# SE Project-2 Team-13

### Team Members and Contributions :

- Chirag Dhamija (2022101039) : Nested Categories and Filtering By RSS Feeds
- Tejas Cavale (2022101087) : Daily Report and Duplicate Detection
- Aditya Mishra (2022101047) : User-Created Feeds and Bug Reporting
- Jay Mupiddi (2021101035) : User Registration and Duplicate Detection
- Hardik Kalia (2022101092) : RSS Feeds and Bonus

# User Registration Feature

## Feature Details

Earlier, the RSS Reader application only allowed the **admin** to create user accounts, making it a closed and restricted system. To make the application more user-friendly and accessible, we have implemented a **user-registration feature** that enables users to register themselves using their email, username, and password.

## Design Approach

This feature was implemented by simply extending the existing admin-based registration logic. We removed the admin authentication check from the user creation endpoint, allowing it to serve both admin and user registrations. This avoided redundant code and made the design minimal and efficient.

Key benefits of this approach:

- **Reused existing user creation logic**, avoiding duplication.
- **Single unified endpoint** for both admin and user registrations.
- **Minimal UI changes** needed to support the feature.

Due to the simplicity of the requirement, **no design patterns were required**.

## Code Overview: Changes Made to Enable User Registration

### `UserDao.java` :

Modified the `create(User user)` method to validate both **username and email for uniqueness**. It throws appropriate exceptions ( `AlreadyExistingUsername` , `AlreadyExistingEmail` ) for duplicates.

### `UserResource.java` :

**Removed admin authentication checks** from the registration endpoint and added proper exception handling to differentiate between **duplicate username** and **duplicate email** cases. Here user's locale is also inferred from the request headers during registration.

### `index.html` :

We added a **"Go to Sign Up"** toggle button to switch between login and signup forms. Also we developed a new **signup form section** with fields such as Email, Username, Password and Confirm Password. We also changed `main.less` in order to do some changes in styling.

`r.main.js` :

Added a call to `r.user.signup()` during app initialization to enable signup form handling.

`r.user.js` :

Added a new method `r.user.signup()` which handles client-side form validation, submits a **PUT request** to `/user/register`, and redirects to the login page, and displays appropriate error messages if the username or email already exists.

## Process Flow

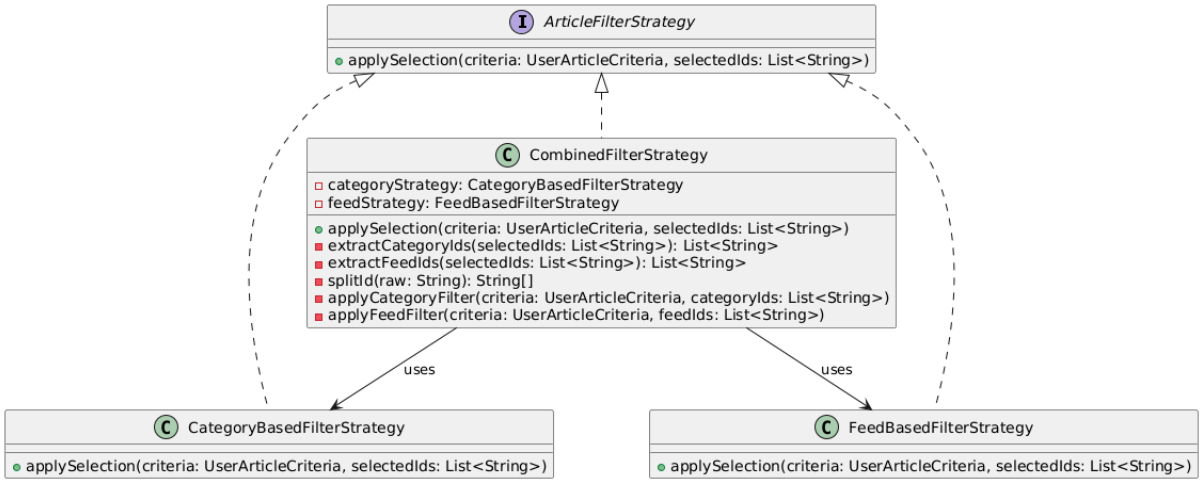
- User lands on the login page.
- User clicks **“Go to Sign Up”** and the UI toggles to display the signup form.
- User enters: all the relevant fields.
- Form inputs are validated on the **client side** (e.g., email format, password match, minimum length).
- If the form is valid a **PUT request** is sent to the backend at `/user/register` with the user’s details.
- On the backend, the system checks for **duplicate username and email**; if valid, the user is created and stored in the database, else appropriate exceptions like `AlreadyExistingUsername` or `AlreadyExistingEmail` are returned.
- On successful registration, a **POST request** is sent to the login page with auto-fill, from where the user can login.

# Filtering By RSS Feeds :

## Feature Details

Earlier, the RSS Reader application only supported basic feed filtering where users could view either all articles or articles from a single feed. There was no provision to filter articles by multiple categories, multiple feeds, or a combination of both. To address this, we enabled dynamic filtering based on selected feeds, selected categories, or both together. In the UI, we added separate buttons—Filter Feed and Filter Categories—which display respective lists for user selection. Upon selecting a combination and clicking the submit button, the corresponding filtered results are shown. This feature has been implemented using a combination of Strategy and Composite design patterns.

## Design Pattern



### Strategy Pattern :

The **Strategy Pattern** allows different filtering behaviors (based on feed or category) to be encapsulated in separate strategy classes. Each class adheres to a common interface and implements specific filtering logic.

- `ArticleFilterStrategy` – Common interface defining a common method `applySelection()` for all strategies.
- `FeedBasedFilterStrategy` – Applies filtering based on feed IDs.
- `CategoryBasedFilterStrategy` – Applies filtering based on category IDs.

This pattern provides a **flexible way to plug in different filtering strategies** without modifying the core logic of article retrieval.

Below are the benefits after applying this pattern.

- **Modular Design:** Each filtering strategy (such as `CategoryBasedFilterStrategy` or `FeedBasedFilterStrategy`) is encapsulated in its own class, promoting cleaner architecture and better code reusability.
- **Enhanced Readability:** Keeping filtering logic in separate, well-defined strategy classes makes the codebase easier to navigate and maintain.
- **Seamless Extensibility:** New filtering mechanisms like tag-based or author-based filtering can be introduced easily by adding new strategy classes, without altering existing logic.
- **Flexible Filtering:** The system can dynamically switch between category-based or feed-based filtering by simply applying the appropriate strategy, without modifying core processing logic.

### Composite Pattern :

The **Composite Pattern** is used to support **combined filtering** — i.e., when users select both feed and category checkboxes in the UI.

- `CombinedFilterStrategy` combines both `FeedBasedFilterStrategy` and `CategoryBasedFilterStrategy`.
- It parses the list of selected IDs, segregates them into feed and category IDs, and delegates the filtering to the respective strategies.

This pattern provides a **clean orchestration layer**, enabling **multi-type filtering without coupling the controller logic to individual strategies**.

Below are the benefits after applying this pattern :

- **Encapsulation of Filtering Logic:** The `CombinedFilterStrategy` consolidates the logic for handling multiple types of filters, improving code organization and avoiding duplication.
- **Support for Multi-level Filtering:** It enables users to filter articles by categories, feeds, or both at the same time, allowing for a highly personalized experience.
- **Intelligent Delegation:** The system automatically categorizes selected IDs (as category or feed) and routes them to the relevant strategy, ensuring clean separation of concerns.
- **Scalability:** Additional filtering types can be integrated into the system in the future with minimal changes, thanks to the composite structure that naturally supports expansion.

