitfalls to Avoid:
- Ensure proper error handling for database operations.
- Be mindful of JavaScript's asynchronous nature.

Database Migration Advice:
- Connect to MongoDB database using the MongoDB Node.js Driver.
- Use asynchronous functions for querying and updating data.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining the functionality and structure of the original code.

112. src/main/java/com/example/momcare/repository/HandBookCollectionRepository.java

Migration Guide: Java to JavaScript

1. HandBookCollectionRepository

Summary:
The `HandBookCollectionRepository` interface is used in a Spring Boot application to interact with a MongoDB database for CRUD operations on `HandBookCollection` entities.

Migration Plan:
1. **Create a MongoDB connection in JavaScript** using libraries like `mongoose`.
2. **Define a schema for `HandBookCollection`** that mirrors the Java entity.
3. **Implement CRUD operations** similar to how they are defined in the Java repository.

Before (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.HandBookCollection;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface HandBookCollectionRepository extends
MongoRepository<HandBookCollection, String> {
}
```

After (JavaScript - using mongoose):
```javascript
const mongoose = require('mongoose');

const handBookCollectionSchema = new mongoose.Schema({
  // Define schema fields similar to HandBookCollection entity
});

const HandBookCollection = mongoose.model('HandBookCollection',
handBookCollectionSchema);
```

```
module.exports = HandBookCollection;
```

Recommended Libraries/Frameworks in JavaScript:
- `mongoose` for MongoDB interaction.

Best Practices and Idioms in JavaScript:
- Use `async/await` for asynchronous operations.
- Follow naming conventions like camelCase for variables.

Common Pitfalls to Avoid:
- Ensure proper error handling for database operations.
- Validate data before saving to the database.

2. Database Migration Advice:
- Update the database connection configuration to work with JavaScript (e.g., change connection strings).
- Ensure data migration if needed when switching database connections.

This guide provides a basic outline for migrating the given Java code to JavaScript, focusing on the repository interface and MongoDB interaction. Additional steps may be needed depending on the overall architecture of the application.

113. src/main/java/com/example/momcare/repository/HandBookRepository.java

Migration Guide: Java to JavaScript

`HandBookRepository.java`

Summary:
The `HandBookRepository` interface defines methods for interacting with a MongoDB database to retrieve `HandBook` objects based on category, ID, and title.

Migration Plan:
1. **Replace Java MongoDB Driver with Mongoose in Node.js.**
2. **Define a Mongoose Schema for `HandBook` model.**
3. **Create a Mongoose Model for `HandBook`.**
4. **Implement repository methods using Mongoose queries.**

Before Migration (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.HandBook;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface HandBookRepository extends MongoRepository<HandBook,String> {
    List<HandBook> findAllByCategory(String categoryId);

    HandBook getHandBookById(String id);

    @Query("{'title': {$regex : ?0, $options: 'i'}}")
    List<HandBook> findByTitleLike(String keyWord);
}
```

After Migration (JavaScript):

```javascript
const mongoose = require('mongoose');

const handBookSchema = new mongoose.Schema({
    // Define schema fields similar to HandBook model
});

const HandBook = mongoose.model('HandBook', handBookSchema);

const HandBookRepository = {
    findAllByCategory: function(categoryId) {
        // Implement logic to find all HandBooks by category
    },

    getHandBookById: function(id) {
        // Implement logic to find a HandBook by ID
    },

    findByTitleLike: function(keyWord) {
        // Implement logic to find HandBooks by title regex
    }
};

module.exports = HandBookRepository;
```

Recommended Libraries/Frameworks in JavaScript:
- **Mongoose**: For MongoDB object modeling.
- **Express.js**: For building APIs and web servers.

Best Practices and Idioms in JavaScript:
- **Use Promises or Async/Await**: Handle asynchronous operations gracefully.
- **Use ES6 Features**: Arrow functions, destructuring, etc., for cleaner code.

Common Pitfalls to Avoid:
- **Not handling asynchronous operations properly**: Ensure proper error handling and callback chaining.
- **Not defining Mongoose schemas/models correctly**: Define schemas with required fields.

Additional Notes:
- **Consider using Express.js routes for API endpoints to interact with the repository.**
- **Ensure proper connection handling with the MongoDB database in Node.js.**

114. src/main/java/com/example/momcare/repository/MenuCategoryRepository.java

Migration Guide: Java to JavaScript

Code Summary:
The given Java code represents a repository interface for MenuCategory entities using Spring Data MongoDB.

Migration Plan:
1. **Create a JavaScript equivalent of the repository interface using a MongoDB driver or ORM library.**
2. **Update import statements to match JavaScript syntax.**
3. **Implement CRUD operations in the JavaScript repository.**

Before/After Code Snippets:
Java (Before):
```java
package com.example.momcare.repository;

import com.example.momcare.models.MenuCategory;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface MenuCategoryRepository extends MongoRepository<MenuCategory, String> {
}
```

JavaScript (After):
```javascript
const { MongoClient } = require('mongodb');

class MenuCategoryRepository {
  constructor() {
    this.client = new MongoClient('mongodb://localhost:27017');
    this.db = this.client.db('yourDBName');
    this.collection = this.db.collection('menuCategories');
  }
```

```
  async findAll() {
    return await this.collection.find({}).toArray();
  }

  async findById(id) {
    return await this.collection.findOne({ _id: id });
  }

  async save(menuCategory) {
    return await this.collection.insertOne(menuCategory);
  }

  async update(id, menuCategory) {
    return await this.collection.updateOne({ _id: id }, { $set: menuCategory });
  }

  async delete(id) {
    return await this.collection.deleteOne({ _id: id });
  }
}

module.exports = MenuCategoryRepository;
```

Recommended Libraries/Frameworks in JavaScript:
- **MongoDB Driver:** For direct interaction with MongoDB.
- **Mongoose:** An ORM library for MongoDB that simplifies data modeling.

Best Practices and Idioms in JavaScript:
- **Use Promises or async/await for asynchronous operations.**
- **Follow ES6 syntax for class definitions and module exports.**
- **Use arrow functions for concise code.**

Common Pitfalls to Avoid:
- **Not handling async operations correctly can lead to unexpected behavior.**
- **Ensure proper error handling for database interactions.**
- **Avoid mixing callback-based and promise-based code.**

Additional Advice:
- **For API usage, consider using Express.js for building RESTful APIs in JavaScript.**
- **Make sure to install necessary dependencies using npm or yarn.**

By following this migration guide, you can successfully convert the given Java repository interface to JavaScript for MongoDB operations.

115. src/main/java/com/example/momcare/repository/MenuRepository.java

Migration Guide from Java to JavaScript

Overview
The provided code is a Java interface `MenuRepository` that extends `MongoRepository` to interact with a MongoDB database. It defines methods for retrieving menu items based on category and ID.

Summary
- The `MenuRepository` interface provides methods to interact with a MongoDB database for menu-related operations.

Migration Plan
1. **Update package and imports**:
   - Update the package declaration to match the JavaScript project structure.
   - Replace Spring Data MongoDB imports with equivalent JavaScript libraries.

2. **Define interface methods**:
   - Define equivalent methods in JavaScript to interact with a MongoDB database using a Node.js driver.

3. **Handle asynchronous operations**:
   - Ensure asynchronous handling for database operations in JavaScript.

4. **Recommended Libraries**:
   - Use `mongoose` as an ORM for MongoDB in JavaScript.

5. **Best Practices and Idioms**:
   - Utilize Promises or async/await for handling asynchronous operations.
   - Follow Node.js module system for organizing code.

6. **Common Pitfalls**:
   - Be mindful of differences in method naming and parameter handling between Java and JavaScript.
   - Ensure proper error handling for database operations in JavaScript.

Migration Steps

1. Update Package and Imports
Before:
```java
package com.example.momcare.repository;

import com.example.momcare.models.Menu;
import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;
```

After (JavaScript):
```javascript
// No package declaration in JavaScript
const mongoose = require('mongoose');
const Menu = require('../models/Menu'); // Assuming Menu model is defined in a separate file
```

2. Define Interface Methods
Before:
```java
public interface MenuRepository extends MongoRepository<Menu, String> {
    List<Menu> findAllByCategory(String category);
    Menu getMenuById(String id);
}
```

After (JavaScript):
```javascript
const MenuSchema = new mongoose.Schema({
    // Define Menu schema fields
});

const MenuModel = mongoose.model('Menu', MenuSchema);

// Define equivalent methods
```

```
const MenuRepository = {
    findAllByCategory: async (category) => {
        return MenuModel.find({ category });
    },
    getMenuById: async (id) => {
        return MenuModel.findById(id);
    }
};
module.exports = MenuRepository;
```

3. Handle Asynchronous Operations
Ensure that all database operations are handled asynchronously using Promises or async/await.

4. Recommended Libraries
- **Mongoose**: An ORM for MongoDB that provides a straightforward schema-based solution for modeling application data.

5. Best Practices and Idioms
- Use `mongoose` schemas to define the structure of MongoDB documents.
- Utilize `async/await` for cleaner asynchronous code in JavaScript.

6. Common Pitfalls
- Watch out for differences in method naming and parameter handling between Java and JavaScript.
- Handle errors properly in asynchronous operations to avoid unhandled promise rejections.

Conclusion
By following this migration guide, you can successfully convert the provided Java code for a MongoDB repository into JavaScript using Node.js and `mongoose`. Remember to test the migrated code thoroughly to ensure its correctness and performance.

116. src/main/java/com/example/momcare/repository/MusicCategoryRepository.java

Migration Guide from Java to JavaScript

Summary:
The given Java code is a repository interface `MusicCategoryRepository` that extends `MongoRepository` for the `MusicCategory` model.

Migration Plan:
1. **Create a JavaScript equivalent interface for MongoDB:**
   - Use a JavaScript ORM library like Mongoose to interact with MongoDB.
   - Define a schema for `MusicCategory` model and create a Mongoose model.
   - Use the model to perform CRUD operations.

2. **Update Imports and Data Types:**
   - Replace Java specific imports with JavaScript equivalents.
   - Update data types if necessary.

3. **Implement CRUD Operations:**
   - Implement methods for CRUD operations like `findById`, `save`, `delete`, etc.
   - Use Mongoose queries to interact with the MongoDB database.

