**High-Level Design (HLD) of the Web Application with Backend Service:**

1. Architecture Overview:

- The web application will follow a client-server architecture.

- The client-side will consist of a web interface built using HTML, CSS, and JavaScript.

- The server-side will include a backend service built using a programming language (e.g., Python, Java, Node.js).

- The backend service will communicate with external services for authentication/authorization and data storage.

- The application will adhere to the OWASP Top 10 security standards to ensure security.

2. Authentication/Authorization:

- Utilize third-party authentication and authorization services like Google Sign-In, Okta, Auth0, or Firebase Authentication.

- Users will be able to sign in to the application using their credentials from these providers.

- Implement OAuth or OpenID Connect for secure authentication and authorization.

3. Web Interface:

- Develop a web interface using HTML, CSS, and JavaScript.

- Use a front-end framework like React, Angular, or Vue.js for a modular and interactive user interface.

- Implement web forms for creating, retrieving, updating, and deleting employee information.

- When retrieving data, load all information into the form and mark it as editable for easy modification.

4. Form Validation:

- Implement client-side form validation to ensure mandatory fields are filled and validate input based on specified length or format.

- Use JavaScript validation libraries like Yup, Joi, or validate.js to simplify validation logic.

- Provide real-time feedback to users for invalid inputs and prevent submitting incomplete or incorrect data.

5. Transactional Logging:

- Save all transactional logs to a file-based database for easy storage and retrieval.

- Consider using MongoDB for its flexibility and document-oriented nature.

- Log request and response payloads along with success or error codes.

- Implement proper log rotation and retention policies to manage log data effectively.

6. External Service Integration:

- Utilize RESTful APIs like https://crudcrud.com/ for CRUD operations on employee data.

- Integrate with a SOAP service like http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL for additional functionality if required.

- Implement client libraries or use HTTP client libraries (e.g., Axios, Retrofit) to communicate with these services.

7. IDE and SonarLint Integration:

- Integrate the chosen IDE with SonarLint for code quality and security analysis.

- Configure SonarLint to enforce best practices and detect potential security vulnerabilities or code smells.

- Regularly analyze the codebase for issues and address them promptly.

8. Cloud-Native Design and Deployment:

- Design the application to be cloud-native, making use of cloud services and technologies.

- Utilize containerization with Docker for easy deployment and scalability.

- Deploy the application on a cloud platform like Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure.

- Leverage cloud-native services like AWS Elastic Beanstalk, Google App Engine, or Azure App Service for application hosting.

9. Data Structure:

a. Based on your requirement to save all REST API calls to a file-based database, here's a sample data structure using MongoDB:

Database: **api\_logs**

Collection: **api\_calls**

Document Structure:

{

"\_id": "Unique identifier",

"timestamp": "Timestamp of the API call",

"request": {

"url": "URL of the API endpoint",

"method": "HTTP method (GET, POST, PUT, DELETE, etc.)",

"headers": {

"key1": "value1",

"key2": "value2",

...

},

"body": "Request payload (if applicable)"

},

"response": {

"statusCode": "HTTP status code",

"headers": {

"key1": "value1",

"key2": "value2",

...

},

"body": "Response payload (if applicable)"

}

}

Explanation:  
Each API call is represented by a document in the **api\_calls** collection.

* **\_id** is a unique identifier for each API call.
* **timestamp** stores the date and time when the API call was made.
* **request** field contains information about the request made to the API.
  + **url** stores the URL of the API endpoint.
  + **method** indicates the HTTP method used for the request.
  + **headers** hold any request headers sent.
  + **body** contains the request payload (if applicable).
* **response** field stores the response received from the API.
  + **statusCode** represents the HTTP status code returned by the API.
  + **headers** hold any response headers received.
  + **body** contains the response payload (if applicable).