## Code Overview :

### ArticleFilterStrategy (Interface) :

Defines a **standard contract** for all filtering strategies.

```
public interface ArticleFilterStrategy {
    void applySelection(UserArticleCriteria criteria, List<String> selectedIds);
```

### FeedBasedFilterStrategy (Class) :

Implements filtering logic **specific to feed IDs**.

```
public class FeedBasedFilterStrategy implements ArticleFilterStrategy {
    @Override
    public void applySelection(UserArticleCriteria criteria, List<String> selectedIds) {

        criteria.setFeedIds(selectedIds);
    }
}
```

### CategoryBasedFilterStrategy (Class) :

Implements filtering logic **specific to category IDs**.

```
public class CategoryBasedFilterStrategy implements ArticleFilterStrategy {
    @Override
    public void applySelection(UserArticleCriteria criteria, List<String> selectedIds) {

        criteria.setCategoryIds(selectedIds);
    }
}
```

### CombinedFilterStrategy (Class) :

Acts as a **Composite Strategy**, combining both category and feed filtering in a single place.

```
public class CombinedFilterStrategy implements ArticleFilterStrategy {

    private final ArticleFilterStrategy categoryStrategy = new CategoryBasedFilterStrategy();
    private final ArticleFilterStrategy feedStrategy = new FeedBasedFilterStrategy();

    @Override
    public void applySelection(UserArticleCriteria criteria, List<String> selectedIds) {
        List<String> categoryIds = extractCategoryIds(selectedIds);
        List<String> feedIds = extractFeedIds(selectedIds);

        applyCategoryFilter(criteria, categoryIds);
        applyFeedFilter(criteria, feedIds);
    }


    private List<String> extractCategoryIds(List<String> selectedIds) {
```

```
        List<String> categoryIds = new ArrayList<>();
        for (String id : selectedIds) {
            String[] parts = splitId(id);
            if (parts != null && "categoryId".equals(parts[0])) {
                categoryIds.add(parts[1]);
            }
        }
        return categoryIds;
    }


    private List<String> extractFeedIds(List<String> selectedIds) {
        List<String> feedIds = new ArrayList<>();
        for (String id : selectedIds) {
            String[] parts = splitId(id);
            if (parts != null && "subscriptionId".equals(parts[0])) {
                feedIds.add(parts[1]);
            }
        }
        return feedIds;
    }


    private String[] splitId(String raw) {
        if (raw == null || !raw.contains(":")) return null;
        String[] parts = raw.split(":", 2);
        if (parts.length < 2) return null;
        return new String[] { parts[0].trim(), parts[1].trim() };
    }

    /**
     * Applies category-based filtering if applicable.
     */
    private void applyCategoryFilter(UserArticleCriteria criteria, List<String> categoryIds) {
        if (!categoryIds.isEmpty()) {
            categoryStrategy.applySelection(criteria, categoryIds);
        }
    }


    private void applyFeedFilter(UserArticleCriteria criteria, List<String> feedIds) {
        if (!feedIds.isEmpty()) {
            feedStrategy.applySelection(criteria, feedIds);
        }
    }
}
```

## Process Flow :

- User selects feeds and categories in the UI and the selection is submitted.

- `ArticleFilterResource` controller receives the list of selected IDs
- Controller instantiates the `CombinedFilterStrategy`
- `CombinedFilterStrategy` processes the selected IDs and delegates:
  - `CategoryBasedFilterStrategy` applies category-based filtering
  - `FeedBasedFilterStrategy` applies feed-based filtering
- Filtered articles are retrieved and returned to the frontend

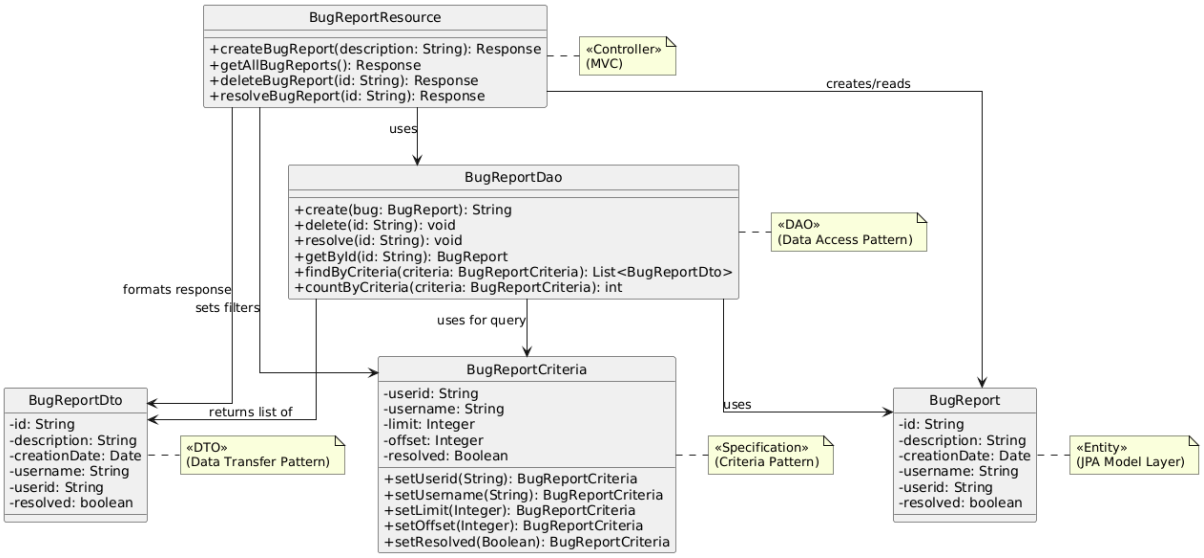
# Bug Reporting Feature

## Feature Details

Previously, the RSS Reader application redirected users to GitHub to report issues. While this was convenient for developers, it wasn't an ideal experience for end-users. To improve usability and streamline feedback collection, a **native in-app bug reporting system** has been implemented from scratch. Now, users can submit bug reports directly within the application. Each report consists of a short **description** , along with the reporting user's identity. Admin users get access to a **bug dashboard** where they can view, resolve, or delete reports. Additionally, users can delete the bugs they submitted.

This system has been designed using a modular architecture based on well-known design patterns such as **MVC (Layered Architecture)**, **DAO**, **DTO**, and **Specification (Criteria)** Pattern.

## Design Pattern




### 1. MVC / Layered Architecture

The code tries to reflect aspects of MVC through the use of `BugReportResource.java` (Controller) and `BugReport.java` (Model), while the presentation responsibilities are handled externally by the frontend. This organization helps structure bug reporting operations like create, fetch, delete, and resolve, making each part independently modifiable and easier to manage.

### 2. DAO (Data Access Object) Pattern

The DAO pattern encapsulates all direct database access logic in one place (`BugReportDao`), isolating it from the business and presentation layers. This centralization simplifies maintenance and promotes code reuse when handling persistence operations such as creating, deleting, and resolving bug reports.

### 3. DTO (Data Transfer Object) Pattern

DTOs (`BugReportDto`) are used to transfer only the necessary bug report data between the DAO and Resource layers, avoiding exposure of the full entity structure. This keeps internal representations hidden while minimizing data transfer and simplifying API responses sent to the frontend.

### 4. Specification (Criteria) Pattern

The criteria class (`BugReportCriteria`) encapsulates dynamic filtering logic (such as filtering by user or resolution status) without cluttering DAO methods. This enables flexible, reusable, and modular query construction without changing the core data access code.