Before Migration (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.Music;
import com.example.momcare.models.MusicCategory;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface MusicCategoryRepository extends MongoRepository<MusicCategory, String> {
}
```

After Migration (JavaScript):
```javascript
const mongoose = require('mongoose');
```

```
const Schema = mongoose.Schema;

const musicCategorySchema = new Schema({
    // Define schema fields here
});

const MusicCategory = mongoose.model('MusicCategory', musicCategorySchema);

module.exports = MusicCategory;
```

Recommended Libraries/Frameworks in JavaScript:
- Mongoose: ORM library for MongoDB in Node.js.
- Express.js: Web framework for Node.js.
- Jest/Mocha: Testing frameworks for JavaScript.

Best Practices and Idioms in JavaScript:
- Use asynchronous operations for database interactions (promises, async/await).
- Follow modular architecture using ES6 modules.
- Use arrow functions for concise code.

Common Pitfalls to Avoid:
- Not handling asynchronous nature of JavaScript properly.
- Mixing callback-based and promise-based code.
- Neglecting error handling in asynchronous operations.

Migration Advice for Database/API:
- For MongoDB, use Mongoose library for schema definition and database interactions.
- Update API endpoints in Express.js routes to handle CRUD operations for `MusicCategory`.

By following the outlined migration plan and best practices, you can successfully convert the given Java repository interface to JavaScript using Mongoose for MongoDB interactions.

117. src/main/java/com/example/momcare/repository/MusicRepository.java

Java to JavaScript Migration Guide

MusicRepository

Summary:
The `MusicRepository` is a Java interface that extends `MongoRepository` for handling CRUD operations on `Music` entities. It also includes custom queries using `@Query` annotation to find music by category and all categories.

Migration Plan:
1. **Convert Interface to Class:** In JavaScript, interfaces are not supported in the same way as Java. Convert `MusicRepository` from an interface to a class.
2. **Use MongoDB Driver:** Instead of Spring Data MongoDB, use the MongoDB Node.js driver to interact with the MongoDB database.
3. **Rewrite Custom Queries:** Rewrite the custom queries using MongoDB aggregation or find methods in JavaScript.
4. **Handle Promises:** As JavaScript is asynchronous, make sure to handle promises or use async/await for database operations.

Before Migration (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.Category;
import com.example.momcare.models.Music;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface MusicRepository extends MongoRepository<Music, String> {
    @Query("{'category': ?0}")
    List<Music> findMusicByCategory(String category);
```

```
    @Query(value = "{}", fields = "{ 'category' : 1 }")
    List<Music> findAllCategories();
}
```

After Migration (JavaScript):
```javascript
const { MongoClient } = require('mongodb');

class MusicRepository {
    constructor() {
        this.client = new MongoClient('mongodb://localhost:27017', { useUnifiedTopology: true });
        this.database = null;
        this.collection = null;
    }

    async connect() {
        await this.client.connect();
        this.database = this.client.db('your-database-name');
        this.collection = this.database.collection('music');
    }

    async findMusicByCategory(category) {
        return await this.collection.find({ category: category }).toArray();
    }

    async findAllCategories() {
        return await this.collection.distinct('category');
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Node.js:** For server-side JavaScript runtime.
- **Express.js:** For building web applications and APIs.
- **MongoDB Node.js Driver:** For interacting with MongoDB.
- **Mongoose:** For ODM (Object Data Modeling) with MongoDB.

Best Practices and Idioms in JavaScript:
- **Use Promises:** Embrace asynchronous nature of JavaScript with promises or async/await.
- **ES6 Features:** Utilize modern JavaScript features like arrow functions, classes, and destructuring.
- **Error Handling:** Implement proper error handling using try/catch blocks or .catch() on promises.

Common Pitfalls to Avoid:
- **Callback Hell:** Avoid nested callbacks by chaining promises or using async/await.
- **Global Variables:** Limit the use of global variables, prefer encapsulation within classes or functions.
- **Not Closing Connections:** Make sure to close database connections after use to prevent memory leaks.

Database Migration Advice:
- **Connection String:** Update the MongoDB connection string to match your MongoDB instance.
- **Database Name:** Replace `'your-database-name'` with the actual name of your MongoDB database.
- **Collection Name:** Adjust the collection name from `'music'` to the actual collection name in your MongoDB database.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining functionality and best practices in the process.

118. src/main/java/com/example/momcare/repository/NotificationRepository.java

Migration Guide from Java to JavaScript

com.example.momcare.repository.NotificationRepository

Summary:
The `NotificationRepository` is a Java interface that extends `MongoRepository` to handle database operations related to notifications.

Migration Plan:
1. **Update Package Declaration:**
   - Change the package declaration to match the JavaScript module structure.

2. **Convert Interface Methods:**
   - Convert interface methods to JavaScript syntax using a suitable library like Mongoose for MongoDB operations.

3. **Handle Pageable and Page Objects:**
   - Implement pagination logic using libraries like `mongoose-paginate-v2` in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.Notification;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;

public interface NotificationRepository extends MongoRepository<Notification, String> {
    Page<Notification> findByReceiverIdAndIsRead(String receiverId, boolean isRead, Pageable pageable);
    Page<Notification> findByReceiverId(String receiverId, Pageable pageable);
    List<Notification> findAllByReceiverIdOrderByIdDesc(String receiverId);
```

```
    List<Notification> findAllByReceiverIdAndIsReadOrderByIdDesc(String receiverId, boolean isRead);
}
```

After Migration (JavaScript):
```javascript
import mongoose from 'mongoose';

const notificationSchema = new mongoose.Schema({
    // Define schema fields based on Notification model
});

const Notification = mongoose.model('Notification', notificationSchema);

const findByReceiverIdAndIsRead = async (receiverId, isRead, pageable) => {
    // Implement logic to find notifications by receiverId and isRead
};

const findByReceiverId = async (receiverId, pageable) => {
    // Implement logic to find notifications by receiverId
};

const findAllByReceiverIdOrderByIdDesc = async (receiverId) => {
    // Implement logic to find all notifications by receiverId ordered by Id descending
};

const findAllByReceiverIdAndIsReadOrderByIdDesc = async (receiverId, isRead) => {
    // Implement logic to find all notifications by receiverId and isRead ordered by Id descending
};

export { Notification, findByReceiverIdAndIsRead, findByReceiverId, findAllByReceiverIdOrderByIdDesc, findAllByReceiverIdAndIsReadOrderByIdDesc };
```

Recommended Libraries/Frameworks:
- Mongoose for MongoDB operations
- mongoose-paginate-v2 for pagination

Best Practices and Idioms in JavaScript:
- Utilize async/await for asynchronous operations.
- Use ES6 features like arrow functions and destructuring.

Common Pitfalls to Avoid:
- Be mindful of data types when migrating from Java to JavaScript.
- Ensure proper error handling in asynchronous operations.

Database Migration Advice:
- Use Mongoose to interact with MongoDB in JavaScript.

API Migration Advice:
- Implement APIs using frameworks like Express.js in JavaScript.

119. src/main/java/com/example/momcare/repository/PeriodicCheckRepository.java

Migration Guide: Java to JavaScript for PeriodicCheckRepository

Summary:
The given code is a Java interface `PeriodicCheckRepository` that extends `MongoRepository` and includes a method `findByWeekFrom` annotated with `@Query`.

Migration Plan:
1. **Update Package Declaration:**
   - Change the package declaration syntax.

2. **Convert Interface Extension:**
   - Replace `extends MongoRepository<PeriodicCheck, String>` with equivalent JavaScript code.

3. **Convert Query Annotation:**
   - Replace `@Query` annotation with appropriate JavaScript equivalent.

4. **Update Method Signature:**
   - Modify method signature for `findByWeekFrom` method.

5. **Handle Dependencies:**
   - Identify and include necessary libraries/frameworks in JavaScript.

Before/After Code Snippets:
Java Code:
```java
package com.example.momcare.repository;

import com.example.momcare.models.PeriodicCheck;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

public interface PeriodicCheckRepository extends MongoRepository<PeriodicCheck, String> {
    @Query("{weekFrom : ?0}")
    PeriodicCheck findByWeekFrom(int weekFrom);
```

}
```

### JavaScript Equivalent:
```javascript
// Import necessary libraries
// Define PeriodicCheckRepository class
class PeriodicCheckRepository {
    findByWeekFrom(weekFrom) {
        // Implementation logic
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **MongoDB Node.js Driver**
- **Mongoose (for Object Data Modeling with MongoDB)**

Best Practices and Idioms in JavaScript:
- **Use Promises or async/await for asynchronous operations.**
- **Follow ES6+ syntax for cleaner code.**
- **Use arrow functions for concise and readable code.**

Common Pitfalls to Avoid:
- **Avoid direct database connections in client-side JavaScript.**
- **Handle asynchronous operations properly to prevent callback hell.**

This migration guide provides a structured approach to converting the given Java code to equivalent JavaScript code for `PeriodicCheckRepository`. Make sure to test thoroughly after migration to ensure the functionality is preserved.
```

120. src/main/java/com/example/momcare/repository/SocialCommentRepository.java

Migration Guide: Java to JavaScript

SocialCommentRepository

Summary:
The code defines a repository interface for handling social comments in a MongoDB database using Spring Data.

Migration Plan:
1. **Update package declaration:** JavaScript doesn't have package declarations, so remove it.
2. **Modify import statements:** JavaScript doesn't have import statements in the same way as Java.
3. **Convert interface definition:** Translate the interface definition and methods to JavaScript syntax.
4. **Replace @Query annotations:** Use MongoDB query syntax directly in JavaScript methods.

Before/After Code Snippets:
```java
// Before
package com.example.momcare.repository;

import com.example.momcare.models.SocialComment;
import org.bson.types.ObjectId;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface SocialCommentRepository extends MongoRepository<SocialComment,String> {

    @Query("{'postId': ?0}")
    List<SocialComment> findSocialCommentByPostId(String postId);
    @Query("{'_id': ?0}")
    SocialComment findSocialCommentById(ObjectId id);
```

```
}
```

```javascript
// After
import { ObjectId } from 'bson';

class SocialCommentRepository {
    constructor() {
        // Initialization code
    }

    findSocialCommentByPostId(postId) {
        // MongoDB query to find social comments by postId
    }

    findSocialCommentById(id) {
        // MongoDB query to find social comment by id
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Express.js:** For building APIs and handling HTTP requests.
- **Mongoose:** MongoDB object modeling library for Node.js.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declarations.
- Embrace asynchronous programming with Promises or async/await.
- Follow ES6 syntax for cleaner and more readable code.

Common Pitfalls to Avoid:
- Mixing synchronous and asynchronous code.
- Not handling errors properly in asynchronous operations.
- Neglecting to close database connections when done.

Database Migration:
- **MongoDB:** MongoDB can be accessed in JavaScript using the `mongodb` or `mongoose` libraries.

- Update connection configurations and queries to match JavaScript syntax.
- Use Promises or async/await for handling asynchronous database operations.

API Migration:
- If the repository is used in an API, consider using Express.js to create API endpoints.
- Define routes that correspond to the repository methods for handling HTTP requests.
- Map request parameters to method arguments and return responses as needed.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining the functionality and structure of the original codebase.

121. src/main/java/com/example/momcare/repository/SocialPostRepository.java

Migration Guide: Java to JavaScript

Summary:
The given Java code is a repository interface for managing social posts in a mom care application. It includes methods for retrieving social posts by user ID, getting posts by ID, and finding posts by a keyword in the description.

Step-by-step Migration Plan:
1. **Model Classes:** Convert the `Diary` and `SocialPost` models to JavaScript classes.
2. **Repository Interface:** Translate the repository interface methods to JavaScript functions.
3. **Query Annotation:** Replace the `@Query` annotation with an equivalent approach in JavaScript.