This data structure allows you to store all the necessary details of each API call in a document within the **api\_calls** collection. You can query, filter, and analyze the API logs based on various criteria, such as timestamp, URL, status code, and more.

b. Based on your requirement to save all SOAP service calls to a file-based database, here's a sample data structure using MongoDB:

Database: **soap\_logs**

Collection: **soap\_calls**

Document Structure:

{

"\_id": "Unique identifier",

"timestamp": "Timestamp of the SOAP call",

"service": {

"wsdl": "WSDL URL of the SOAP service",

"method": "SOAP method called",

"headers": {

"key1": "value1",

"key2": "value2",

...

},

"body": "SOAP request body"

},

"response": {

"statusCode": "HTTP status code",

"headers": {

"key1": "value1",

"key2": "value2",

...

},

"body": "SOAP response body"

}

}

Explanation:

* Each SOAP service call is represented by a document in the **soap\_calls** collection.
* **\_id** is a unique identifier for each SOAP call.
* **timestamp** stores the date and time when the SOAP call was made.
* **service** field contains information about the SOAP service call.
  + **wsdl** stores the WSDL URL of the SOAP service.
  + **method** indicates the SOAP method being called.
  + **headers** hold any headers sent in the SOAP request.
  + **body** contains the SOAP request body.
* **response** field stores the response received from the SOAP service.
  + **statusCode** represents the HTTP status code returned by the SOAP service.
  + **headers** hold any headers received in the SOAP response.
  + **body** contains the SOAP response body.

This data structure allows you to store all the necessary details of each SOAP service call in a document within the **soap\_calls** collection. You can query, filter, and analyze the SOAP logs based on various criteria, such as timestamp, WSDL URL, status code, and more.

10. Sequence Diagram:

a. Here's a sequence diagram illustrating the process of utilizing a third-party authentication and authorization service like Google Sign-In, Okta, Auth0, or Firebase Authentication via a web application:

@startuml

actor User

participant WebApplication

participant ThirdPartyAuth

User -> WebApplication: Access web application

WebApplication -> ThirdPartyAuth: Redirect to authentication provider

ThirdPartyAuth --> WebApplication: Authentication page

User -> ThirdPartyAuth: Provide credentials

ThirdPartyAuth -> ThirdPartyAuth: Verify credentials

ThirdPartyAuth --> WebApplication: Authorization token

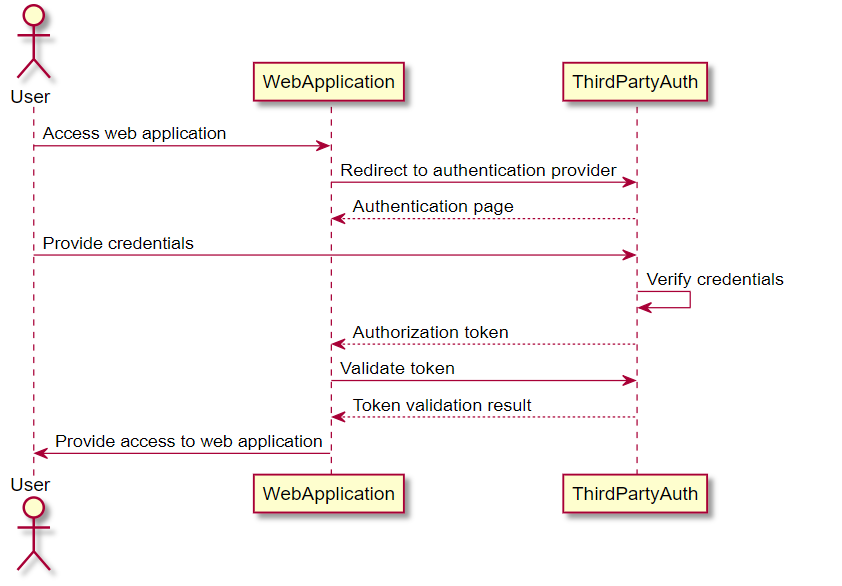
WebApplication -> ThirdPartyAuth: Validate token