## Code Overview

### `BugReportResource.java` (Presentation Layer – Controller) :

This class handles REST endpoints for creating a bug (`POST /bugreport`), viewing all bugs (`GET /bugreport/list`), deleting a bug (`DELETE /bugreport/{id}/delete`), and marking a bug as resolved (`PUT /bugreport/{id}/resolve`).

```
BugReport bugReport = new BugReport();
bugReport.setDescription(description);
bugReport.setUsername(principal.getName());
bugReport.setUserid(principal.getId());
bugReport.setCreationDate(new Date());
bugReport.setResolved(false);
bugReportDao.create(bugReport);
```

### `BugReportDao.java` (Persistence Layer – DAO) :

This class encapsulates all interaction with the database, including methods of `create()`, `delete()`, `resolve()` for bug reports, and and also provides filtering functionalities using `findByCriteria()` and `countByCriteria()`.

```
public String create(BugReport bugReport) {
    bugReport.setId(UUID.randomUUID().toString());
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    em.persist(bugReport);
    return bugReport.getId();
}
```

### `BugReportDto.java` (DTO Layer) :

This class is used to transfer simplified bug report data between the DAO and Resource layers, containing essential attributes like `id`, `description`, `creationDate`, `username`, `userid`, and `resolved`.

```
public class BugReportDto {
    private String id;
    private String description;
    private Date creationDate;
    private String username;
    private String userid;
```

```
    private boolean resolved;
}
```

### `BugReportCriteria.java` (Specification Pattern) :

Encapsulates filtering logic for dynamic queries such as filtering only unresolved bugs, filtering by user ID or username, and supporting pagination through `limit` and `offset`.

```
public BugReportCriteria setResolved(Boolean resolved) {
    this.resolved = resolved;
    return this;
}
```

### `BugReport.java` (Model Layer – Entity) :

This entity class represents the database structure for the `T_BUG_REPORT` table and maps fields such as `BUR_ID_C`, `BUR_IDUSER_C`, `BUR_DESCRIPTION_C`, `BUR_CREATEDATE_D`, and `BUR_RESOLVED_B`.

```
@Entity
@Table(name = "T_BUG_REPORT")
public class BugReport {
    @Id
    @Column(name = "BUR_ID_C")
    private String id;

    @Column(name = "BUR_DESCRIPTION_C")
    private String description;

    @Column(name = "BUR_CREATEDATE_D")
    private Date creationDate;

    @Column(name = "BUR_RESOLVED_B")
    private boolean resolved;
}
```

### SQL Schema Changes:

Defines the database structure for storing bug reports, including fields for description, creation timestamp, user details, and a foreign key relationship with the user table.

```
create cached table T_BUG_REPORT (
  BUR_ID_C varchar(36) not null,
  BUR_IDUSER_C varchar(36) not null,
  BUR_NAMEUSER_C varchar(50) not null,
  BUR_DESCRIPTION_C longvarchar not null,
  BUR_CREATEDATE_D datetime not null,
  primary key (BUR_ID_C)
);

alter table T_BUG_REPORT
  add constraint FK_BUR_IDUSER_C foreign key (BUR_IDUSER_C)
```

```
  references T_USER (USE_ID_C)
  on delete restrict on update restrict;
```

## Process Flow

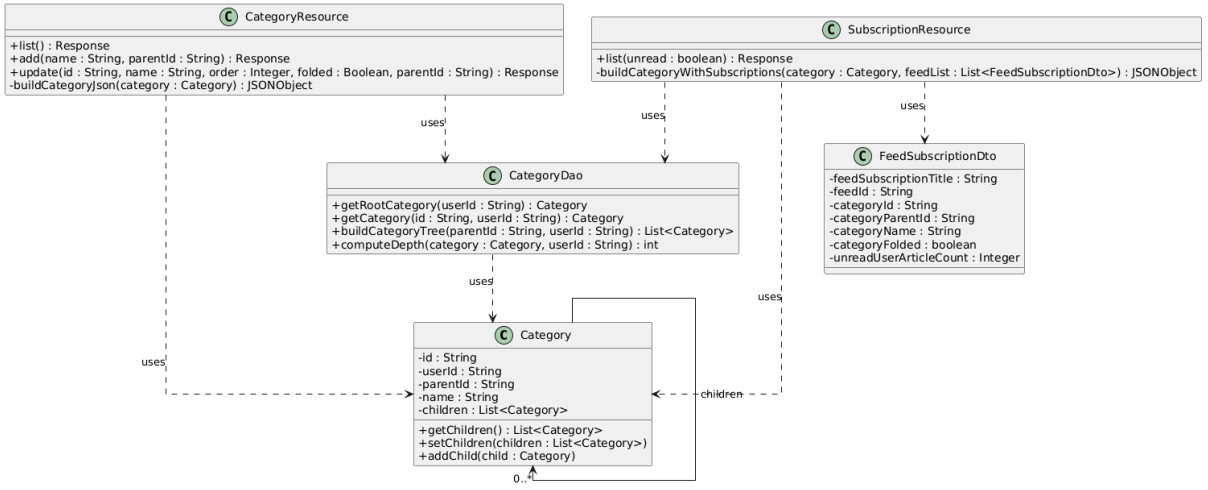
- **User submits bug report** from frontend.
- **BugReportResource** receives the request and performs validation.
- A **BugReport entity object** is created and populated with user data.
- **BugReportDao.create()** persists the data into the database.
- On dashboard load, **BugReportResource.getAllBugReports()** is called.
- It uses **BugReportDao.findByCriteria()** with appropriate filters based on the user/admin.
- DAO returns a list of **BugReportDto**, which is then formatted into JSON and sent to frontend.
- Admin actions like **resolve or delete** are handled via `PUT` or `DELETE` endpoints respectively.

# Nested Categories Feature

## Feature Details

Previously, categories in the RSS Reader were strictly single-level, limiting users to basic feed groupings. We now support **nested categories** up to five levels deep, each capable of holding both feeds and child categories, while displaying **unread** and **total article** counts (including all subcategories). A **drag-and-drop** interface also lets users easily move categories and feeds to reorganize their hierarchy. This collapsible, hierarchical structure significantly improves organization, offering an intuitive and efficient way to navigate and manage large numbers of feeds.

## Design Pattern




### Composite Pattern

We employed the **Composite Pattern** to handle categories that can contain both subcategories (composites) and feeds (leaf-like structures). In this pattern:

- **Component**: A generic “Category” object, capable of referencing children.

- **Composite**: A `Category` that has one or more child categories.
- **Leaf**: A `Category` with no children.

The Composite pattern is ideal here becuase of the benefits it offers which are mentioned below.

### Benefits of the Composite Pattern

1. **Uniform Treatment**
   - We can manage top-level categories, subcategories, and nested subcategories using the same operations (add, update, delete).
2. **Recursive Nesting**
   - The same code logic builds or traverses subcategories at *any* level of nesting (up to 5 levels).
3. **Simplified UI Rendering**
   - The client receives a nested JSON tree. Collapsing/expanding is straightforward because each `Category` node is structured the same way.
4. **Scalability**
   - Any future enhancements like adding icons or metadata can be seamlessly supported at every node.

## Code Overview

Below are the main classes that implement the nested categories feature.

### 1. `Category.java` (Entity) :

Added a `children` field (with getters and setters) to store subcategories. A category with an empty `children` list behaves like a leaf, while one with multiple children is a composite.

```
@Transient
private List<Category> children = new ArrayList<>();

public List<Category> getChildren() {
    return children;
}

public void setChildren(List<Category> children) {
    this.children = children;
}

public void addChild(Category child) {
    this.children.add(child);
}
```

### 2. `CategoryDao.java` (Data Access) :

- Implemented `buildCategoryTree(parentId, userId)` to recursively fetch subcategories for a given parent category. Also added `computeDepth(...)` function to enforce a maximum nesting limit. Each time we go up one parernt, we increment depth and if depth hits 5, we block further nesting.

```
public List<Category> buildCategoryTree(String parentId, String userId) {
    List<Category> roots = findSubCategory(parentId, userId);
```

```
        for (Category category : roots) {
            List<Category> children = buildCategoryTree(category.getId(), userId);
            category.setChildren(children);
        }
        return roots;
    }
    public int computeDepth(Category category, String userId) {
        int depth = 0;
        String parentId = category.getParentId();
        while (parentId != null) {
            Category parent = getCategory(parentId, userId);
            if (parent == null) break;
            depth++;
            parentId = parent.getParentId();
            if (depth > 5) break;
        }
        return depth;
    }
```

### 3. `CategoryResource.java` (REST) :

- Modified the `list()` endpoint to build the entire nested tree (using `buildCategoryTree`) and return it as JSON. We have also updated the `add()` and `update()` endpoints to check `computeDepth(...)` and prevent exceeding 5 levels.