Before/After Code Snippets:
Java:
```java
public interface SocialPostRepository extends MongoRepository<SocialPost, String> {
    List<SocialPost> getSocialPostsByUserId(String userId);

    Page<SocialPost> getSocialPostsByUserId(String userId, Pageable pageable);
    SocialPost getSocialPostById(String id);

    @Query("{'description': {$regex : ?0, $options: 'i'}}")
    List<SocialPost> findByDescriptionLike(String keyWord);
}
```

JavaScript:
```javascript
class SocialPostRepository {
    constructor() {
        // Constructor code for initializing repository
    }

    getSocialPostsByUserId(userId) {
```

```
        // Implementation for fetching social posts by user ID
    }

    getSocialPostsByUserId(userId, pageable) {
        // Implementation for fetching social posts by user ID with pagination
    }

    getSocialPostById(id) {
        // Implementation for fetching a specific social post by ID
    }

    findByDescriptionLike(keyWord) {
        // Implementation for finding social posts by description keyword
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Express.js:** For building RESTful APIs.
- **Mongoose:** For MongoDB object modeling.
- **Jest:** For testing JavaScript code.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declaration.
- Embrace asynchronous programming with Promises or async/await.
- Follow ES6+ syntax for cleaner and more readable code.

Common Pitfalls to Avoid:
- Be mindful of JavaScript's loose typing compared to Java.
- Handle asynchronous operations properly to avoid callback hell.
- Ensure proper error handling in asynchronous code.

Database Migration Advice:
- Consider using MongoDB with Mongoose for a similar database setup in JavaScript.
- Update database connection configurations according to the JavaScript environment.

API Migration Advice:
- Use Express.js to create RESTful API endpoints in JavaScript.

- Update API routes and request handling logic accordingly.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and best practices.

122. src/main/java/com/example/momcare/repository/TrackingRepository.java

Migration Guide: Java to JavaScript

TrackingRepository.js

Summary:
This code defines a repository interface for managing Tracking objects in a MongoDB database.

Migration Plan:
1. Create a new JavaScript file (TrackingRepository.js).
2. Define a class for the TrackingRepository.
3. Use a library like Mongoose to interact with MongoDB in JavaScript.
4. Implement the findTrackingByWeek method using Mongoose.

Before (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.Tracking;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

public interface TrackingRepository extends MongoRepository<Tracking,String> {
    @Query("{week : ?0}")
    Tracking findTrackingByWeek(int week);
}
```

After (JavaScript):
```javascript
const mongoose = require('mongoose');

const trackingSchema = new mongoose.Schema({
    week: Number,
    // Add other fields from Tracking model
});
```

```
const Tracking = mongoose.model('Tracking', trackingSchema);

class TrackingRepository {
    async findTrackingByWeek(week) {
        return Tracking.findOne({ week });
    }
}

module.exports = TrackingRepository;
```

Recommended Libraries/Frameworks:
- Mongoose for MongoDB interaction in Node.js

Best Practices and Idioms:
- Use async/await for asynchronous operations.
- Follow a consistent code style like Airbnb JavaScript Style Guide.

Common Pitfalls to Avoid:
- Not handling asynchronous operations properly.
- Forgetting to define the schema before creating the model in Mongoose.

Database Migration Advice:
- Use Mongoose to define schemas and models.
- Update connection configurations to use Mongoose in the JavaScript application.

API Migration Advice:
- Update API endpoints to handle JavaScript requests.
- Use Express.js to create API routes and controllers in JavaScript.

This migration guide provides a step-by-step plan for converting the given Java code to JavaScript, focusing on the TrackingRepository interface. It includes code snippets, recommended libraries, best practices, and common pitfalls to avoid during the migration process.

123. src/main/java/com/example/momcare/repository/UserRepository.java

Migration Guide from Java to JavaScript

Summary:
The given Java code is a repository interface for interacting with a MongoDB database. It defines methods to query and manipulate User objects stored in the database using Spring Data MongoDB annotations.

Migration Plan:
1. **Repository Interface:**
   - Create a JavaScript equivalent of the UserRepository interface.
   - Use a MongoDB client library in JavaScript to interact with the MongoDB database.

2. **Query Methods:**
   - Convert the @Query annotations to MongoDB queries in JavaScript.
   - Implement the query methods using the MongoDB client library.

3. **Additional Methods:**
   - Implement the existsByUserName method using the MongoDB client library.

4. **Testing:**
   - Test the migrated code to ensure it functions as expected.

Before/After Code Snippets:
Before (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.User;
import org.bson.types.ObjectId;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface UserRepository extends MongoRepository<User, String> {
```

```
    @Query("{userName : ?0}")
    User findUserByUserName(String userName);
    // Other query methods...
    boolean existsByUserName(String userName);
}
```

After (JavaScript):
```javascript
// JavaScript equivalent of UserRepository interface
class UserRepository {
    constructor(dbClient) {
        this.collection = dbClient.collection('users');
    }

    async findUserByUserName(userName) {
        return await this.collection.findOne({ userName: userName });
    }

    // Implement other query methods...

    async existsByUserName(userName) {
        const user = await this.collection.findOne({ userName: userName });
        return user !== null;
    }
}
```

Recommended Libraries or Frameworks:
- **MongoDB Client Library:** Use `mongodb` or `mongoose` library for interacting with MongoDB in JavaScript.

Best Practices and Idioms in JavaScript:
- **Asynchronous Operations:** Use async/await for asynchronous operations.
- **Error Handling:** Handle errors using try/catch blocks or promise rejections.

Common Pitfalls to Avoid:
- **Data Types:** Be aware of JavaScript's dynamic typing compared to Java's static typing.

- **Callback Hell:** Avoid nesting callbacks by using async/await or promises.

Database Migration Advice:
- **Connectivity:** Update connection configurations to match the JavaScript library requirements.
- **Query Syntax:** Update query syntax based on the JavaScript library's conventions.

By following this migration guide, you can successfully convert the given Java code to JavaScript for interacting with a MongoDB database.

124. src/main/java/com/example/momcare/repository/UserStoryRepository.java

Migration Guide: Java to JavaScript Migration for UserStoryRepository

Summary:
The given Java code defines a repository interface `UserStoryRepository` that extends `MongoRepository` to interact with a MongoDB database. It includes a method to find a `UserStory` by `userId`.

Migration Plan:
1. **Create a JavaScript equivalent for the repository interface `UserStoryRepository` using a suitable library like Mongoose for MongoDB interaction.
2. Define a method to find a `UserStory` by `userId`.
3. Update the import statements to reflect JavaScript modules.

Before/After Code Snippets:
Java Code:
```java
package com.example.momcare.repository;

import com.example.momcare.models.UserStory;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface UserStoryRepository extends MongoRepository<UserStory, String> {
    UserStory findByUserId(String userId);
}
```

JavaScript Equivalent:
```javascript
const mongoose = require('mongoose');

const UserStorySchema = new mongoose.Schema({
    // Define UserStory schema fields
});

const UserStoryModel = mongoose.model('UserStory', UserStorySchema);
```

```
const UserStoryRepository = {
    findByUserId: function(userId) {
        return UserStoryModel.findOne({ userId: userId });
    }
};

module.exports = UserStoryRepository;
```

Recommended Libraries/Frameworks in JavaScript:
- **Mongoose**: For interacting with MongoDB in JavaScript.
- **Express**: For building RESTful APIs.
- **Jest/Mocha**: For testing JavaScript code.

Best Practices and Idioms in JavaScript:
- Use asynchronous functions for database operations to avoid blocking the event loop.
- Follow the module pattern for encapsulation and code organization.
- Use ES6 features like arrow functions, destructuring, and promises for cleaner code.

Common Pitfalls to Avoid:
- Not handling asynchronous operations properly.
- Mixing callback-based and promise-based code.
- Neglecting error handling in database operations.

Database Migration Advice:
- **MongoDB to Mongoose**: Define schemas and models using Mongoose for MongoDB interaction.
- **MongoRepository to Mongoose Model**: Replace repository methods with Mongoose model methods for database operations.

By following this migration guide, you can successfully convert the given Java code to JavaScript using best practices and suitable libraries.

125. src/main/java/com/example/momcare/repository/VideoCategoryRepository.java

Migration Guide: Java to JavaScript

VideoCategoryRepository

Summary:
The `VideoCategoryRepository` is a Java interface that extends `MongoRepository` to handle CRUD operations for `VideoCategory` entities in a MongoDB database.

Migration Plan:
1. **Create a JavaScript class or module** to handle similar CRUD operations using MongoDB in Node.js.
2. **Define schema** for VideoCategory entities if not already done.
3. **Implement methods** for CRUD operations like `findById`, `findAll`, `save`, `delete`, etc.
4. **Use MongoDB Node.js driver** to interact with the MongoDB database.

Before/After Code Snippets:
Before (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.VideoCategory;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface VideoCategoryRepository extends MongoRepository<VideoCategory, String> {

}
```

After (JavaScript):
```javascript
// videoCategoryRepository.js
const { MongoClient } = require('mongodb');

class VideoCategoryRepository {
  constructor() {
```

```
    this.client = new MongoClient('mongodb://localhost:27017');
    this.db = this.client.db('your-db-name');
    this.collection = this.db.collection('videoCategories');
  }

  async findById(id) {
    return this.collection.findOne({ _id: id });
  }

  async findAll() {
    return this.collection.find().toArray();
  }

  async save(videoCategory) {
    return this.collection.insertOne(videoCategory);
  }

  async delete(id) {
    return this.collection.deleteOne({ _id: id });
  }
}

module.exports = VideoCategoryRepository;
```

Recommended Libraries/Frameworks in JavaScript:
- **MongoDB Node.js Driver**: For interacting with MongoDB in Node.js.
- **Express.js**: For building REST APIs if needed.

Best Practices and Idioms in JavaScript:
- **Use Promises or Async/Await**: For handling asynchronous operations.
- **ES6 Features**: Utilize ES6 syntax like arrow functions, classes, etc.
- **Modularization**: Organize code into modules for better maintainability.

Common Pitfalls to Avoid:
- **Ignoring Asynchronous Nature**: Ensure proper handling of asynchronous operations in Node.js.
- **Not Handling Errors**: Implement error handling for database operations.

Database Migration:
- **Migrate Data**: Use tools like `mongoexport` and `mongoimport` to migrate data from one MongoDB instance to another.
- **Update Connection String**: Update the connection string in the JavaScript code to point to the new MongoDB instance.

By following this migration guide, you can successfully convert the Java `VideoCategoryRepository` code to JavaScript for MongoDB operations.

126. src/main/java/com/example/momcare/repository/VideoRepository.java

Migration Guide: Java to JavaScript

Summary:
The code provided is a Java repository interface `VideoRepository` that extends `MongoRepository` for handling video data in a MongoDB database. It includes methods to find videos by category and to retrieve all categories.

Migration Plan:
1. **Convert to JavaScript**: Convert the Java code to JavaScript using appropriate libraries for MongoDB interactions.
2. **Use Promises or Async/Await**: JavaScript uses asynchronous operations, so ensure to handle database queries in a non-blocking way.
3. **Update Query Syntax**: MongoDB query syntax in JavaScript is different from Java, so update the queries accordingly.
4. **Replace Annotations**: Annotations like `@Query` are Java-specific, replace them with JavaScript equivalents.

Before Migration (Java):
```java
package com.example.momcare.repository;

import com.example.momcare.models.Category;
import com.example.momcare.models.Video;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.List;
import java.util.Set;

public interface VideoRepository extends MongoRepository<Video, String> {
    @Query("{'category': ?0}")
    List<Video> findVideosByCategory(String category);

    @Query(value = "{}", fields = "{ 'category' : 1 }")
    List<Video> findAllCategories();
```

}
```

## After Migration (JavaScript):
```javascript
// Assuming the usage of Mongoose library for MongoDB interactions

const mongoose = require('mongoose');

const VideoSchema = new mongoose.Schema({
    // Define Video schema properties
});

const VideoModel = mongoose.model('Video', VideoSchema);

const findVideosByCategory = async (category) => {
    return await VideoModel.find({ category: category });
};

const findAllCategories = async () => {
    return await VideoModel.find({}, { category: 1 });
};
```

Recommended Libraries/Frameworks in JavaScript:
- **Mongoose**: For MongoDB interactions in JavaScript.
- **Express.js**: For building REST APIs to interact with the database.

Best Practices and Idioms in JavaScript:
- **Use Promises or Async/Await**: Handle asynchronous operations effectively.
- **Use ES6 Features**: Take advantage of modern JavaScript features like arrow functions, destructuring, etc.

Common Pitfalls to Avoid:
- **Mismatch in Query Syntax**: Ensure queries are written in the correct MongoDB syntax.
- **Blocking Operations**: Avoid blocking operations that can affect performance.

Migration Advice for Database/API Usage:
```

- **Database Migration**: Update database connection configurations to work with JavaScript.
- **API Migration**: Update API endpoints and controllers to work with JavaScript frameworks like Express.js.

This guide provides a structured approach to migrating Java code to JavaScript, focusing on MongoDB interactions and best practices in JavaScript development.