ThirdPartyAuth --> WebApplication: Token validation result

WebApplication -> User: Provide access to web application

@enduml

<https://plantuml.atug.com/svg/ZP9B3i8m34JtaNe7Nu0Bi43q1WZnMPl4W4Le8iS5mUb9L25e2uYjupSp6iVCaw9eU_QLGQj1O9r8AXFpa2r7R1IsjAjZz6nHEJJzjzMHnItou5QtUgnCPGeFazc0ca9jBQK4Lze1lglrzmhPrvp2aXmBMGKDW7b2ZJwteuGBknAu3y7aKu81YmTwnHsQBZfbi4AkGEZJs6ICRKXuV_jDVOaMXEzTCWqd-gkP3NfsgFGa_d1QbKsuT62H5KgjrqzkfPfN4TZTBzzWTC8vDIx_8FC0>



Explanation:

1. The user accesses the web application.
2. The web application redirects the user to the chosen third-party authentication and authorization service.
3. The third-party authentication service presents an authentication page to the user.
4. The user provides their credentials on the authentication page.
5. The third-party authentication service verifies the user's credentials.
6. Upon successful verification, the third-party authentication service generates an authorization token.
7. The web application requests validation of the authorization token from the third-party authentication service.
8. The third-party authentication service validates the token.
9. The third-party authentication service sends the result of the token validation back to the web application.
10. If the token validation is successful, the web application provides the user with access to the web application.

b. Here's a sequence diagram illustrating the process of invoking a REST API from a web application and retrieving data in a web form, as well as posting data from a valid web form:

@startuml

actor User as "User"

participant WebApp as "Web Application"

participant RestApi as "REST API"

participant WebForm as "Web Form"

User -> WebApp: Interacts with the web application

WebApp -> RestApi: Sends REST API request

activate RestApi

RestApi --> WebApp: Returns REST API response

WebApp -> WebForm: Displays web form

activate WebForm

User -> WebForm: Fills out the form

WebForm --> WebApp: Sends form data

WebApp -> RestApi: Sends data to REST API

activate RestApi

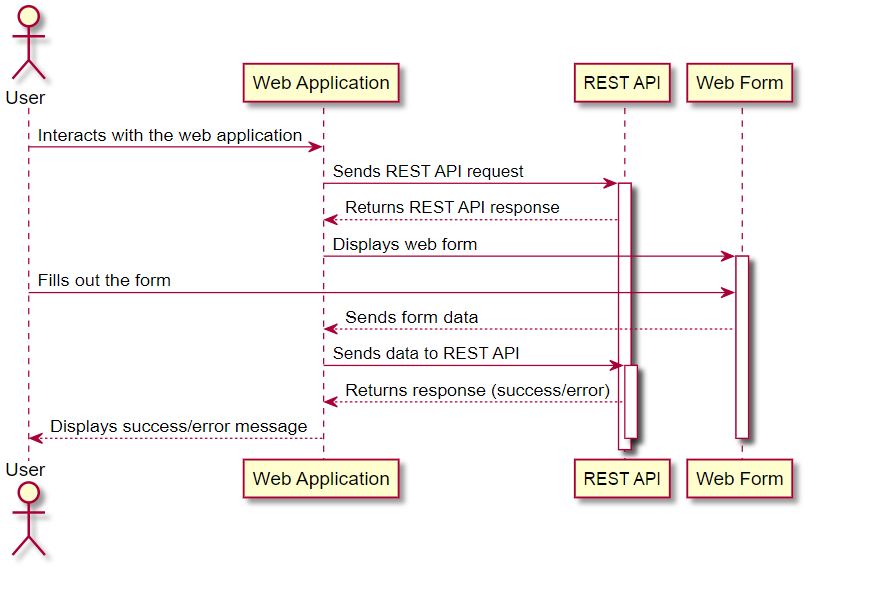
RestApi --> WebApp: Returns response (success/error)