### 4. `SubscriptionResource.java` (REST) :

- Similar recursion is used in `SubscriptionResource` to attach feed subscription data under each category in the hierarchy.
- This way, the UI receives a single JSON tree containing both categories (with possible subcategories) **and** the feeds inside them. Also added a helper function `buildCateogryWithSubscriptions(...)`

## Process Flow

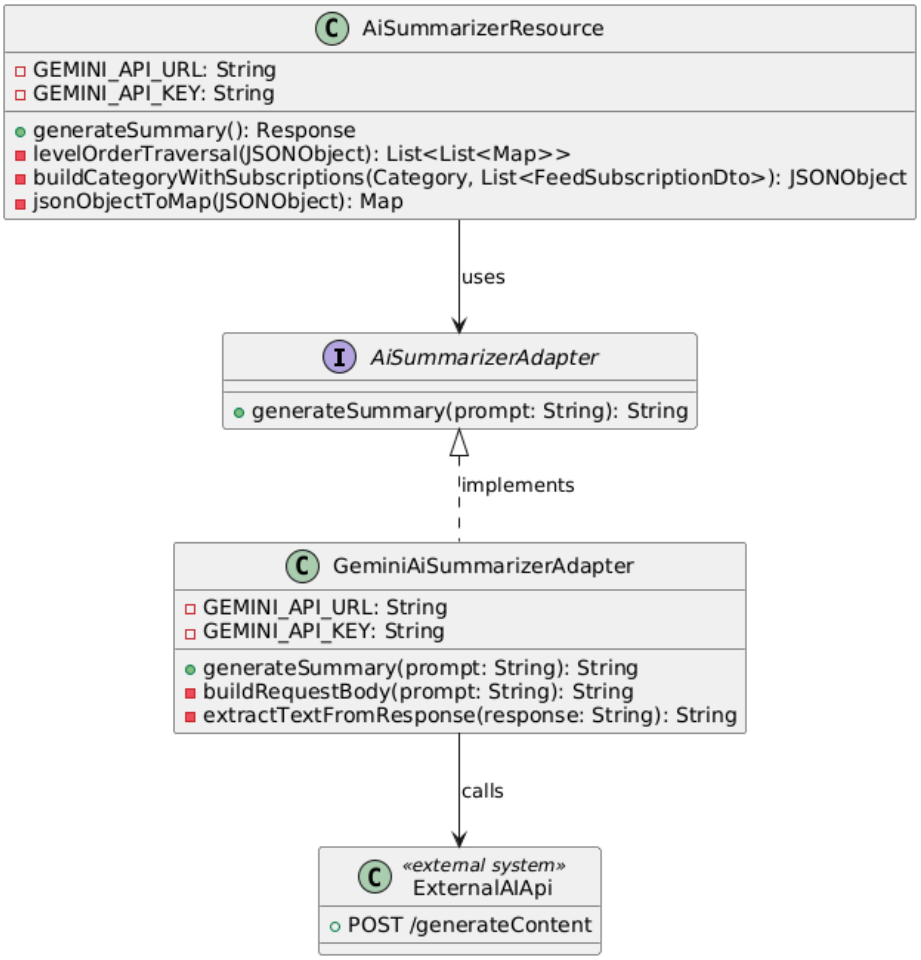
- A user creates or edits a category in the UI. They can optionally choose a parent category.
- The server checks the parent category's depth before allowing a new child category. If the parent is at level 5, it rejects with an error.
- A call to `GET /category` triggers `buildCategoryTree(...)`, which recursively gathers subcategories and sets them in `children`.
- The result is converted to nested JSON for the frontend.
- In `SubscriptionResource`, a similar tree-building process merges each category's feed subscriptions so the UI can show them in a collapsible structure.
- If a user changes a category's , we recalculate the depth and reorder. The composite structure remains consistent, and no cycles are allowed.

# AI-Based Daily Summary Generator

## Feature Details:

The feature generates AI-powered daily summaries from a user's subscribed feeds, giving higher weight to articles from top-level categories. It traverses the feed category tree, formats the summary using a generative AI model. We have used the **Adapter Design Pattern** to separate summarization logic from the core application.

## Design Pattern:




### Adapter Pattern:

The **Adapter Pattern** enables the application to interface with external AI services (e.g., Gemini API) without coupling the core business logic to a specific provider. It defines a common interface (`AiSummarizerAdapter`) that can be implemented for any AI provider such as Gemini, OpenAI, or Claude. Currently, only Gemini is used, but this design allows easy integration of other AI services as a future enhancement. We used the adapter pattern because of its below benefits :

- **Loose Coupling**: AI summarization logic is abstracted from the application logic.
- **Easy Extensibility**: Additional AI services can be integrated just by adding new adapter classes (e.g., `OpenAiSummarizerApiAdapter`, `ClaudeAiSummarizerApiAdapter`).
- **Code Reusability**: Multiple parts of the system can reuse the adapter interface for summarization in different contexts.

## Code Overview:

### 1. AiSummarizerAdapter.java

This defines a universal interface for all AI summarization adapters.

```
public interface AiSummarizerAdapter {
    String generateSummary(String prompt) throws Exception;
}
```

## 2. GeminiAiSummarizerAdapter.java

Implements the interface and encapsulates Gemini API call logic.

```
package com.sismics.reader.core.ai.adapter;

public class GeminiAiSummarizerAdapter implements AiSummarizerAdapter {

    private static final String GEMINI_API_URL = "https://generativelanguage.googleapis.com/v1beta/m
odels/gemini-2.0-flash:generateContent";
    private static final String GEMINI_API_KEY = "AIzaSyAA5×7L5KWC2tFA16JoattUtOmEt1-rTTo";

    @Override
    public String generateSummary(String prompt) throws Exception {
        URL url = new URL(GEMINI_API_URL + "?key=" + GEMINI_API_KEY);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();

        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);

        String requestBody = buildRequestBody(prompt);
        try (OutputStream os = connection.getOutputStream()) {
            byte[] input = requestBody.getBytes(StandardCharsets.UTF_8);
            os.write(input);
        }

        int responseCode = connection.getResponseCode();
        StringBuilder response = new StringBuilder();

        try (BufferedReader reader = new BufferedReader(new InputStreamReader(
                responseCode >= 200 ? connection.getInputStream() : connection.getErrorStream(),
                StandardCharsets.UTF_8))) {
            String line;
            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
        }

        connection.disconnect();
        return extractTextFromResponse(response.toString());
    }
```

```
    private String buildRequestBody(String prompt) {
        String escapedPrompt = prompt.replace("\"", "\\\"");
        return String.format("{\"contents\":[{\"parts\":[{\"text\":\"%s\"}]}]}", escapedPrompt);
    }

    private String extractTextFromResponse(String jsonResponse) throws JSONException {
        JSONObject responseObj = new JSONObject(jsonResponse);
        JSONArray candidates = responseObj.getJSONArray("candidates");
        JSONObject content = candidates.getJSONObject(0).getJSONObject("content");
        return content.getJSONArray("parts").getJSONObject(0).getString("text");
    }
}
```

### 3. AiSummarizerResource.java

This is the main REST endpoint that generates a prompt by traversing the feed-category tree and delegates summarization to the adapter. This ensures the controller remains decoupled from specific API logic and future adapters can be used without changes here.

```
package com.sismics.reader.rest.resource;


@Path("/aisummary")
public class AiSummarizerResource extends BaseResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response generateSummary() throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
              // Other Helper Functions
        AiSummarizerAdapter summarizerAdapter = new GeminiAiSummarizerApiAdapter();
        String summaryText;
        try {
            summaryText = summarizerAdapter.generateSummary(prompt);
        } catch (Exception e) {
            // error handling
        }
// Other Helper Functions

}
```

## Process Flow:

- **User sends a request** to generate a daily AI summary via the `/aisummary` endpoint.
- `AiSummarizerResource` **REST controller** authenticates the user and initiates the summary generation process.
- The system **retrieves all subscribed feeds and their associated categories** for the authenticated user.