127. src/main/java/com/example/momcare/security/CheckAccount.java

Migration Guide: Java to JavaScript

`CheckAccount` Class

Summary:
The `CheckAccount` class in Java is responsible for checking the validity of a user during signup by verifying their email, password strength, and username availability.

Migration Plan:
1. Convert the class to a JavaScript function.
2. Update method signatures and syntax for JavaScript.
3. Utilize JavaScript's string manipulation functions for password strength check.
4. Implement asynchronous calls if needed for database services.

Code Snippets:
Before Migration (Java):
```java
package com.example.momcare.security;

import com.example.momcare.models.User;
import com.example.momcare.service.UserService;
import org.springframework.stereotype.Component;

@Component
public class CheckAccount {
    public int checkSignup(User user, UserService service){
        // existing Java code
    }

    public boolean checkPassWordstrength(String passWord){
        // existing Java code
    }
}
```

After Migration (JavaScript):

```javascript
function checkSignup(user, service) {
    // migrated JavaScript code
}

function checkPasswordStrength(password) {
    // migrated JavaScript code
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building APIs
- bcrypt.js for password hashing
- Jest for testing

Best Practices in JavaScript:
- Use `const` and `let` for variable declarations.
- Utilize arrow functions for concise syntax.

Common Pitfalls to Avoid:
- Avoid blocking operations in JavaScript, use asynchronous calls instead.
- Remember that JavaScript is a loosely typed language, handle type conversions carefully.

Database and API Migration:
- Use Node.js with libraries like Sequelize or MongoDB for database operations.
- Utilize Express.js for creating RESTful APIs.

By following this migration guide, you should be able to successfully convert the given Java code to JavaScript while adhering to best practices and avoiding common pitfalls.

128. src/main/java/com/example/momcare/security/Encode.java

Java to JavaScript Code Migration Guide

com.example.momcare.security.Encode

Summary:
The given Java code is responsible for encoding a password using the MD5 hashing algorithm.

Migration Plan:
1. Translate the Java syntax to JavaScript.
2. Use the crypto module in Node.js for hashing operations.
3. Implement the password encoding function in JavaScript.

Before Migration (Java):
```java
package com.example.momcare.security;

import org.springframework.context.annotation.Bean;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

public class Encode {

    public String encoderPassword(String password) {
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("MD5");
            messageDigest.update(password.getBytes());
            byte[] resultByteArray = messageDigest.digest();
            StringBuilder sb = new StringBuilder();
            for (byte b : resultByteArray){
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
```

```
        }
    }
}
```

After Migration (JavaScript):
```javascript
const crypto = require('crypto');

class Encode {
    encoderPassword(password) {
        const md5Hash = crypto.createHash('md5');
        md5Hash.update(password);
        return md5Hash.digest('hex');
    }
}
module.exports = Encode;
```

Recommended Libraries/Frameworks in JavaScript:
- **Node.js**: for server-side JavaScript environment.
- **crypto module**: for cryptographic operations like hashing.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declarations.
- Use ES6 arrow functions for concise syntax.

Common Pitfalls to Avoid:
- Ensure that the hashing algorithm used in JavaScript is secure.
- Handle errors appropriately when dealing with cryptographic operations.

Database/API Migration Advice:
If this code interacts with a database or API, ensure that the corresponding database queries or API calls are also migrated to JavaScript using appropriate libraries like `mongoose` for MongoDB or `axios` for HTTP requests.

By following this migration guide, you can effectively convert the given Java code to JavaScript,

maintaining functionality and security in the process.

129. src/main/java/com/example/momcare/security/JwtAuthenticationFilter.java

Java to JavaScript Migration Guide

Summary:
The provided Java code represents a JwtAuthenticationFilter class that extends OncePerRequestFilter. It is responsible for filtering and processing JWT authentication in a Spring application.

Migration Plan:

1. **Remove Java Annotations**:
   - Remove `@Component` and `@RequiredArgsConstructor` annotations as they are not needed in JavaScript.

2. **Convert Class Structure**:
   - Convert the class structure to a JavaScript class.

3. **Modify Method Signatures**:
   - Modify method signatures to match JavaScript syntax.

4. **Handle HttpServletRequest and HttpServletResponse**:
   - Use equivalent methods from the Express.js framework to handle requests and responses.

5. **Implement JWT Handling Logic**:
   - Implement logic to extract JWT token from the request header and handle authentication.

Code Snippets:

Before:
```java
package com.example.momcare.security;

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Component;
```

```
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter{} // extends OncePerRequestFilter {
   // Java code snippet
}
```

After:
```javascript
// JavaScript equivalent
class JwtAuthenticationFilter {
   // JavaScript code snippet
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for handling HTTP requests and responses.
- jsonwebtoken for handling JWT tokens.

Best Practices and Idioms in JavaScript:
- Use ES6 class syntax for defining classes.
- Utilize Promises or async/await for asynchronous operations.

Common Pitfalls to Avoid:
- Ensure proper error handling for asynchronous operations.
- Be mindful of the differences in handling middleware in Express.js compared to Spring.

By following this migration guide, you can successfully convert the provided Java code to JavaScript while maintaining the functionality of JWT authentication in your application.

130. src/main/java/com/example/momcare/security/JwtService.java

Migration Guide: Java to JavaScript for JwtService

Summary:
The provided Java code is a JwtService class responsible for handling JWT (JSON Web Token) generation, validation, and extraction of claims.

Migration Plan:
1. **Set up the project environment**:
   - Ensure Node.js is installed.
   - Use a modern code editor like Visual Studio Code for JavaScript development.

2. **Convert dependencies**:
   - Replace Spring libraries with their JavaScript equivalents if needed.

3. **Convert JwtService class**:
   - Translate the class structure and methods to JavaScript.
   - Utilize JavaScript libraries for JWT handling.

4. **Test thoroughly**:
   - Test the migrated code to ensure functionality is preserved.

Code Migration Snippets:

Before:
```java
// Java code snippet
package com.example.momcare.security;

import org.springframework.stereotype.Service;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
```

```
@Service
public class JwtService {
    // Methods and logic here
}
```

After:
```javascript
// JavaScript code snippet
const jwt = require('jsonwebtoken');

class JwtService {
    // Methods and logic here
}
```

Recommended Libraries/Frameworks:
- **jsonwebtoken**: For JWT generation and verification.
- **bcryptjs**: For hashing and salting passwords securely.

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declaration.
- Embrace asynchronous programming with Promises or async/await.
- Follow ES6+ standards for cleaner and more readable code.

Common Pitfalls to Avoid:
- Incorrectly handling asynchronous operations.
- Not properly validating user input before processing.
- Forgetting to handle error cases in Promises.

Database/API Migration Advice:
- If interacting with a database, consider using an ORM like Sequelize for JavaScript.
- For API calls, use libraries like Axios to make HTTP requests.

By following this migration guide, you can successfully convert the Java JwtService class to JavaScript while maintaining its functionality and security features.

131. src/main/java/com/example/momcare/service/BabyHealthIndexService.java

Migration Guide: Java to JavaScript

Package: com.example.momcare.service

Summary:
This Java code represents a service class `BabyHealthIndexService` that handles operations related to baby health indices, such as creating, updating, deleting, and retrieving baby health data.

Migration Plan:
1. **Syntax Conversion**:
   - Convert Java syntax to JavaScript syntax.
2. **Library/Framework Replacement**:
   - Replace Spring annotations with Node.js libraries like Express.js.
3. **Architecture**:
   - Use async/await for asynchronous operations.
4. **API/Database**:
   - Replace database calls with Node.js database libraries like Sequelize or Mongoose.
5. **Best Practices**:
   - Use ES6 features like arrow functions, destructuring, and promises.
6. **Code Structure**:
   - Maintain modularity and separation of concerns.

Before/After Code Snippets:
Java (Before):
```java
package com.example.momcare.service;

import com.example.momcare.exception.ResourceNotFoundException;
import com.example.momcare.models.BabyHealthIndex;
...
```

JavaScript (After):
```javascript
```

```
// Import necessary libraries/modules
const ResourceNotFoundException = require('./exception');
const BabyHealthIndex = require('./models/BabyHealthIndex');
...
```

Recommended Libraries/Frameworks in JavaScript:
1. **Express.js**: For building RESTful APIs.
2. **Sequelize/Mongoose**: For interacting with databases.
3. **Jest/Mocha**: For testing.

Best Practices and Idioms in JavaScript:
- Use Promises or async/await for asynchronous operations.
- Use arrow functions for concise syntax.
- Follow ES6+ standards for cleaner code.

Common Pitfalls to Avoid During Migration:
- Ensuring proper error handling in asynchronous operations.
- Paying attention to data type conversions between Java and JavaScript.

Database/API Migration Advice:
- Use Sequelize or Mongoose for interacting with databases.
- Use Express.js for building APIs.

Conclusion:
By following this migration guide, you can effectively convert the Java code to JavaScript while maintaining functionality and best practices.

132. src/main/java/com/example/momcare/service/DiaryService.java

Migration Guide: Java to JavaScript

Summary
The given Java code is a service class for managing Diary objects. It includes methods for creating, updating, retrieving, and deleting Diary entries. The code interacts with a DiaryRepository for database operations.

Migration Plan
1. **Dependencies**: Replace Spring Framework dependencies with equivalent JavaScript libraries.
2. **Class Structure**: Convert class structure to JavaScript class syntax.
3. **Method Conversions**: Convert Java methods to JavaScript methods.
4. **Database Operations**: Replace Java repository methods with equivalent JavaScript database operations using frameworks like Sequelize or Mongoose.
5. **Error Handling**: Implement error handling in a JavaScript-friendly way.

Migration Steps
1. Dependencies
- **Java Libraries**: Spring Framework
- **Recommended JavaScript Libraries**: Express.js, Sequelize/Mongoose (for database operations)

2. Class Structure
Before:
```java
@Service
public class DiaryService {
    // Class implementation
}
```

After:
```javascript
class DiaryService {
    // Class implementation
}
```

```

### 3. Method Conversions
#### - `createDiary`
Before:
```java
@Transactional
public Diary createDiary(DiaryRequest diaryRequest) throws ResourceNotFoundException {
    // Method implementation
}
```

After:
```javascript
async createDiary(diaryRequest) {
    // Method implementation
}
```

- `findDiaryByIdUser`
Before:
```java
public List<Diary> findAllDiaryByUser(String idUser) {
    // Method implementation
}
```

After:
```javascript
async findAllDiaryByUser(idUser) {
    // Method implementation
}
```

4. Database Operations
- Replace `diaryRepository` operations with equivalent Sequelize or Mongoose operations for database interaction in JavaScript.
```

5. Error Handling
- Use `try-catch` blocks in JavaScript for error handling instead of throwing exceptions.

Best Practices and Pitfalls
- **Best Practices**:
  - Use `async/await` for asynchronous operations.
  - Utilize JavaScript array methods for data manipulation.
- **Pitfalls**:
  - Be cautious with data types and conversions between Java and JavaScript.
  - Ensure proper error handling to prevent unexpected behavior.

Conclusion
The migration from Java to JavaScript involves converting the class structure, methods, and database operations while adhering to JavaScript best practices and handling potential pitfalls. By following the outlined steps and recommendations, the code can be effectively migrated to JavaScript for continued development and maintenance.

133. src/main/java/com/example/momcare/service/EmailService.java

Migration Guide from Java to JavaScript

EmailService Class

Summary:
The EmailService class in Java is responsible for sending different types of email messages (verification, OTP, etc.) using the JavaMailSender and MimeMessage classes. It includes HTML templates for success and error messages.

Migration Plan:
1. Convert the Java code to JavaScript, focusing on the email sending functionality and HTML template rendering.
2. Use appropriate libraries in JavaScript to handle email sending and HTML template generation.
3. Update the CSS styling to be compatible with JavaScript frameworks.

