An-Najah National University
Faculty Of Engineering And Information Technology

Computer Engineering Department

GRADUATION PROJECT II

QUENTITY

< BY QUENTITY NOT BY QUANTITY>

Prepared by:

AMR KURDI
DEEMA SALEH

Supervised by:

Dr.MUHANNAD AL-JABI

January 28,2025

Acknowledgements

"We would like to express our heartfelt gratitude to Dr. Muhannad Al-Jabi, our supervisor, for his dedication and continuous support throughout our project. His valuable scientific guidance has been invaluable to our success. We would also like to thank the academics in the Department of Computer Engineering for their expertise and assistance, as well as our friends and family for their belief in our abilities. We are deeply grateful for their contributions, support, and belief in our capabilities."

Disclaimer

The following report has been authored by students from the Computer Engineering Department, Faculty of Engineering, An-Najah National University. The report has undergone minimal modifications, limited to editorial corrections, and may still contain errors in language and content. It is important to note that the opinions expressed within the report, including any conclusions and recommendations, solely belong to the students. An-Najah National University bears no responsibility or liability for any consequences arising from the utilization of this report for purposes other than its intended commission.

Table of Contents

List of Figures

Abstract

This project introduces Quentity, a Java-based framework developed to make application development easier by automatically creating the back-end,front-end, and database.

Quentity saves work by having the developer define a single class for each entity, with the field types, their validation rules, and behaviors.

The framework then produces the database schema and front-end components from this definition so that the integration of all layers will be seamless.

It encapsulates communications between the front-end and back-end, including input validation on both the front-end and back-end sides, and efficiently executes tasks by paginating them, sorting directly from the database, among others.

It's powered with a strong search engine using Lucene. Multilingual is implemented in the framework; it only needs the localization of resources to provide multiple language functions.

In short Quentity framework allows for rapid development of responsive full-stack web applications by writing just a minimal amount of code.

Chapter 1

1 Introduction

Full-stack web application development is complex and time-consuming in modern fast-moving software development. The developers have to face big difficulties in effectively integrating front-end, back-end, and database layers, with performance, scalability, and maintainability assured. Quentity addresses these challenges by providing a Java-based framework for simplifying application development through automation and abstraction.

Quentity simplifies the development process by allowing the developer to define entities with a minimum of effort. It automatically builds backend logic, frontend components, and database schemes. Some of the inbuilt features include input validation, seamless communication of front-end-back-end, efficient data management by pagination and sorting, multilingual support, which make it ideal for rapid application development.

The project illustrates Quentity's capability in developing a Website Builder and shows well how the underlying framework empowers developers to create responsive, full-of-features web applications with a minimal amount of lines of code, raising productivity and reducing development time.

1.1 Problem statement

Web development is not easy-it requires integration at three layers:front-end, back- end, and database, all of which generally require different codebases, adding time to development while giving rise to inconsistencies. Besides that, communication, performance, and features like input validation and multilingual support increase the complexity.

1.2 Objectives

- Automate the back-end, front-end, and database layers' generation.
- Enable seamless integration or communication among an application's peer parts layers.
- Provide built-in support for validation, pagination, sorting, and multilingua capability.

- Reduce development time and effort by reducing manual coding.
- Provide mobile application for both IOS and Android.
- Provide Native desktop application for all Windows, Mac, and Linux.
- The aim of this project is to reveal how Quentity smooths web development in an easier, scalable, and more maintainable manner.

1.3 Scope of work

- **Framework implementation:**Set up Quentity to automate back-end, front-end, and database generation from entity definitions.
- **Website Builder:** A dynamic website builder that will be used to help prove Quentity is capable of developing responsive web applications with a minimum amount of coding.
- **Database and API Management:** Generate database schemes and expose RESTful API endpoints for CRUD operations.
- **Performance Optimization:** Add pagination, sorting, and Lucene-powered search to enhance efficiency.
- **Multilinguality:** Realize the capability of multi-language by resource localization.
- **Testing and validation:**Ensuring system accuracy, performance, and smooth front-end–back-end communication
- **Documentation and deployment:**Providing usage documentation and deploying the solution for demonstration purposes.

1.4 Signifcance

This project presents the Quentity framework, which changes the whole paradigm in web development-automating backend, frontend, and database creation. Quentity increases productivity by reducing manual coding; hence, it reduces the development time with consistency. The provided features are validation, pagination, sorting, and multilingual, making it easy for developers to build scalable and efficient applications. Quentity is ideal for full-stack development because seamless front-end and back-end integration in its core can satisfy rapid needs.

Development and hence ideal for businesses to develop responsive web applications effectively.

Chapter 2
Constraints and Earlier work

2.1 Constraints and limitations

We have overcome so many difficulties and challenges since the very beginning of this project.

- **Large Project Scalability:** Although Quentity makes the development of small to medium-scale applications easier, complex large-scale enterprise-level projects need to be optimized and tuned.

- **Dependence on Java:**As Quentity is a pure Java-based framework, it might not be of direct use for those preferring other languages or ecosystems.

- **Limited Community and Support:** Quentity is a rather new framework; hence, it doesn't have such an active community for developers and rich documentation that would be an important part for its extensive implementation.