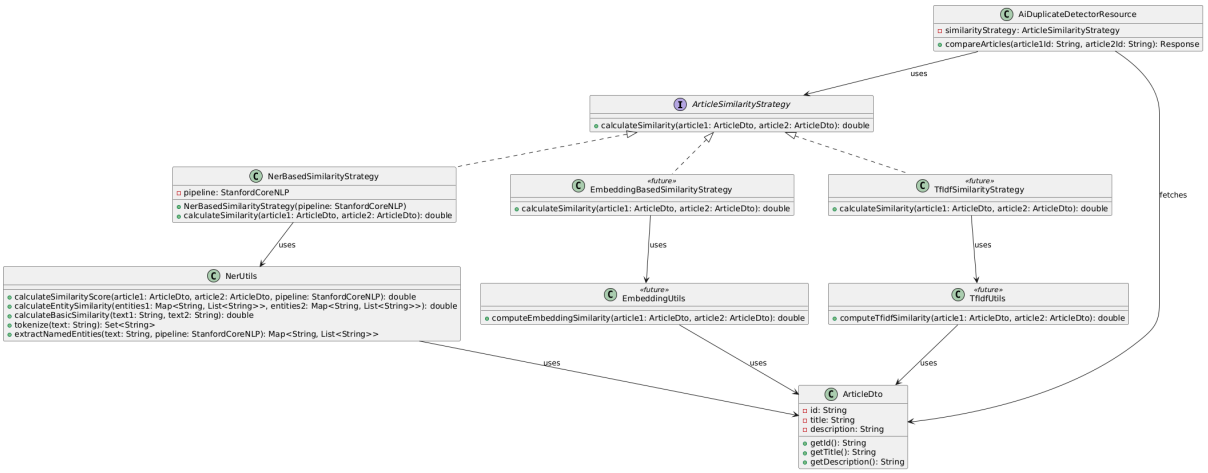
- A **level-order traversal of the category tree** is performed, giving **more weight to articles from top-level categories** and appropriately prioritizing the content.
- A **structured and formatted prompt** is dynamically generated based on the feed content, ensuring readability and logical segmentation.
- The prompt is passed to a **common** `AiSummarizerAdapter` **interface**, which abstracts the underlying AI summarization implementation.
- The system delegates the summarization task to a **concrete adapter implementation** (for example, currently an adapter for Gemini AI is used).
- The adapter handles communication with the **external AI service**, sends the prompt, and receives the generated summary.
- The **adapter returns the summary text** to the REST controller.
- The controller packages the result into a **structured JSON response** and sends it back to the client.

# Duplicate Detection Feature

## Feature Details

Earlier, the RSS Reader application lacked the ability to intelligently identify duplicate articles. To enhance user experience and avoid clutter, we introduced a **Duplicate Detection feature** using a combination of **Named Entity Recognition (NER)** and **keyword-based similarity scoring currently**. This functionality compares two user-selected articles and returns a similarity score along with a boolean flag indicating whether the articles are duplicates.

## Design Pattern




### Strategy Pattern

The **Strategy Pattern** has been used to encapsulate different similarity calculation algorithms (such as NER-based, keyword-based, or future ML-based) in their own strategy classes. This ensures that the application can easily switch between or extend similarity approaches without altering the core logic.

### Benefits of Applying Strategy Pattern:

- **Extensibility**: Easily add new similarity strategies (e.g., semantic matching) without affecting existing code.

- **Clean Separation of Concerns**: The REST layer doesn't worry about implementation details; it only depends on the interface.
- **Testability**: Each strategy can be tested independently.
- **Maintainability**: Changes in logic remain localized to the respective strategy class.

## Code Overview

### 1. `ArticleSimilarityStrategy` (Interface)

Defines a contract for all similarity strategies.

```
public interface ArticleSimilarityStrategy {
    double calculateSimilarity(ArticleDto article1, ArticleDto article2);
}
```

### 2. `NerBasedSimilarityStrategy` (NER + Token Matching Strategy)

Implements NER-based similarity and delegates computation to `SimilarityUtils`.

```
public class NerBasedSimilarityStrategy implements ArticleSimilarityStrategy {
    private final StanfordCoreNLP pipeline;

    public NerBasedSimilarityStrategy(StanfordCoreNLP pipeline) {
        this.pipeline = pipeline;
    }

    @Override
    public double calculateSimilarity(ArticleDto article1, ArticleDto article2) {
        return SimilarityUtils.calculateSimilarityScore(article1, article2, pipeline);
    }
}
```

### 3. `NerUtils` (Utility Class for Similarity Computation)

This class is responsible for performing the core similarity computation between articles. It handles tasks such as extracting named entities using Stanford CoreNLP, computing entity-type weighted similarity scores, calculating token-based similarity between article titles and descriptions, and finally combining these components into a single weighted similarity score.

```
public class NerUtils{
    public static double calculateSimilarityScore(...) { ... }
    public static double calculateEntitySimilarity(...) { ... }
    public static double calculateBasicSimilarity(...) { ... }
    public static Set<String> tokenize(...) { ... }
    public static Map<String, List<String>> extractNamedEntities(...) { ... }
}
```

### 4. `AiDuplicateDetectorResource` (REST Controller)

This class handles the REST API endpoint `/ai_duplicate_detector/compare_articles`. It is responsible for receiving two article IDs from the client, fetching the corresponding articles using the appropriate DAO classes, invoking

the `calculateSimilarity()` method through the selected strategy implementation, and finally returning the computed similarity score along with a duplicate flag in JSON format as the response.

```
@Path("/ai_duplicate_detector")
public class AiDuplicateDetectorResource extends BaseResource {

    // Initialize Stanford CoreNLP pipeline once for efficiency
    private static final StanfordCoreNLP pipeline;
    private final ArticleSimilarityStrategy similarityStrategy = new NerBasedSimilarityStrategy(pipeline);


    static {
        // Set up properties for the pipeline
        Properties props = new Properties();
        props.setProperty("annotators", "tokenize,ssplit,pos,lemma,ner");
        pipeline = new StanfordCoreNLP(props);
    }


    @GET
    @Path("/compare_articles")
    @Produces(MediaType.APPLICATION_JSON)
    public Response compareArticles(
        @QueryParam("article1Id") String article1Id,
        @QueryParam("article2Id") String article2Id) throws JSONException {

        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        UserArticleDao userArticleDao = new UserArticleDao();
        UserArticle userArticle = userArticleDao.getUserArticle(article1Id, principal.getId());
        if (userArticle == null) {
            throw new ClientException("ArticleNotFound", MessageFormat.format("Article not found: {0}",
article1Id));
        }

        ArticleDto article1 = new ArticleDao().findFirstByCriteria(
            new ArticleCriteria().setId(userArticle.getArticleId()));
        System.out.println(article1.getTitle());
        System.out.println(article1.getDescription());

        userArticle = userArticleDao.getUserArticle(article2Id, principal.getId());
        if (userArticle == null) {
            throw new ClientException("ArticleNotFound", MessageFormat.format("Article not found: {0}",
article2Id));
        }

        ArticleDto article2 = new ArticleDao().findFirstByCriteria(
            new ArticleCriteria().setId(userArticle.getArticleId()));
        System.out.println(article2.getTitle());
```

```
        System.out.println(article2.getDescription());

        System.out.println(userArticle);
        String userId = principal.getId();
        System.out.println("AiDuplicateDetectorResource userId: " + userId);
        System.out.println("Comparing articles with article ids: " + article1Id + " and " + article2Id);

        // Generate a similarity score between 0 and 100
        double similarityScore = similarityStrategy.calculateSimilarity(article1, article2);
        boolean isDuplicate = similarityScore > 50;

        // Create response object
        JSONObject response = new JSONObject();
        response.put("status", "ok");
        response.put("isDuplicate", isDuplicate);
        response.put("similarityScore", Math.round(similarityScore * 10) / 10.0); // Round to 1 decimal pla
ce

        return Response.ok().entity(response).build();
    }


}
```