Before Migration:
```java
// Java code
import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Service;

@Service
public class EmailService {
    // Methods and HTML templates
}
```

After Migration:
```javascript
// JavaScript code using Nodemailer for email sending and template literals for HTML templates
const nodemailer = require('nodemailer');
```

```
class EmailService {
    // Methods and HTML templates
}
```

Recommended Libraries/Frameworks in JavaScript:
- Nodemailer: For sending emails in JavaScript.
- Express.js: For handling HTTP requests if needed.

Best Practices and Idioms in JavaScript:
- Use asynchronous functions for email sending operations.
- Utilize template literals for HTML templates.
- Follow the Node.js style guide for coding conventions.

Common Pitfalls to Avoid:
- Ensure proper error handling in asynchronous functions.
- Pay attention to CORS issues if sending emails from a frontend application.

Additional Migration Advice:
- For handling HTTP requests in JavaScript, consider using Express.js with appropriate routing for email-related endpoints.
- Update the CSS styles in the HTML templates to be compatible with JavaScript frameworks like React or Angular.

By following this migration guide, you can successfully convert the Java EmailService class to JavaScript, maintaining functionality and best practices in the process.

134. src/main/java/com/example/momcare/service/HandBookCategoryService.java

Migration Guide: Java to JavaScript

HandBookCategoryService

Summary:
The `HandBookCategoryService` class in Java is a service responsible for interacting with `HandBookCategory` entities through a repository.

Migration Plan:
1. Create a new JavaScript class to represent `HandBookCategoryService`.
2. Implement methods to interact with the repository using JavaScript syntax.
3. Replace Java-specific annotations with JavaScript equivalents.

Before Migration (Java):
```java
package com.example.momcare.service;

import com.example.momcare.models.HandBookCategory;
import com.example.momcare.repository.HandBookCategoryRepository;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class HandBookCategoryService {
    HandBookCategoryRepository handBookCategoryRepository;

    public HandBookCategoryService(HandBookCategoryRepository
handBookCategoryRepository) {
        this.handBookCategoryRepository = handBookCategoryRepository;
    }

    public List<HandBookCategory> findAllCategories() {
        return this.findAll();
    }
```

```
    public List<HandBookCategory> findCategoriesByCollection(String collectionId) {
        return this.findCategoryByCollection(collectionId);
    }

    public List<HandBookCategory> findAll(){
        return this.handBookCategoryRepository.findAll();
    }

    public List<HandBookCategory> findCategoryByCollection(String collection){
        return this.handBookCategoryRepository.findByCollectionIn(collection);
    }
}
```

After Migration (JavaScript):
```javascript
class HandBookCategoryService {
    constructor(handBookCategoryRepository) {
        this.handBookCategoryRepository = handBookCategoryRepository;
    }

    findAllCategories() {
        return this.findAll();
    }

    findCategoriesByCollection(collectionId) {
        return this.findCategoryByCollection(collectionId);
    }

    findAll() {
        return this.handBookCategoryRepository.findAll();
    }

    findCategoryByCollection(collection) {
        return this.handBookCategoryRepository.findByCollectionIn(collection);
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building RESTful APIs
- Mongoose for MongoDB database interaction
- Jest for unit testing

Best Practices in JavaScript:
- Use ES6 classes for object-oriented design
- Follow asynchronous programming using Promises or async/await
- Use arrow functions for concise syntax

Common Pitfalls to Avoid:
- Ensure proper error handling in asynchronous operations
- Be mindful of data types when interacting with databases
- Watch out for differences in method naming conventions between Java and JavaScript

Database Migration Advice:
If the `HandBookCategoryRepository` interacts with a database, consider migrating to a JavaScript database library like Mongoose for MongoDB or Sequelize for SQL databases. Update the connection configuration and query methods accordingly.

API Migration Advice:
For API endpoints, consider using Express.js to create routes and handle HTTP requests in JavaScript. Update the controller methods to match the new syntax and conventions.

By following this migration guide, you can successfully convert the Java code for `HandBookCategoryService` to JavaScript while leveraging the best practices and libraries in the JavaScript ecosystem.

135. src/main/java/com/example/momcare/service/HandBookCollectionService.java

Migration Guide: Java to JavaScript

HandBookCollectionService.java

Summary:
The `HandBookCollectionService` class is a service in a Spring Boot application that interacts with a repository to fetch `HandBookCollection` data.

Migration Plan:
1. Create a JavaScript class that mimics the functionality of `HandBookCollectionService`.
2. Use JavaScript modules for encapsulation.
3. Utilize Promises for asynchronous operations.
4. Implement the equivalent of `findAllCollection` and `findAll` methods.

Before Migration:
```java
package com.example.momcare.service;

import com.example.momcare.models.HandBookCollection;
import com.example.momcare.repository.HandBookCollectionRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class HandBookCollectionService {
    HandBookCollectionRepository repository;

    public HandBookCollectionService(HandBookCollectionRepository repository) {
        this.repository = repository;
    }

    public List<HandBookCollection> findAllCollection() {
        return repository.findAll();
    }
```

```
    public List<HandBookCollection> findAll(){ return this.repository.findAll();}
}
```

After Migration:
```javascript
import HandBookCollectionRepository from './HandBookCollectionRepository.js';

class HandBookCollectionService {
    constructor() {
        this.repository = new HandBookCollectionRepository();
    }

    async findAllCollection() {
        return await this.repository.findAll();
    }

    async findAll() {
        return await this.repository.findAll();
    }
}

export default HandBookCollectionService;
```

Recommended Libraries/Frameworks:
- Express.js for building APIs.
- Axios for making HTTP requests.
- Jest for unit testing.

Best Practices in JavaScript:
- Use Promises or async/await for asynchronous operations.
- Utilize ES6 features like classes, modules, and arrow functions.
- Follow consistent coding style and naming conventions.

Common Pitfalls to Avoid:
- JavaScript's asynchronous nature may require adjustments in coding logic.

- Ensure proper error handling for Promise rejections.

HandBookCollectionRepository.java

Summary:
The `HandBookCollectionRepository` class is a repository in a Spring Boot application that interacts with the database to fetch `HandBookCollection` data.

Migration Plan:
1. Implement an equivalent data access layer in JavaScript.
2. Use a library like Sequelize or Mongoose for database interaction.
3. Map database queries to JavaScript functions.

Before Migration:
```java
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.momcare.models.HandBookCollection;

public interface HandBookCollectionRepository extends JpaRepository<HandBookCollection, Long> {
}
```

After Migration:
```javascript
import { Sequelize, DataTypes } from 'sequelize';

const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql'
});

const HandBookCollection = sequelize.define('HandBookCollection', {
  // Define model attributes
});

export default HandBookCollection;
```

Recommended Libraries/Frameworks:
- Sequelize or Mongoose for ORM in JavaScript.
- Express.js for creating RESTful APIs.

Best Practices in JavaScript:
- Use ORM libraries for database interactions.
- Separate concerns by creating models for database entities.

Common Pitfalls to Avoid:
- Ensure database connection configurations are set up correctly.
- Handle database query results properly to avoid unexpected behavior.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining the functionality and structure of the original application.

136. src/main/java/com/example/momcare/service/HandBookService.java

Java to JavaScript Migration Guide

Overview:
The code provided is a Java service class for handling operations related to a `HandBook` entity. It contains methods for finding `HandBook` entities by category, ID, time, key, and for retrieving top newest and random `HandBook` entries.

Migration Summary:
- **Purpose:** Convert Java code to JavaScript for `HandBookService` operations.
- **Key Features:**
  - Finding `HandBook` by category, ID, time, and key.
  - Retrieving top newest and random `HandBook` entries.
- **Libraries Used:** Spring Data, Random (Java).
- **Database:** Utilizes a `HandBookRepository` for database operations.

Migration Plan:

Initialization and Constructor:
1. **Summary:** Initialize the `HandBookService` class and set the `handBookRepository`.
2. **Migration Steps:**
   - Initialize `HandBookService` class in JavaScript.
   - Set `handBookRepository` in the constructor.

Before (Java):
```java
public class HandBookService {
    HandBookRepository handBookRepository;

    public HandBookService(HandBookRepository handBookRepository) {
        this.handBookRepository = handBookRepository;
    }
}
```

After (JavaScript):

```javascript
class HandBookService {
    constructor(handBookRepository) {
        this.handBookRepository = handBookRepository;
    }
}
```

Finding HandBook by Category:
1. **Summary:** Retrieve a list of `HandBook` entities based on a given category.
2. **Migration Steps:**
   - Implement `findHandBookByCategory` method in JavaScript.
   - Utilize the equivalent of `findAllByCategory` in JavaScript.

Finding HandBook by ID:
1. **Summary:** Retrieve a single `HandBook` entity based on its ID.
2. **Migration Steps:**
   - Implement `findHandBookByID` method in JavaScript.
   - Use the equivalent of `getHandBookById` in JavaScript.

Retrieving Top Newest HandBooks:
1. **Summary:** Get the top 8 newest `HandBook` entries.
2. **Migration Steps:**
   - Implement `top8Newest` method in JavaScript.
   - Sort and limit the results accordingly.

Retrieving HandBooks Per Time:
1. **Summary:** Retrieve `HandBook` entities based on a given time.
2. **Migration Steps:**
   - Implement `handBookPerTime` method in JavaScript.
   - Skip entries based on the given time.

Retrieving Top 8 Random HandBooks:
1. **Summary:** Get a random selection of 8 `HandBook` entries.
2. **Migration Steps:**
   - Implement `top8Random` method in JavaScript.
   - Generate random indexes to select entries.

Searching HandBook by Key:
1. **Summary:** Search for `HandBook` entries based on a keyword.
2. **Migration Steps:**
   - Implement `searchHandBook` method in JavaScript.
   - Use the equivalent of `findByTitleLike` in JavaScript.

Recommendations for JavaScript Migration:
- **Recommended Libraries/Frameworks:**
  - Express.js for backend routing.
  - Mongoose for MongoDB database operations.
- **Best Practices in JavaScript:**
  - Use async/await for asynchronous operations.
  - Utilize arrow functions for concise code.
- **Common Pitfalls to Avoid:**
  - Beware of differences in Java and JavaScript syntax.
  - Ensure proper error handling in asynchronous operations.

By following this migration guide, you can successfully convert the provided Java `HandBookService` code to JavaScript while maintaining its functionality and structure. Remember to test thoroughly to ensure the correctness of the migrated code.

137. src/main/java/com/example/momcare/service/MenuCategoryService.java

Migration Guide: Java to JavaScript

MenuCategoryService Class

Summary:
The `MenuCategoryService` class is a service class that interacts with the `MenuCategoryRepository` to retrieve menu categories.

Migration Plan:
1. Create a JavaScript class `MenuCategoryService` that will interact with the JavaScript equivalent of `MenuCategoryRepository`.
2. Update method implementations to use JavaScript syntax and conventions.
3. Utilize JavaScript libraries for dependency injection, if needed.

Before Migration (Java):
```java
package com.example.momcare.service;

import com.example.momcare.models.MenuCategory;
import com.example.momcare.repository.MenuCategoryRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class MenuCategoryService {
    MenuCategoryRepository repository;

    public MenuCategoryService(MenuCategoryRepository repository) {
        this.repository = repository;
    }

    public List<MenuCategory> findAllMenuCategory() {
        return findAll();
    }
```