WebApp --> User: Displays success/error message

deactivate RestApi

deactivate WebForm

@enduml  
  
<https://plantuml.atug.com/svg/bP9FQyCm3CNlXdm7qTDsABldC1hO2hsLT6DdBz5MGs9xbhAoRp_P-LEtXS5kilMattl6QaEi0mzzLoZTi0lmIXX04wnYiIgKbxPfZDUMuGtVA-zJLqgGkZEDPkFibR16uigRfAoV3ozGxNUtkxOkzCkoU119ePA1zUC4As5d6ODu8pWPFW8V4KwYrsTueIPZCZMHIpYWRGbcE0Jy6gILKfflpJWh8tAskywmDV8GxCK6yiuIvhGfH0bFXdodVoXP-v2h33I9hhADSrlJTGHku9Hh79nV9dSpfebzQ3NhFnB7Dh1RVFy_yHmKxcXe6YHwm11Sk3z3PIWcoM9VAA6NKdzY1BLuYy_kbhVPY7dvXUeN>



Explanation:

1. The User interacts with the web application.
2. The Web Application sends a REST API request to the REST API.
3. The REST API processes the request and returns a response to the Web Application.
4. The Web Application displays a web form to the User.
5. The User fills out the web form.
6. The Web Form sends the form data to the Web Application.
7. The Web Application sends the data to the REST API.
8. The REST API processes the data and returns a response (success or error) to the Web Application.
9. The Web Application displays the success or error message to the User.

This sequence diagram illustrates the flow of invoking a REST API from a web application, using a web form to retrieve or post data. It showcases the interaction between the User, Web Application, REST API, and Web Form.  
  
  
11. Wireframe  
To create a wireframe for a web application supporting CRUD operations on employee data using open-source wireframe tools, you can follow these guidelines and instructions:

1. Select an Open-Source Wireframe Tool:
   * There are several open-source wireframe tools available. Here are a few popular options:
     + Pencil Project: <https://pencil.evolus.vn/>
     + MockFlow: <https://www.mockflow.com/>
     + Wireframe.cc: <https://wireframe.cc/>
     + Figma (Free plan available): <https://www.figma.com/>
   * Choose the tool that best suits your requirements, considering features, ease of use, and compatibility with your operating system.
2. Define the Purpose and Scope:
   * Clearly define the purpose and scope of your web application supporting CRUD operations on employee data.
   * Identify the key functionalities and features that need to be included in the wireframe.
3. Start with the Main Layout:
   * Begin by creating the main layout of the web application.
   * Include a header section for the application title and navigation.
   * Allocate space for a sidebar or menu to provide access to different sections of the application.
4. Design the Employee List View:
   * Create a wireframe for the employee list view.
   * Include columns for displaying employee information such as ID, Name, Email, and actions (Edit, Delete).
   * Consider including pagination or infinite scrolling if there are a large number of employees.
5. Design the Employee Details View:
   * Create a wireframe for the employee details view.
   * Include form fields for entering employee information, such as Name, Email, Department, Position, etc.
   * Consider adding validation messages or indicators for mandatory fields or field formatting requirements.
6. Include CRUD Functionality:
   * Ensure that the wireframe supports CRUD operations (Create, Read, Update, Delete) on employee data.
   * Design appropriate buttons or icons for adding a new employee, editing employee details, and deleting an employee.
   * Consider incorporating confirmation prompts for delete operations to prevent accidental deletions.
7. Add Navigation and Flow:
   * Connect different wireframe screens together using navigation elements like buttons or links.
   * Create a flow between screens to represent user interactions and transitions.
8. Keep it Simple and Iterative:
   * Start with basic wireframes and gradually refine them as you gather feedback and iterate on the design.
   * Focus on the core functionalities and usability, avoiding excessive visual details at this stage.
9. Collaborate and Gather Feedback:
   * Share your wireframes with relevant stakeholders, such as team members or clients, to gather feedback and make improvements.
   * Encourage open communication and consider incorporating suggestions to enhance the wireframe design.
10. Export and Share:
    * Once you have finalized the wireframe, export it in a suitable format (e.g., PDF, image) to share it with others involved in the project.
    * You can also use collaborative features of the wireframing tool to allow real-time collaboration and feedback.

Remember, wireframing is an iterative process, and the wireframes serve as a visual representation of the application's structure and functionality. It's essential to focus on usability, simplicity, and clarity during the wireframing stage before proceeding to the actual application development.