## Process Flow

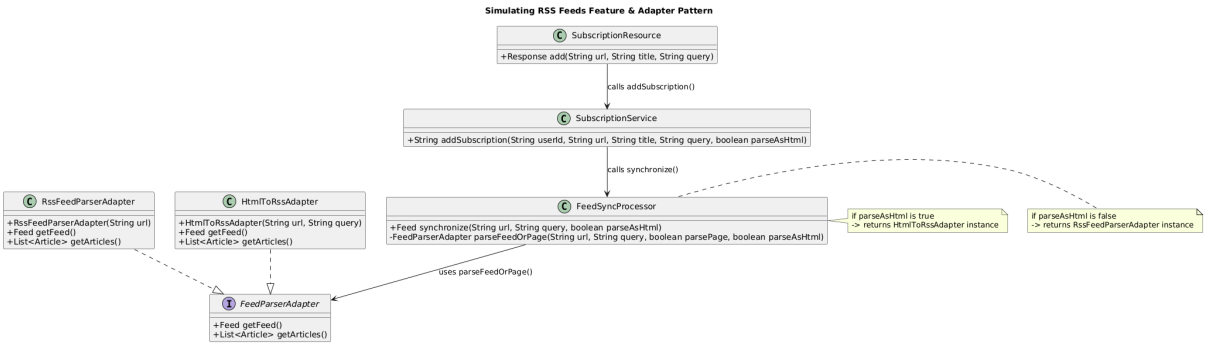
- User selects two articles from the UI and clicks "Compare".
- Frontend sends a GET request to the backend API: `GET /ai_duplicate_detector/compare_articles?article1Id=...&article2Id=...`.
- The `AiDuplicateDetectorResource` class handles the request.
- The system authenticates the user.
- The corresponding `ArticleDto` objects for the two article IDs are fetched using DAO classes.
- The `calculateSimilarity()` method is called using the selected strategy implementation (e.g., `NerBasedSimilarityStrategy`).
- The strategy class internally performs the similarity computation using techniques like Named Entity Recognition (NER), keyword matching, or any other implemented logic.
- The final similarity score (0–100) is calculated and a threshold (e.g., 50) is used to decide if the articles are duplicates.
- A JSON response is returned containing:
    - `"similarityScore"` – the computed similarity score
    - `"isDuplicate"` – a boolean indicating whether the articles are considered duplicates.

# Simulating RSS Feeds

## Feature Details

Many websites today do not offer native RSS feeds, making it challenging for users who rely on RSS readers to stay updated. To overcome this limitation, we implemented a feature that simulates RSS feeds. This feature extracts content from websites—using external APIs (like NewsAPI) when needed—and then formats the results into an RSS feed. The new functionality ensures that even if a website lacks a proper feed, users can still subscribe and receive updates with minimal API overhead.

# Design Pattern




## Adapter Pattern

The Adapter Pattern is central to this feature. It enables the system to treat both standard RSS feeds and simulated feeds uniformly by defining a common interface. The key components are:

- **FeedParserAdapter (Interface):** Establishes the contract for feed parsing by defining methods to retrieve the feed and its articles.
- **RssFeedParserAdapter (Class):** Implements the interface for normal RSS feeds by wrapping existing RSS parsing logic.
- **HtmlToRssAdapter (Class):** Implements the interface to simulate an RSS feed when a website lacks one. It uses the NewsAPI to fetch articles and converts the data into a feed structure.

By using this pattern, the rest of the system (e.g., the synchronization process) can operate without knowing whether the feed is coming from a standard RSS source or is being simulated. We make use of below benefits of the adapter pattern :

- **Uniform Interface:** Both standard RSS and simulated feeds are accessed uniformly through the `FeedParserAdapter` interface.
- **Loose Coupling:** Changes in one adapter (e.g., HTML-to-RSS simulation) do not affect other system components.
- **Simplified Feed Synchronization:** The FeedSyncProcessor can invoke `getFeed()` and `getArticles()` without extra conditional logic.
- **Ease of Extension:** New feed extraction methods can be added by implementing the `FeedParserAdapter` interface.

# Code Overview

## FeedParserAdapter.java

This interface defines the common methods required to retrieve the feed metadata and its articles.

```
public interface FeedParserAdapter {
    Feed getFeed() throws Exception;
    List<Article> getArticles() throws Exception;
```

```
}
```

### RssFeedParserAdapter.java

This adapter handles normal RSS feeds. It wraps the original RSS parsing logic inside an adapter that conforms to the common interface.

```
public class RssFeedParserAdapter implements FeedParserAdapter {
    private final RssReader rssReader;

    public RssFeedParserAdapter(String url) throws Exception {
        this.rssReader = new RssReader();
        new ReaderHttpClient() {
            @Override
            public Void process(InputStream is) throws Exception {
                rssReader.readRssFeed(is);
                return null;
            }
        }.open(new URL(url));
        rssReader.getFeed().setRssUrl(url);
    }

    @Override
    public Feed getFeed() {
        return rssReader.getFeed();
    }

    @Override
    public List<Article> getArticles() {
        return rssReader.getArticles();
    }
}
```

### HtmlToRssAdapter.java

This adapter simulates an RSS feed by extracting articles from an HTML page using NewsAPI. It creates a feed structure with a title, RSS URL, and an article list based on the API response.

```
public class HtmlToRssAdapter implements FeedParserAdapter {
    private final Feed feed;
    private List<Article> articles = new ArrayList<>();;
    private static final String NEWSAPI_KEY = "daa494d111554b3ca4d3081652d219dc"; // Replace with
your API key

    public HtmlToRssAdapter(String url, String query) throws Exception {
        String domain = extractDomain(url);
        this.feed = new Feed();
        this.feed.setTitle("News from " + domain + " about " + query);
        this.feed.setRssUrl("https://" + domain);
        try {
            this.articles = fetchArticlesFromNewsAPI(domain, query);
```

```
        } catch (Exception e) {
            this.articles = new ArrayList<>();
            System.err.println("Error fetching articles from NewsAPI: " + e.getMessage());
        }
        validateFeed();
        fixGuid();
    }



    /**
     * Fetch articles using NewsAPI
     */
    private List<Article> fetchArticlesFromNewsAPI(String domain, String query) throws Exception {
        List<Article> articleList = new ArrayList<>();
        // String apiUrl = "https://newsapi.org/v2/everything?q=" + url + "&apiKey=" + NEWSAPI_KEY;
        String apiUrl = String.format(
            "https://newsapi.org/v2/everything?q=%s&domains=%s&apiKey=%s",
            query, domain, NEWSAPI_KEY
        );

        String jsonResponse = sendGetRequest(apiUrl);
        JSONObject jsonObject = new JSONObject(jsonResponse);
        JSONArray articlesArray = jsonObject.getJSONArray("articles");
        for (int i = 0; i < articlesArray.length(); i++) {
            JSONObject articleJson = articlesArray.getJSONObject(i);
            Article article = new Article();
            article.setTitle(articleJson.getString("title"));
            article.setUrl(articleJson.getString("url"));
            article.setDescription(articleJson.optString("description", "No description available"));
            // article.setPublicationDate(articleJson.optString("publishedAt", ""));
            String publishedAt = articleJson.optString("publishedAt", "");
            Date parsedDate = DateUtil.parseDate(publishedAt, DF);
            article.setPublicationDate(parsedDate);
            articleList.add(article);
        }

        return articleList;
    }

    /**
     * Send an HTTP GET request
     */
    private String sendGetRequest(String apiUrl) throws Exception {
        URL url = new URL(apiUrl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "application/json");

        BufferedReader br = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        StringBuilder response = new StringBuilder();
```

```
    String line;
    while ((line = br.readLine()) != null) {
        response.append(line);
    }
    br.close();
    return response.toString();
}

@Override
public Feed getFeed() {
    return feed;
}

@Override
public List<Article> getArticles() {
    return articles;
}

private void validateFeed() throws Exception {
    if (feed == null) {
        throw new Exception("No feed found");
    }
}

/**
 * Try to guess a value for GUID element values in RSS feeds.
 */
private void fixGuid() {
    if (articles != null) {
        for (Article article: articles) {
            GuidFixer.fixGuid(article);
        }
    }
}

private String extractDomain(String url) throws MalformedURLException {
    URL parsedUrl = new URL(url);
    return parsedUrl.getHost(); // e.g., "bbc.com"
}

public static final DateTimeFormatter DF = new DateTimeFormatterBuilder()
        .append(null, new DateTimeParser[] {
                DateTimeFormat.forPattern("EEE, dd MMM yyyy HH:mm zzz").getParser(),
                DateTimeFormat.forPattern("EEE, dd MMM yyyy HH:mm:ss Z").getParser(),
                DateTimeFormat.forPattern("EEE, dd MMM yyyy HH:mm:ss zzz").getParser(),
                DateTimeFormat.forPattern("EEE,  d MMM yyyy HH:mm:ss zzz").getParser(),
                DateTimeFormat.forPattern("dd MMM yyyy HH:mm:ss Z").getParser(),
                DateTimeFormat.forPattern("yyyy-mm-dd HH:mm:ss").getParser(),
                DateTimeFormat.forPattern("EEE, dd MMM yyyy HH:mm:ss").getParser(),
                DateTimeFormat.forPattern("dd MMM yyyy HH:mm:ss zzz").getParser(),
                DateTimeFormat.forPattern("EEE MMM dd yyyy HH:mm:ss 'GMT'Z Z").getParser(),
```

```
                DateTimeFormat.forPattern("yyyy-MM-dd'T'HH:mm:ssZ").getParser(),
                DateTimeFormat.forPattern("yyyy-MM-dd'T'HH:mm:ss.SSSZ").getParser(),
                DateTimeFormat.forPattern("yyyy-MM-dd'T'HH:mm:ssZ").getParser(),
                DateTimeFormat.forPattern("yyyy-MM-dd'T'HH:mm:ss.SSSZ").getParser(),
            }).toFormatter().withOffsetParsed().withLocale(Locale.ENGLISH);

}
```