```
    public List<MenuCategory> findAll() {
        return this.repository.findAll();
    }
}
```

After Migration (JavaScript):
```javascript
class MenuCategoryService {
    constructor(repository) {
        this.repository = repository;
    }

    findAllMenuCategory() {
        return this.findAll();
    }

    findAll() {
        return this.repository.findAll();
    }
}
```

Recommended Libraries/Frameworks in JavaScript:
- **Dependency Injection:** InversifyJS, Awilix
- **Unit Testing:** Jest, Mocha
- **Promises Handling:** Async/Await, Bluebird

Best Practices and Idioms in JavaScript:
- Use `class` syntax for defining classes.
- Use `constructor` for initializing class properties.
- Utilize ES6 features like arrow functions for concise code.

Common Pitfalls to Avoid:
- Beware of differences in handling asynchronous code between Java and JavaScript.
- Ensure proper handling of dependencies using libraries like InversifyJS.

Conclusion:
By following the migration plan and considering the recommended libraries and best practices, the conversion of the `MenuCategoryService` class from Java to JavaScript can be done effectively while avoiding common pitfalls.

138. src/main/java/com/example/momcare/service/MenuService.java

Java to JavaScript Migration Guide

com.example.momcare.service.MenuService

Summary:
This Java code defines a `MenuService` class that interacts with a `MenuRepository` to perform operations related to menus. It includes methods to find menus by category, find a menu by ID, and retrieve menus by category or ID.

Migration Plan:
1. **Replace Java syntax with JavaScript equivalents.**
2. **Update method implementations to suit JavaScript syntax.**
3. **Handle asynchronous operations if necessary.**

Before Migration:
```java
package com.example.momcare.service;

import com.example.momcare.exception.ResourceNotFoundException;
import com.example.momcare.models.Menu;
import com.example.momcare.repository.MenuRepository;
import com.example.momcare.util.Constant;
import org.springframework.stereotype.Service;

import java.util.List;
@Service
public class MenuService {
    MenuRepository repository;

    public MenuService(MenuRepository repository) {
        this.repository = repository;
    }

    public List<Menu> findMenusByCategory(String idCategory) {
        return findByCategory(idCategory);
```

```
    }
    public List<Menu> findMenuById(String id) throws ResourceNotFoundException {
        Menu menu = findById(id);
        if (menu != null) {
            return List.of(menu);
        } else {
            throw new ResourceNotFoundException(Constant.MENU_NOT_FOUND);
        }
    }
    public List<Menu> findByCategory (String category){return
this.repository.findAllByCategory(category);}
    public Menu findById (String id){return this.repository.getMenuById(id);}

}
```

After Migration:
```javascript
class MenuService {
    constructor(repository) {
        this.repository = repository;
    }

    async findMenusByCategory(idCategory) {
        return this.findByCategory(idCategory);
    }

    async findMenuById(id) {
        try {
            const menu = await this.findById(id);
            return [menu];
        } catch (error) {
            throw new Error(Constant.MENU_NOT_FOUND);
        }
    }

    findByCategory(category) {
        return this.repository.findAllByCategory(category);
```

```
    }

    findById(id) {
        return this.repository.getMenuById(id);
    }
}
```

Recommended Libraries/Frameworks:
- Express.js for building RESTful APIs
- Mongoose for MongoDB integration

Best Practices in JavaScript:
- Use `async/await` for asynchronous operations.
- Use ES6 syntax features like classes and arrow functions.
- Follow consistent naming conventions.

Common Pitfalls to Avoid:
- Ensure proper error handling for asynchronous operations.
- Be mindful of variable scoping in JavaScript.

Additional Migration Advice:
- Ensure proper setup of Node.js environment and dependencies.
- Consider using an ORM like Sequelize for database operations in JavaScript.

139. src/main/java/com/example/momcare/service/MomHealthIndexService.java

Java to JavaScript Migration Guide

MomHealthIndexService

Summary:
The `MomHealthIndexService` class in Java is responsible for managing mom health index data, including creating, updating, deleting, and retrieving mom health indices. It also includes methods for checking various health parameters and retrieving standard mom index values.

Migration Plan:
1. **Dependencies/Libraries**: You can use Node.js for JavaScript development along with libraries like Express.js for REST APIs and Sequelize for database operations.
2. **Step-by-Step Migration**:
   - Create a JavaScript service class `MomHealthIndexService`.
   - Convert methods like `createMomHealthIndex`, `updateMomHealthIndex`, `deleteMomHealthIndex`, `getMomHealthIndex`, `checkBMI`, `checkHealthRate`, `checkGlycemicIndex`, `getIndexStandard`, and `getStandardMomIndex`.
   - Replace Java-specific syntax with JavaScript equivalents.
3. **Before/After Code Snippets**:

Before (Java):
```java
package com.example.momcare.service;

...

@Service
public class MomHealthIndexService {
    ...
}
```

After (JavaScript):
```javascript
const MomHealthIndexService = require('./models/MomHealthIndex');
```

...
```

4. **Recommended Libraries**:
   - Express.js: For building RESTful APIs.
   - Sequelize: For handling database operations.
   - Moment.js: For date/time handling.

5. **Best Practices**:
   - Use Promises or async/await for handling asynchronous code.
   - Use arrow functions for cleaner syntax.
   - Follow ES6+ standards for modern JavaScript development.

6. **Common Pitfalls**:
   - Watch out for differences in syntax and data types between Java and JavaScript.
   - Be cautious with date/time handling in JavaScript.

### Conclusion:
By following this migration guide, you can successfully convert the Java `MomHealthIndexService` code to JavaScript, ensuring a seamless transition while adhering to best practices and avoiding common pitfalls.
```

140. src/main/java/com/example/momcare/service/MusicService.java

Migration Guide: Java to JavaScript

MusicService

Summary:
The `MusicService` class in Java is responsible for handling music-related operations by interacting with `MusicRepository` and `MusicCategoryRepository`.

Migration Plan:
1. Create a new JavaScript class `MusicService` with similar functionality.
2. Use JavaScript array methods for data manipulation.
3. Utilize ES6 syntax for cleaner code.
4. Use Promises for asynchronous operations.
5. Replace Java libraries with equivalent JavaScript libraries.

Before Migration (Java):
```java
package com.example.momcare.service;

import com.example.momcare.models.Music;
import com.example.momcare.models.MusicCategory;
import com.example.momcare.repository.MusicCategoryRepository;
import com.example.momcare.repository.MusicRepository;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

@Service
public class MusicService {
    // Constructor and methods
}
```

After Migration (JavaScript):

```javascript
// Import required modules
// Define MusicService class
// Implement methods using JavaScript syntax
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building RESTful APIs
- Mongoose for interacting with MongoDB (if applicable)
- Lodash for array manipulation

Best Practices in JavaScript:
- Use arrow functions for concise and readable code.
- Avoid using global variables.
- Use `const` and `let` for variable declarations.

Common Pitfalls to Avoid:
- Incorrectly handling asynchronous operations.
- Not using proper error handling techniques.
- Mixing synchronous and asynchronous code.

MusicRepository and MusicCategoryRepository

Summary:
These repositories are responsible for interacting with the database to fetch music and music categories.

Migration Plan:
1. Implement equivalent repository classes in JavaScript.
2. Use a JavaScript ORM like Sequelize for database operations.
3. Update method implementations to use Promises.

Before Migration (Java):
```java
// MusicRepository and MusicCategoryRepository classes
```

After Migration (JavaScript):

```javascript
// Implement equivalent repositories using JavaScript syntax
```

Recommended Libraries/Frameworks in JavaScript:
- Sequelize for ORM operations
- Axios for making HTTP requests
- Jest for testing

Best Practices in JavaScript:
- Use async/await for asynchronous operations.
- Follow naming conventions for consistency.
- Use parameter destructuring for cleaner code.

Common Pitfalls to Avoid:
- Not handling database connections properly.
- Incorrectly mapping Java classes to JavaScript objects.
- Failing to handle errors in database queries.

Conclusion:
By following this migration guide, you can successfully convert the Java code to JavaScript while adhering to best practices and avoiding common pitfalls. Remember to test thoroughly and ensure the functionality remains intact after migration.

141. src/main/java/com/example/momcare/service/NotificationService.java

An error occurred while generating the migration guide for this file.

142. src/main/java/com/example/momcare/service/PeriodicCheckService.java

Migration Guide: Java to JavaScript

Summary:
The provided Java code is a service class named `PeriodicCheckService` that interacts with a `PeriodicCheckRepository` to perform CRUD operations on `PeriodicCheck` objects. It includes methods to get all periodic checks, find by week, and find by specific week.

Migration Plan:
1. **Setup Environment:**
   - Install Node.js and npm for JavaScript development.
   - Choose a suitable code editor like Visual Studio Code.

2. **Convert Class Structure:**
   - Create a JavaScript class `PeriodicCheckService`.
   - Use ES6 class syntax to define the class and constructor.

3. **Handle Dependencies:**
   - Identify equivalent libraries or frameworks in JavaScript.
   - For data access, consider using libraries like Axios or Fetch for API calls.

4. **Convert Methods:**
   - Translate Java methods to JavaScript methods.
   - Handle asynchronous operations using Promises or async/await.

5. **Update Variable Declarations:**
   - Replace Java data types with JavaScript data types.
   - Update imports and exports as required.

6. **Testing and Debugging:**
   - Test the JavaScript code thoroughly to ensure it functions correctly.
   - Use debugging tools like Chrome DevTools for troubleshooting.

Code Migration:

1. Java Class to JavaScript Class:

```javascript
class PeriodicCheckService {
    constructor(periodicCheckRepository) {
        this.periodicCheckRepository = periodicCheckRepository;
    }

    getAll() {
        return this.findAll();
    }

    findByWeekFromService(weekFrom) {
        return this.findByWeekFrom(weekFrom);
    }

    findAll() {
        const sort = { direction: 'asc', property: Constant.WEEK_FROM };
        return this.periodicCheckRepository.findAll(sort);
    }

    findByWeekFrom(weekFrom) {
        return this.periodicCheckRepository.findByWeekFrom(weekFrom);
    }
}
```

2. Recommended Libraries/Frameworks in JavaScript:
- **Data Access:** Axios, Fetch API
- **Framework:** Express.js (for backend services)
- **Testing:** Jest, Mocha

3. Best Practices and Idioms:
- Use ES6 features like arrow functions, classes, and Promises.
- Follow asynchronous programming patterns to handle API calls efficiently.
- Use module system (CommonJS or ES modules) for code organization.

4. Common Pitfalls to Avoid:
- Beware of the differences in handling asynchronous operations between Java and JavaScript.
- Ensure proper error handling for API calls to prevent crashes.

Database/API Migration:
- If using REST APIs, rewrite API calls using Fetch or Axios in JavaScript.
- For database operations, consider using Node.js with libraries like Sequelize or Mongoose for MongoDB.

By following this migration guide, you can successfully convert the provided Java code to JavaScript while adhering to best practices and utilizing suitable libraries and frameworks.

143. src/main/java/com/example/momcare/service/SocialCommentService.java

Migration Guide: Java to JavaScript

Summary:
The code provided is a Java service class (SocialCommentService) responsible for handling social comments, reactions, and related operations in a social media application. It interacts with other services and repositories to perform CRUD operations on social comments.

Migration Plan:
1. **Dependencies/Libraries**: Use Node.js with Express.js for backend development.
2. **Syntax Conversion**: Convert Java syntax to JavaScript.
3. **API/Database Calls**: Utilize MongoDB/Mongoose for database operations.
4. **Code Structure**: Implement ES6 modules and classes for better code organization.
5. **Error Handling**: Use try-catch blocks and Promises for asynchronous operations.

Migration Steps:
1. **Setup Node.js Environment**:
   - Install Node.js and set up a new project.
   - Install required npm packages like Express, Mongoose, etc.

2. **Convert Java Syntax to JavaScript**:
   - Change package imports to JavaScript module imports.
   - Convert class definitions to ES6 class syntax.

