Shared Validation Framework: UI (React) & API (Spring Boot) with Database-Driven Configuration

This document summarizes a comprehensive approach to implementing a shared validation framework across a React frontend and a Spring Boot backend. The key characteristic is that component-specific validation rules are stored in and fetched from a database, allowing for dynamic updates without application redeployment, while common validation patterns are defined in a shared definitions file.

Core Principles

1. **Single Source of Truth (for Rules Logic):** Validation rules are defined in a structured, language-agnostic format (JSON).
2. **Consistency:** The same rule definitions are used for both client-side (React) and server-side (Spring Boot) validation, preventing discrepancies.
3. **Reusability:** Common validation patterns (e.g., "nonEmptyString", "validEmail") are defined once and referenced by multiple specific configurations.
4. **Dynamic Configuration:** Component-specific validation configurations are fetched from a database, allowing for updates without code changes or deployments.
5. **Clear Separation of Concerns:**
    - JSON files define *what* to validate.
    - Code (TypeScript in React, Java in Spring Boot) defines *how* to interpret and apply these JSON rules.

Components of the Framework

1. **Common** Validation Definitions (validationDefinitions.json)
    - **Format:** JSON.
    - **Content:** A collection of reusable validation rule sets, each identified by a key (e.g., nonEmptyString, positiveAmount, accountNumber). Each definition specifies type, required status, patterns, min/max values, error messages, etc.
    - **Location (Backend):** Typically loaded from the classpath (e.g., src/main/resources/validation/common/validationDefinitions.json) by the Spring Boot service at startup.
    - **Location (Frontend):** Can be bundled with the React app or also fetched from a central configuration service/API if it needs to be dynamic. For simplicity in the current setup, it's often bundled or passed alongside the specific config.
2. **Component-Specific Validation Configurations (Stored in Database)**
    - **Format:** JSON strings.

- **Content:** Validation rules for a specific form (React component) or Data Transfer Object (Spring Boot DTO). These rules often use $ref pointers to the common definitions in validationDefinitions.json and can also include overriding or additional rules specific to that component/DTO.
- **Storage:** A database table (e.g., validation_configurations) where each row contains:
  - config_key (e.g., "WireTransferRequest", "LoanApplicationForm") - typically the DTO/component simple name.
  - config_json (the JSON string of the validation rules for that key).
- **Access (Backend):** The JsonValidationService in Spring Boot fetches these JSON strings from the database via a repository/service layer.
- **Access (Frontend):** The React app's validationConfigLoader.ts fetches the relevant JSON string for a form via an API endpoint exposed by the backend (or a dedicated configuration service).

3. **Frontend Implementation (React + TypeScript)**
   - **validationConfigLoader.ts:**
     - Responsible for asynchronously fetching the component-specific validation JSON (e.g., for WireTransferForm) from a backend API.
     - May also fetch or import the validationDefinitions.json.
     - Provides the fetched configSection (specific rules) and definitions to the form component.
   - **buildYupSchema.ts:**
     - A utility function that takes the configSection and definitions (as JSON objects/nodes).
     - Dynamically constructs a Yup validation schema by interpreting the JSON rules (type, required, minLength, pattern, custom rules, etc.).
     - Handles $ref resolution by looking up definitions.
   - **Form Component (e.g., WireTransferForm.tsx):**
     - Uses useEffect to call the validationConfigLoader to fetch its validation configuration when the component mounts.
     - Manages loading and error states for the configuration.
     - Once the configuration is loaded, it calls buildYupSchema to generate the Yup schema.
     - Uses react-hook-form with yupResolver, passing the dynamically generated schema for client-side validation.
     - Displays validation errors to the user.
   - **holidays.ts (or similar for custom logic):** Provides client-side implementations for custom validation rules (e.g., checking for weekends/holidays).