### SubscriptionResource.java

In the `add()` method, we added a new `query` parameter. When the query is not empty, a boolean flag (`parseAsHtml`) is set to true. This flag is then passed to the subscription service to simulate an RSS feed.

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
public Response add(
        @FormParam("url") String url,
        @FormParam("title") String title,
        @FormParam("query") String query) throws JSONException {
    authenticateOrThrow();
    boolean parseAsHtml = (query != null && !query.trim().isEmpty());
    String subscriptionId = subscriptionService.addSubscription(principal.getId(), url, title, query, parse
AsHtml);
    JSONObject response = new JSONObject();
    response.put("id", subscriptionId);
    return Response.ok().entity(response).build();
}
```

### SubscriptionService.java

In the `addSubscription()` method, the service now calls the feed synchronization processor with the additional `query` and `parseAsHtml` parameters:

```
FeedSyncProcessor syncProcessor = AppContext.getInstance().getFeedService().getSyncManager().
getFeedSyncProcessor();
Feed feed;
try {
    feed = syncProcessor.synchronize(url, query, parseAsHtml);
} catch (Exception e) {
    throw new ServerException("FeedError", MessageFormat.format("Error retrieving feed at {0}", url),
e);
}
```

### FeedSyncProcessor.java

Within the `parseFeedOrPage()` method, the logic is updated to choose between the standard RSS adapter and the HTML-to-RSS adapter based on the `parseAsHtml` flag:

```
private FeedParserAdapter parseFeedOrPage(String url, String query, boolean parsePage, boolean pa
rseAsHtml) throws Exception {
    try {
        if (parseAsHtml) {
```

```
            return new HtmlToRssAdapter(url, query);
        } else {
            return new RssFeedParserAdapter(url);
        }
    } catch (Exception e) {
        // Recovery and logging logic omitted for brevity
        throw e;
    }
}
```

## Process Flow / Code Flow :

- A user submits a subscription request with a URL, title, and an optional query; the `SubscriptionResource.add()` method sets the `parseAsHtml` flag based on whether the query is provided.
- The `SubscriptionService.addSubscription()` method receives the URL, title, query, and `parseAsHtml` flag, then calls the feed synchronization processor.
- The `FeedSyncProcessor.synchronize()` method is invoked; inside it, the `parseFeedOrPage()` method decides which adapter to use:
  - If `parseAsHtml` is true, an instance of `HtmlToRssAdapter` is created.
  - Otherwise, an instance of `RssFeedParserAdapter` is created.
- The chosen adapter processes the URL:
  - `RssFeedParserAdapter` parses a standard RSS feed.
  - `HtmlToRssAdapter` calls NewsAPI (using the provided query) to fetch articles and creates a simulated RSS feed.
- Both adapters return a feed object and a list of articles that conform to the common interface.
- The feed and its articles are further processed—completing article data, handling deletions, and updating metadata.
- Finally, the subscription is created and stored.

# User-Created Feeds

## Feature Details

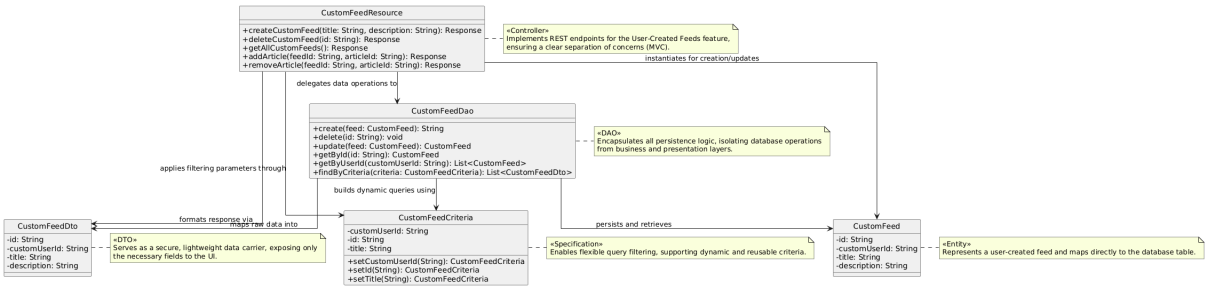
The existing RSS Reader application has been enhanced with a new feature — **User-Created Feeds** — implemented completely from scratch. This feature allows users to create their own **custom feeds**, comprising articles selected from various sources. These curated feeds can then be **shared across the platform**, enabling **other users to subscribe** and view these user-curated content collections.

Users can:

- Create a **Custom Feed** by specifying a title and description.
- Add **articles** to their custom feed.
- **Subscribe or unsubscribe** to custom feeds created by other users.
- View **articles from feeds they have subscribed to**.

This transforms the RSS Reader into a **personalized content sharing platform**, fostering community-based content curation.

# Design Pattern




## 1. Model-View-Controller (MVC) / Layered Architecture

The codebase for user-created feeds clearly follows the MVC layered architecture. The model layer contains entity classes like `CustomFeed`, `CustomArticle`, and `CustomFeedSubscription`, which directly map to database tables. The DAO layer manages database interactions and REST resource classes act as controllers. This separation improves modularity, simplifies debugging, and promotes independent development and testing of each layer.

## 2. DAO (Data Access Object) Pattern

For each entity (CustomFeed, CustomArticle, and CustomFeedSubscription), a corresponding DAO class handles all database-related operations such as creation, deletion, and querying. This encapsulation of persistence logic ensures that business logic or controllers don't need to worry about database details, thus improving maintainability and making the data access layer easily replaceable or upgradable in the future.

## 3. DTO (Data Transfer Object) Pattern

DTOs like `CustomFeedDto`, `CustomArticleDto`, and `CustomFeedSubscriptionDto` are used to carry only the required data from the DAO layer to the controller/UI layer. These DTOs ensure that internal database entities remain encapsulated, reducing security risks and data redundancy in API responses. The use of DTOs also enables flexible API structuring, allowing future changes without altering the entity layer.

## 4. Specification (Criteria) Pattern

Criteria classes such as `CustomFeedCriteria` and `CustomArticleCriteria` encapsulate filtering parameters like `userId`, `title` These criteria objects are passed to DAO methods to dynamically construct database queries. This pattern enhances flexibility by enabling query reuse and easily supporting new filters in the future without changing the core DAO methods.

# Code Overview

We have displayed below all the newly created classes for implementing the **User-Created Feeds** feature. The implementations for `CustomFeedSubscription`, `CustomArticle`, and their respective classes follow a similar structure and pattern as shown for `CustomFeed`.

## `CustomFeed.java` (Model Layer - Entity)

This class represents the entity for a user-created feed and maps to the `T_CUSTOMFEED` table in the database. It contains fields for feed ID, user ID, title, and description, with appropriate annotations for persistence mapping.