3. **Database Migration**:
   - Replace MongoDB/Mongoose for database operations.
   - Update repository functions for MongoDB queries.

4. **Code Refactoring**:
   - Refactor methods using async/await for asynchronous operations.
   - Use arrow functions for callbacks and event handlers.

5. **Error Handling and Promises**:
   - Implement error handling using try-catch blocks.
   - Use Promises for handling asynchronous operations.

Code Snippets:
Before migration (Java):
```java
package com.example.momcare.service;
import com.example.momcare.exception.ResourceNotFoundException;
// Other imports...

@Service
public class SocialCommentService {
  // Class implementation...
}
```

After migration (JavaScript):
```javascript
import express from 'express';
import mongoose from 'mongoose';
// Other imports...

class SocialCommentService {
  // Class implementation...
}
```

Recommended Libraries/Frameworks in JavaScript:
1. **Express.js**: For building RESTful APIs.
2. **Mongoose**: MongoDB ORM for interacting with MongoDB.
3. **Jest/Mocha**: Testing frameworks for unit testing.

Best Practices in JavaScript:
- Use ES6 features like arrow functions, template literals, and destructuring.
- Follow async/await pattern for asynchronous operations.
- Modularize code using ES6 modules.
- Use linters like ESLint for code quality.

Common Pitfalls to Avoid:
- Mixing synchronous and asynchronous code.
- Not handling promises and errors properly.

- Overcomplicating code structure.

Migration Advice for Database/API Usage:
- Replace Spring Data JPA with Mongoose for MongoDB operations.
- Update repository functions to use Mongoose queries.

Additional Notes:
- Ensure proper testing of migrated code using Jest/Mocha.
- Follow JavaScript coding standards and conventions.

By following this migration guide, you can successfully convert the Java code to JavaScript for a Node.js backend application.

144. src/main/java/com/example/momcare/service/SocialPostService.java

Migration Guide: Java to JavaScript

SocialPostService

Summary:
The SocialPostService class is responsible for handling social post-related operations such as creating, updating, deleting posts, sharing posts, fetching posts by user, and more.

Migration Plan:
1. **Dependencies/Libraries:**
   - No direct equivalent for Spring Data JPA in JavaScript.
   - Use Node.js with Express.js for server-side operations.

2. **Code Migration Steps:**
   - Create a new JavaScript file for SocialPostService.
   - Translate class structure, method declarations, and member variables.
   - Replace Spring annotations with equivalent JavaScript approaches.
   - Use Node.js modules for managing dependencies.

3. **Before/After Code Snippets:**

Java (Before):
```java
package com.example.momcare.service;
...
@Service
public class SocialPostService {
    ...
}
```

JavaScript (After):
```javascript
const SocialPostRepository = require('./SocialPostRepository');
const UserService = require('./UserService');
```

```
class SocialPostService {
    constructor(socialPostRepository, userService) {
        this.socialPostRepository = socialPostRepository;
        this.userService = userService;
    }

    // Methods and implementation here
}

module.exports = SocialPostService;
```

4. **Recommended Libraries/Frameworks in JavaScript:**
    - Express.js for server-side routing.
    - Mongoose for MongoDB interactions.
    - Axios for making HTTP requests.

5. **Best Practices and Idioms in JavaScript:**
    - Use asynchronous functions for database operations.
    - Follow ES6 syntax for cleaner code.
    - Avoid callback hell by utilizing Promises or async/await.

6. **Common Pitfalls to Avoid:**
    - Be mindful of differences in asynchronous behavior compared to Java.
    - Handle error conditions gracefully in JavaScript.

Additional Notes:
- Consider using a NoSQL database like MongoDB for data storage.
- Implement authentication and error handling in Express.js endpoints.

This is a high-level guide to help you get started with migrating the Java code to JavaScript. Feel free to reach out for more detailed assistance or specific migration challenges.

145. src/main/java/com/example/momcare/service/TrackingService.java

Migration Guide: Java to JavaScript

TrackingService.java

Summary:
The `TrackingService` class in Java is a service that interacts with a `TrackingRepository` to retrieve tracking data and transform it into specific response formats.

Migration Plan:
1. Create a JavaScript class `TrackingService` to mirror the functionality of the Java class.
2. Use JavaScript syntax and data structures to refactor the methods for tracking data retrieval and response generation.
3. Utilize JavaScript modules and functions for improved code organization and readability.

Before Migration (Java):
```java
// Java code
```

After Migration (JavaScript):
```javascript
// JavaScript code
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for API routing
- Axios for making HTTP requests
- Mongoose for interacting with MongoDB (if needed)

Best Practices and Idioms in JavaScript:
- Use Promises or async/await for asynchronous operations.
- Use arrow functions to maintain the context of `this`.
- Follow ES6 syntax for cleaner code.

Common Pitfalls to Avoid:
- Be mindful of variable scoping differences between Java and JavaScript.

- Handle asynchronous operations properly to prevent callback hell.

Additional Considerations:
- If the `TrackingRepository` interacts with a database, consider using an ORM like Sequelize for Node.js.
- If an API is used for data retrieval, ensure proper error handling in JavaScript using try/catch blocks.

146. src/main/java/com/example/momcare/service/UserDetailCustom.java

Java to JavaScript Migration Guide

UserDetailCustom Service

Summary:
The `UserDetailCustom` service is responsible for loading user details from a UserRepository based on the provided username.

Migration Plan:
1. **Dependencies**:
   - Ensure similar libraries are available in JavaScript for security and user management (e.g., bcrypt for password hashing).
2. **Code Structure**:
   - Convert the class definition to JavaScript syntax.
   - Use ES6 classes for defining classes.
3. **UserRepository**:
   - If UserRepository is an API call, consider using fetch API in JavaScript to make similar requests.
4. **Error Handling**:
   - Convert `UsernameNotFoundException` handling to JavaScript error handling approach.
5. **Return Statement**:
   - Update the return statement to match JavaScript syntax for creating objects.

Before/After Code Snippets:
```java
// Before
public class UserDetailCustom implements UserDetailsService {
...
}

// After
class UserDetailCustom {
...
}
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building APIs.
- bcrypt.js for password hashing.
- passport.js for authentication.

Best Practices and Idioms in JavaScript:
- Use ES6 classes for defining classes.
- Utilize Promises/Async-Await for asynchronous operations.
- Follow a modular approach using modules and imports.

Common Pitfalls to Avoid:
- Directly translating Java code to JavaScript without considering asynchronous behavior.
- Ignoring error handling differences between Java and JavaScript.

Database/API Migration Advice:
- If UserRepository involves database queries, consider using Node.js with libraries like Sequelize or MongoDB for database interactions.
- For API calls, utilize fetch API in JavaScript or Axios library for making HTTP requests.

By following this migration guide, you can successfully convert the given Java code to JavaScript while maintaining functionality and best practices in the target language.

147. src/main/java/com/example/momcare/service/UserService.java

Migration Guide: Java to JavaScript

Package: com.example.momcare.service

Summary:
The code in this package contains a `UserService` class that handles various operations related to user accounts such as updating account details, changing passwords, handling OTPs, managing followers, and more.

Migration Plan:
1. **Dependencies and Setup**:
   - Ensure Node.js is installed for running JavaScript code.
   - Use npm to install necessary packages like MongoDB driver, bcrypt for password hashing, nodemailer for sending emails, etc.

2. **Code Migration**:
   - Convert the Java code to JavaScript by translating syntax and replacing Java-specific libraries with their JavaScript equivalents.
   - Handle asynchronous operations using promises or async/await.
   - Implement necessary error handling for exceptions and resource not found scenarios.

3. **Database Migration**:
   - If MongoDB is used, ensure the necessary database connection setup in JavaScript using MongoDB client libraries.
   - Adjust queries to work with JavaScript-based database interactions.

4. **API Migration**:
   - If the code interacts with external APIs, ensure API calls are made using JavaScript's fetch or axios library.

Before/After Code Snippets:
```javascript
// Before: Importing Java libraries
import jakarta.mail.MessagingException;
import org.bson.types.ObjectId;
import org.springframework.security.crypto.password.PasswordEncoder;
```

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

// After: Equivalent JavaScript imports
const nodemailer = require('nodemailer');
const bcrypt = require('bcrypt');
const { ObjectId } = require('mongodb');
```

Recommended Libraries/Frameworks in JavaScript:
- Express.js for building REST APIs.
- Mongoose for MongoDB object modeling.
- bcrypt for password hashing.
- Nodemailer for sending emails.
- Jest for testing.

Best Practices and Idioms in JavaScript:
- Follow asynchronous programming patterns using promises or async/await.
- Use arrow functions for concise and clear code.
- Utilize ES6 features like destructuring, template literals, and spread syntax.

Common Pitfalls to Avoid:
- Ensure proper error handling to avoid uncaught exceptions.
- Watch out for differences in date/time handling between Java and JavaScript.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining functionality and performance.

148. src/main/java/com/example/momcare/service/UserStoryService.java

Migration Guide: Java to JavaScript

Overview
The provided code is a service class in Java that handles user stories. It interacts with a repository for data access and performs CRUD operations on user stories. The code uses Spring annotations like `@Service` and `@Transactional` for defining service beans and transactional behavior.

Major Sections
1. **UserStoryService.java**
   - Defines a service class for managing user stories.
   - Contains methods for creating, deleting, and fetching user stories.

2. **UserStoryRepository.java**
   - Not included in the provided code snippet, assumed to be a repository interface for data access.

3. **User.java**
   - Not included in the provided code snippet, assumed to be a model class representing a user.

Migration Plan
UserStoryService.js
1. **New User Story Functionality**
   - Check if the user ID exists, create a new user story, and return a response.
   - **Migration Steps:**
     - Check for the existence of the user ID.
     - Create a new user story object.
     - Save the user story.
     - Fetch user details and construct a response.

2. **Delete User Story Functionality**
   - Delete a user story based on the user ID and return a response.
   - **Migration Steps:**
     - Find the user story by user ID.

    - Delete the user story.
    - Fetch user details and construct a response.

3. **Get All User Stories Functionality**
   - Retrieve all user stories, fetch user details for each story, and construct responses.
   - **Migration Steps:**
     - Fetch all user stories.
     - Iterate through each user story.
     - Fetch user details and construct responses.

Key Changes
- Replace Java syntax with JavaScript syntax.
- Translate Spring annotations to equivalent JavaScript functionality.
- Use JavaScript libraries for asynchronous operations and data manipulation.

Recommended Libraries/Frameworks
- **Express.js:** For building RESTful APIs in Node.js.
- **Mongoose:** For MongoDB object modeling.
- **Axios:** For making HTTP requests.
- **Jest:** For testing JavaScript code.

Best Practices in JavaScript
- Use `async/await` for asynchronous operations.
- Follow ES6 syntax for cleaner code.
- Utilize arrow functions for concise function definitions.
- Use Promises for handling asynchronous tasks.

Common Pitfalls to Avoid
- Ensure proper error handling for database operations.
- Validate input data to prevent security vulnerabilities.
- Handle asynchronous operations correctly to avoid callback hell.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining functionality and adhering to best practices in the JavaScript ecosystem.

149. src/main/java/com/example/momcare/service/VideoService.java

Java to JavaScript Migration Guide

com.example.momcare.service.VideoService

Summary:
The `VideoService` class in Java provides methods to retrieve videos based on categories and select random videos from a list.

Migration Plan:
1. **Rewrite the class in JavaScript using ES6 syntax.**
2. **Use asynchronous functions where necessary for better performance.**
3. **Replace Java specific libraries with their JavaScript equivalents.**

Before/After Code Snippets:
```java
// Before
package com.example.momcare.service;

import com.example.momcare.models.Video;
import com.example.momcare.models.VideoCategory;
import com.example.momcare.repository.VideoCategoryRepository;
import com.example.momcare.repository.VideoRepository;
import org.springframework.stereotype.Service;

import java.util.*;