4. **Backend Implementation (Spring Boot + Java)**
    - **ValidationConfigRepository (Interface & Implementation):**
        - An interface defining methods to fetch validation configurations from the database.
        - An implementation (e.g., using Spring Data JPA or JdbcTemplate) that interacts with the database table storing the JSON configurations.
    - **JsonValidationService.java (Artifact ID: springboot_validation_service_generic_v1):**
        - Loads the common validationDefinitions.json from the classpath at startup.
        - Loads all component-specific validation configurations from the database (via ValidationConfigRepository) at startup and caches them (e.g., in a Map).
        - Provides a validate(Object data) method that:
            - Determines the configKey based on the simple class name of the input DTO (e.g., WireTransferRequest.class.getSimpleName()).
            - Retrieves the corresponding JsonNode (specific rules) from its cache.
            - Iterates through the fields defined in the JSON configuration.
            - For each field, resolves $ref pointers using the loaded common definitions.
            - Uses reflection to get the value of the field from the DTO.
            - Applies validation checks (required, type, pattern, min/max, custom rules like holiday checks) based on the JSON rules.
            - Collects any validation errors.
        - Throws a custom ValidationException containing a list of ErrorDetail objects if validation fails.
    - **DTOs (e.g., WireTransferRequest.java):** Plain Java objects representing the request payloads.
    - **Controller (e.g., WireTransferController.java):**
        - Receives the HTTP request and its payload (DTO).
        - Calls jsonValidationService.validate(dto).
        - Handles ValidationException to return a 400 Bad Request with structured error details.
        - Proceeds with business logic if validation passes.
    - **Holiday Logic:** Implemented within JsonValidationService for server-side date checks.

Workflow

1. **Configuration Setup:**

    - validationDefinitions.json is deployed with the Spring Boot application (in classpath).
    - Component-specific validation JSON (e.g., for WireTransferRequest) is stored as a JSON string in the database, keyed by "WireTransferRequest".
2. **Application Startup (Spring Boot):**
    - JsonValidationService loads validationDefinitions.json.
    - JsonValidationService fetches all specific configurations from the database and caches them.
3. **User Interaction (React Frontend):**
    - User navigates to a form (e.g., Wire Transfer).
    - The React form component (WireTransferForm.tsx) mounts.
    - useEffect triggers validationConfigLoader.ts to fetch the "WireTransferRequest" validation JSON from a backend API (which in turn reads from the database or its cache).
    - The loader also provides the common definitions.
    - buildYupSchema.ts uses this data to create a Yup schema.
    - The form is initialized with react-hook-form and this schema.
4. **Client-Side Validation (React):**
    - As the user types or attempts to submit, react-hook-form and Yup perform client-side validation based on the generated schema.
    - Immediate feedback is provided to the user.
5. **Form Submission (React to Spring Boot):**
    - If client-side validation passes (or if it's bypassed/incomplete), the form data is submitted as a JSON payload to the Spring Boot API endpoint.
6. **Server-Side Validation (Spring Boot):**
    - WireTransferController receives the WireTransferRequest DTO.
    - It calls jsonValidationService.validate(wireTransferRequestDto).
    - JsonValidationService uses "WireTransferRequest" (derived from the DTO class name) as the key to get the specific rules from its cache (originally from the DB).
    - It validates the DTO against these rules and the common definitions.
7. **API Response (Spring Boot to React):**
    - If server-side validation fails, the API returns a 400 Bad Request with a JSON body detailing the validation errors. The React app can then display these server-validated errors.
    - If validation passes, the API proceeds with business logic and returns a success response (e.g., 200 OK or 201 Created).

Benefits

- **Consistency:** Ensures identical validation logic on both client and server.
- **Maintainability:** Validation rules are centralized and managed (potentially in a DB).
- **Reduced Redundancy:** Define common rules once.
- **Dynamic Updates:** Specific validation rules can be changed in the database without needing to redeploy the frontend or backend (though a cache refresh mechanism might be needed for the backend if not just loading at startup).
- Clarity: JSON provides a readable format for validation rules.

This approach provides a robust and flexible framework for handling validations