```
@Entity
@Table(name = "T_CUSTOMFEED")
public class CustomFeed {
```

```
    @Id
    @Column(name = "CFD_ID_C", length = 50)
    private String id;

    @Column(name = "CFD_CUSTOMUSERID_C", nullable = false, length = 36)
    private String customUserId;

    @Column(name = "CFD_TITLE_C", nullable = false, length = 100)
    private String title;

    @Column(name = "CFD_DESCRIPTION_C", nullable = false, length = 4000)
    private String description;
}
```

### `CustomFeedDao.java` (DAO Layer)

This class handles all database operations related to custom feeds, such as creating, retrieving, updating, and deleting feeds. It also supports dynamic querying via criteria-based filtering and mapping results to DTOs.

```
public class CustomFeedDao extends BaseDao<CustomFeedDto, CustomFeedCriteria> {
    public String create(CustomFeed customFeed) {
        customFeed.setId(UUID.randomUUID().toString());
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.persist(customFeed);
        return customFeed.getId();
    }

    public CustomFeed update(CustomFeed customFeed) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        CustomFeed existing = em.find(CustomFeed.class, customFeed.getId());
        existing.setCustomUserId(customFeed.getCustomUserId());
        existing.setTitle(customFeed.getTitle());
        existing.setDescription(customFeed.getDescription());
        return existing;
    }

    public List<CustomFeedDto> findByCriteria(CustomFeedCriteria criteria) {
        // Builds and executes dynamic query using criteria fields
    }
}
```

### `CustomFeedDto.java` (DTO Layer)

This class serves as a data transfer object for sending structured custom feed data from DAO to controller layers. It contains only the fields required at the UI/controller level, decoupling internal database schema from presentation logic.

```
public class CustomFeedDto {
    private String id;
    private String customUserId;
    private String title;
```

```
    private String description;
}
```

### `CustomFeedCriteria.java` (Criteria Layer)

This class allows dynamic construction of query filters based on user inputs such as custom user ID, feed ID, or title. It helps modularize and decouple filtering logic from DAO methods.

```
public class CustomFeedCriteria {
    private String customUserId;
    private String id;
    private String title;

    public CustomFeedCriteria setCustomUserId(String customUserId) { this.customUserId = customUs
erId; return this; }
    public CustomFeedCriteria setId(String id) { this.id = id; return this; }
    public CustomFeedCriteria setTitle(String title) { this.title = title; return this; }
}
```

### `CustomFeedMapper.java` (Mapper Layer)

This class acts as a result mapper that **converts native SQL query result sets into DTO objects**. It is used by the DAO to map each row of a query result into a corresponding `CustomFeedDto`, ensuring a clean separation between raw database result parsing and data transfer.

```
public class CustomFeedMapper extends ResultMapper<CustomFeedDto> {
    @Override
    public CustomFeedDto map(Object[] o) {
        int i = 0;
        CustomFeedDto dto = new CustomFeedDto();
        dto.setId(stringValue(o[i++]));
        dto.setCustomUserId(stringValue(o[i++]));
        dto.setTitle(stringValue(o[i++]));
        dto.setDescription(stringValue(o[i]));
        return dto;
    }
}
```

### SQL Schema Changes

```
-- Custom Feed Table
CREATE CACHED TABLE T_CUSTOMFEED (
  CFD_ID_C VARCHAR(36) NOT NULL,
  CFD_CUSTOMUSERID_C VARCHAR(36) NOT NULL,
  CFD_TITLE_C VARCHAR(100) NOT NULL,
  CFD_DESCRIPTION_C VARCHAR(4000) NOT NULL,
  PRIMARY KEY (CFD_ID_C)
);
ALTER TABLE T_CUSTOMFEED ADD CONSTRAINT FK_CFD_CUSTOMUSERID_C
FOREIGN KEY (CFD_CUSTOMUSERID_C) REFERENCES T_USER (USE_ID_C)
ON DELETE CASCADE ON UPDATE CASCADE;
```

```
-- Custom Article Table
CREATE CACHED TABLE T_CUSTOMARTICLE (
  CFA_ID VARCHAR(36) NOT NULL,
  CFA_IDCUSTOMFEED_C VARCHAR(36) NOT NULL,
  CFA_TITLE_C VARCHAR(4000),
  CFA_DESCRIPTION_C LONGVARCHAR,
  CFA_URL_C VARCHAR(2000),
  CFA_PUBLICATIONDATE_D DATETIME,
  PRIMARY KEY (CFA_ID)
);
ALTER TABLE T_CUSTOMARTICLE ADD CONSTRAINT FK_CFA_IDCUSTOMFEED_C
FOREIGN KEY (CFA_IDCUSTOMFEED_C) REFERENCES T_CUSTOMFEED (CFD_ID_C)
ON DELETE CASCADE ON UPDATE CASCADE;

-- Subscription Table
CREATE CACHED TABLE T_CUSTOM_FEED_SUBSCRIPTION (
  CFS_ID_C VARCHAR(36) NOT NULL,
  CFS_IDUSER_C VARCHAR(36) NOT NULL,
  CFS_IDFEED_C VARCHAR(36) NOT NULL,
  CFS_TITLE_C VARCHAR(100)
);
ALTER TABLE T_CUSTOM_FEED_SUBSCRIPTION ADD CONSTRAINT FK_CFSUSER_IDCUSTOMFEED_C
FOREIGN KEY (CFS_IDUSER_C) REFERENCES T_USER (USE_ID_C) ON DELETE CASCADE ON UPDATE
CASCADE;
ALTER TABLE T_CUSTOM_FEED_SUBSCRIPTION ADD CONSTRAINT FK_CFSID_USER
FOREIGN KEY (CFS_IDFEED_C) REFERENCES T_CUSTOMFEED (CFD_ID_C) ON DELETE RESTRICT ON
UPDATE RESTRICT;
```

## Process Flow

- User creates a custom feed through the frontend form.
- The request is handled by the controller layer (resource class), which invokes `CustomFeedDao.create()` to persist the feed in the `T_CUSTOMFEED` table.
- The user adds articles to their custom feed; the controller calls `CustomArticleDao.addArticle()` to insert each article into the `T_CUSTOMARTICLE` table.
- Other users view available custom feeds and subscribe to them; their subscriptions are stored using `CustomFeedSubscriptionDao.subscribe()` in the `T_CUSTOM_FEED_SUBSCRIPTION` table.
- Users can view the custom feeds they have subscribed to; the controller retrieves these using `CustomFeedDao.findByCriteria()` with filters like user ID.
- The controller layer handles all REST API communication between frontend and backend.
- DTOs are used to transfer structured and filtered data from DAOs to the frontend, ensuring clean and secure data exchange.