@Service
public class VideoService {
    VideoRepository videoRepository;
    VideoCategoryRepository videoCategoryRepository;

    public VideoService(VideoRepository videoRepository, VideoCategoryRepository
videoCategoryRepository) {
        this.videoRepository = videoRepository;
        this.videoCategoryRepository = videoCategoryRepository;
    }
```

```
    public List<Video> top8Random(){
        // Method implementation
    }

    public List<Video> findByCategory(String category){
        // Method implementation
    }

    public List<VideoCategory> getAllCategories() {
        // Method implementation
    }
}
```

```javascript
// After
class VideoService {
    constructor(videoRepository, videoCategoryRepository) {
        this.videoRepository = videoRepository;
        this.videoCategoryRepository = videoCategoryRepository;
    }

    async top8Random() {
        // Method implementation
    }

    async findByCategory(category) {
        // Method implementation
    }

    async getAllCategories() {
        // Method implementation
    }
}
```

Recommended Libraries/Frameworks in JavaScript:

1. **Express.js** for building web APIs.
2. **Mongoose** for MongoDB database interactions.

Best Practices and Idioms in JavaScript:
- **Use async/await for asynchronous operations.**
- **Follow ES6 syntax for cleaner code.**
- **Use arrow functions for concise and clear code.**

Common Pitfalls to Avoid:
- **Avoid using Java-specific libraries like Spring in JavaScript.**
- **Handle asynchronous operations properly to prevent callback hell.**

Database/API Migration Advice:
- **Replace the Java repository classes with equivalent functions for interacting with the database in JavaScript.**
- **Use Express.js routes to handle API endpoints and integrate with the video service class.**

By following this migration guide, you can successfully convert the Java `VideoService` class to JavaScript while maintaining functionality and performance.

150. src/main/java/com/example/momcare/util/Constant.java

Java to JavaScript Migration Guide

Constant.java

Summary:
The `Constant` class in Java contains a set of constant strings used throughout the application for various purposes.

Migration Plan:
1. Create a JavaScript file to store the constant strings.
2. Define each constant string as a variable in the JavaScript file.
3. Export the variables for use in other JavaScript files.

Before/After Snippets:
```java
// Before (Java)
public static final String JWT_DECODE_FAILED = "JWT decode failed";

// After (JavaScript)
export const JWT_DECODE_FAILED = "JWT decode failed";
```

Recommended Libraries/Frameworks in JavaScript:
- No specific library/framework is required for this migration.

Best Practices and Idioms in JavaScript:
- Use `const` to declare constant variables.
- Export variables using `export` for better modularity.

Common Pitfalls to Avoid:
- Ensure that the constant values are not modified after declaration.

Overall, the migration process involves converting each constant string from Java to JavaScript by defining them as variables in a separate JavaScript file. Remember to maintain

the naming conventions and ensure that the constants are easily accessible throughout the application.

151. src/main/java/com/example/momcare/util/SecurityUtil.java

Java to JavaScript Migration Guide

Summary:
The Java code provided is a `SecurityUtil` class that handles the creation of access tokens using JWT (JSON Web Token) for authentication purposes.

Migration Plan:
1. **Set up project environment:** Ensure Node.js is installed, and choose a suitable JavaScript framework like Express.js for backend development.
2. **Convert dependencies:** Replace Spring Security dependencies with equivalent libraries in JavaScript.
3. **Rewrite the `SecurityUtil` class:** Convert the Java code to JavaScript, utilizing libraries like `jsonwebtoken` for JWT encoding.
4. **Handle configuration:** Manage configuration values using environment variables or configuration files in JavaScript.

Before/After Code Snippets:
Java (Before):
```java
// Java code snippet
// To be migrated to JavaScript
```

JavaScript (After):
```javascript
// JavaScript equivalent code
// After migration
```

Recommended Libraries/Frameworks in JavaScript:
- `jsonwebtoken` for JWT encoding and decoding.
- `express` for building RESTful APIs.
- `dotenv` for managing environment variables.

Best Practices and Idioms in JavaScript:
- Use asynchronous functions with `async` and `await` for handling promises.

- Follow modular design patterns like CommonJS or ES6 modules for code organization.
- Utilize arrow functions for concise syntax.

Common Pitfalls to Avoid:
- Avoid using synchronous functions that block the event loop.
- Be cautious of data type differences between Java and JavaScript (e.g., handling dates).
- Ensure proper error handling to prevent uncaught exceptions.

Database/API Migration Advice:
- If database operations are involved, consider using an ORM like Sequelize for JavaScript databases.
- For API endpoints, implement routing using Express.js and handle HTTP requests/responses accordingly.

By following this migration guide, you can successfully convert the provided Java code to JavaScript, maintaining the functionality and security of the `SecurityUtil` class.

152. src/main/java/com/example/momcare/util/Validator.java

Migration Guide: Java to JavaScript

Validator Class Migration

Summary:
The `Validator` class in Java provides a method `isValidHexString` to check if a given string is a valid hexadecimal string with exactly 24 characters.

Migration Plan:
1. Translate the regular expression pattern to JavaScript syntax.
2. Convert the method to a JavaScript function.
3. Update the method invocation to reflect JavaScript syntax.

Before Migration (Java):
```java
package com.example.momcare.util;

public class Validator {
    public static boolean isValidHexString(String str) {
        String regex = "^[0-9a-fA-F]{24}$";
        return str.matches(regex);
    }
}
```

After Migration (JavaScript):
```javascript
function isValidHexString(str) {
    const regex = /^[0-9a-fA-F]{24}$/;
    return regex.test(str);
}
```

Recommended Libraries/Frameworks in JavaScript:
- No additional libraries are required for this migration.

Best Practices and Idioms in JavaScript:
- Use `const` for variables that do not need to be reassigned.
- Use `regex.test()` method instead of `string.matches()`.

Common Pitfalls to Avoid:
- Ensure the regular expression syntax is correctly translated to JavaScript format.

Overall Migration Advice:
- Verify the functionality of the `isValidHexString` method after migration.
- Consider adding unit tests to validate the migration results.

153. src/main/resources/application.properties

Migration Guide from Java to JavaScript

Configuration File

Summary:
The Java code provided is a properties file containing configurations for MongoDB, server port, Eureka client, email server, JWT secret, and token validity settings.

Migration Plan:
1. Convert the properties file into a JavaScript configuration file.
2. Update the MongoDB URI format to the JavaScript equivalent.
3. Update the server port configuration.
4. Update the email server configuration.
5. Update the JWT secret and token validity configurations.
6. Remove any Java-specific configurations that are not needed in JavaScript.

Before Migration (Java Properties):
```properties
#spring.data.mongodb.database=momcare_db
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
spring.data.mongodb.uri= mongodb+srv://MeanIsme:
%40Enter0123@momcare.bfmrlyx.mongodb.net/test?retryWrites=true&w=majority
spring.data.mongodb.database=MomCare_db

server.port=${PORT:8080}
#eureka.client.register-with-eureka=false
#eureka.client.fetch-registry=false
spring.mail.host= smtp.gmail.com
spring.mail.port= 587
spring.mail.username= momcarenoreply@gmail.com
spring.mail.password= ctzwbytrmyvgglfr

backend.jwt.base64-secret= qoAEABDke07+AVLepXB4aCMtsT0wMAqR5x2VFyldsnx6e75YQk
JH2UcZKTjEyoNgG71SBCXfq5N6NVZxWOfsHQ==
backend.jwt.access-token-validity-in-seconds=86400
```

backend.jwt.refresh-token-validity-in-seconds=86400

spring.main.allow-bean-definition-overriding=true
```

### After Migration (JavaScript Configuration):
```javascript
module.exports = {
  mongodb: {
    uri: "mongodb://MeanIsme:%40Enter0123@momcare.bfmrlyx.mongodb.net/test?
retryWrites=true&w=majority",
    database: "MomCare_db"
  },
  server: {
    port: process.env.PORT || 8080
  },
  mail: {
    host: "smtp.gmail.com",
    port: 587,
    username: "momcarenoreply@gmail.com",
    password: "ctzwbytrmyvgglfr"
  },
  jwt: {
    base64Secret: "qoAEABDke07+AVLepXB4aCMtsT0wMAqR5x2VFyldsnx6e75YQkJH2UcZKT
jEyoNgG71SBCXfq5N6NVZxWOfsHQ==",
    accessTokenValidityInSeconds: 86400,
    refreshTokenValidityInSeconds: 86400
  }
};
```

Recommended Libraries/Frameworks in JavaScript:
- For MongoDB interactions: `mongoose`
- For JWT token handling: `jsonwebtoken`
- For server setup: `Express.js`

Best Practices and Idioms in JavaScript:
- Use `const` and `let` instead of `var` for variable declarations.
```

- Use ES6 arrow functions for concise code.
- Use destructuring assignment for object properties.
- Use `process.env` to access environment variables.

Common Pitfalls to Avoid:
- Ensure proper handling of sensitive information like passwords and secrets.
- Be mindful of differences in URI formats between MongoDB drivers in Java and JavaScript.

Conclusion:
By following this migration guide, you can effectively convert the provided Java properties file into a JavaScript configuration file while adhering to best practices and avoiding common pitfalls. Remember to test thoroughly after migration to ensure the functionality remains intact.

154. src/test/java/com/example/momcare/MomcareApplicationTests.java

Migration Guide: Java to JavaScript

MomcareApplicationTests.java

Summary:
The code is a test class for the Momcare application. It uses JUnit and Spring Boot annotations for testing.

Migration Plan:
1. Remove Java-specific annotations and imports.
2. Use a JavaScript testing framework like Jest for testing.
3. Update the test function to match Jest syntax.

Before Migration (Java):
```java
package com.example.momcare;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class MomcareApplicationTests {

” @est
—`oid contextLoads() {
—Đ

}
```

After Migration (JavaScript):
```javascript
// Example test using Jest
test('context loads', () => {
  // Test implementation
});
```

```

### Recommended Libraries/Frameworks in JavaScript:
- Jest for testing

### Best Practices and Idioms in JavaScript:
- Use Jest or other modern JavaScript testing frameworks for unit testing.
- Follow JavaScript naming conventions and coding style.

### Common Pitfalls to Avoid:
- Not understanding the differences in testing frameworks between Java and JavaScript.

## Database/API Migration:
If the code interacts with a database or API, the migration plan should include:
1. Use Node.js libraries like Mongoose for MongoDB or Sequelize for SQL databases.
2. For API calls, use fetch or Axios in JavaScript.

By following this migration guide, you can successfully convert the Java code to JavaScript while maintaining functionality and best practices.
```

155. system.properties

Migration Guide: Converting Java Properties File to JavaScript

Summary:
The given code snippet sets the Java runtime version to 17 in a properties file.

Migration Plan:
1. **Identify equivalent configuration in JavaScript:** JavaScript does not have a direct equivalent to setting runtime versions in a properties file.
2. **Remove the specific configuration:** Remove the line `java.runtime.version=17` as it is not applicable in JavaScript.

Before:
```java
java.runtime.version=17
```

After:
No equivalent configuration in JavaScript.

Recommended Libraries/Frameworks in JavaScript:
- No specific libraries or frameworks are needed for this specific configuration.

Best Practices and Idioms in JavaScript:
- JavaScript does not have a direct equivalent to setting runtime versions, so it is best to focus on relevant configurations or settings available in JavaScript.

Common Pitfalls to Avoid:
- Trying to directly replicate Java configurations in JavaScript may lead to confusion and unnecessary complexity. It's important to understand the differences in configuration between the two languages.

This is a basic configuration migration guide for converting a Java properties file to JavaScript.