2.2 Standards

2.2.1 Coding Standards

- **Java Coding Convention:** Clean, readable, maintainable; according to Oracle standards during coding in Java.
- **SOLID Principles:**This improves scalability and reusability due to the following object-oriented design principles.
- **Lombok annotations**: provide a convenient way to avoid boilerplate for common patterns like getters, setters, and constructors.

2.2.2 Security Standards

- **OWASP Guidelines** (Open Web Application Security Project): Best practices to protect against common vulnerabilities

  such as:

  SQL Injection: prevents injection of malicious SQL code into database queries.

  CSRF: This provides protection to the users from forged requests executed on their behalf without their consent.
  For example

  about CSRF:
  To enhance security when logging in as by an any user, the following mechanism is applied:

1. **Cookies in Header:**
   A **Cookie** is stored in the **Header** containing a unique identifier (e.g., token or address) to validate the user's session.
2. **Payload in Body:**
   In the request Body, a CSRF Token is included so that the form validation process knows whether the update or deletion action is being made by an authorized user.
3. **Purpose:**
   It prevents the attackers from spoofing requests; hence, the request comes from a legitimate by any user.

- **Data Encryption**: To keep sensitive data secured, advanced encryption techniques are used including:

AES: Advanced Encryption Standard, which secures sensitive information, such as passwords, in a manner not easily accessed by unauthorized personnel.

HTTPS Protocol: It allows data transmission to be very secure, right from the client to the server; and all communications are encrypted so they will not be interfered with.

Benefits:

- Protection of data from theft or tampering when in storage and/or in transit
- Enhances user trust in the system's security

- **Role-Based Access Control (RBAC)**: implements a fine-grained access control mechanism which authorizes access depending on user roles:

- A certain user gets the permissions included in his role, by an any user.
- A user who doesn't have any permissions can't perform restricted actions.

How It Works:

1. A user logs in; his role is checked.
2. Sensitive actions like Edit/Delete are granted to authenticated users only.
3. Disallow access that is unauthorized. Log·

Benefits:

- Prevention of illegal activities present in the system.
- Allow every user to work strictly within their premises.

Summary : For example, it provides CSRF Token security features to any user who logs in, which guarantees that protected actions are performed by authorized persons. It plays together with encryption and role-based access control in making the system secure as well as reliable.

2.2.3 Frontend Standards

- **HTML5 and CSS3 Compliance**: Cross-browser compatibility and responsiveness to create a consistent user experience across all browsers.

2.3 Earlier coursework

1. Object-Oriented Programming (OOP)

- Wrote modular, re-usable code with concepts such as encapsulation, inheritance, and Polymorphism

- Supported the Java-based implementation of the core framework for your project.

2. Database Systems

- Relational database design, schema normalization, SQL queries

- Automated Database Schema Generation with its direct application

3. Web Development

- Front-end/Back-end Integration, User Interface Communication
- Laid the foundation to realize seamless communication between application layers.

4. Software Engineering

- Development Methodologies, Project Management, Version Control.

- Ensured high quality of codes with effective collaboration within a team.

5. Algorithms and Data Structures

- Enhancement in problem-solving by efficient utilization of algorithms along with proper representation of data.

- Implemented on Project Pagination, data sorting, retrieving of data.

Chapter 3
Literature Review

Basically, the need for Quentity has grown from the present market demand when almost all software development focuses on financial and management systems, adding e-commerce to the list. We wanted to speed up development because time means money. We started with financial and management systems, but soon we plan scaling our work to include E-commerce.

Quentity also saves developers' time and effort by giving them an extensible framework whereby the hassle of handling frontend, backend, and database management is significantly minimized. The developers will be free to actually build the features instead of handling the technical complexities, hence delivering faster and being more productive because automation of complex tasks like communication and security will be taken care of.

1. **Limitations of Current Frameworks:**
This section describes what is not in existing tools and frameworks, hence the need for a solution like Quentity.

2. **Distinctive Contribution of Quentity:**
It proves the feature and novel angle in respect of Quentity as compared to other frameworks.

3. **Solved Integrations Challenges:**
This tells how Quentity solves problems at the front-end, back-end, and database layers of smooth integrations.

Each of them will be described in detail, but the importance and contribution of the project to the web application development area will be explained consecutively and logically.

Chapter 4
Methodology

4.1 Tools, Programming Languages, APIs Technologies

At every stage of development in modern software solution development, different programming languages, tools, frameworks, and technologies were used to assure efficiency, scalability, and flexibility.
Below are presented the main technologies used in this project:

Java: A powerful, platform-independent language used in the development of back-ends with frameworks like Spring.

Postman : Testing and debugging of RESTful APIs to ensure that everything is in place and working.

RESTful API: For the integration of the back-end with external services using HTTP methods, such as GET, POST, PUT, and DELETE.

HTTP : The protocol used for client-to-server communication, responsible for resource fetching on the web.

Websocket: It provides real-time, bidirectional communication between clients and servers at low latency.

Spring Framework: Java framework for comfortably building Back-ends and respective features like dependency injection or security.

Vaadin: Java Web Front-end framework for building web applications interactively, targeting the modern web browsers.

Apache Lucene: A search engine library for high-performance text indexing and searching of applications.

QueryDSL: An intuitive framework for defining type-safe and dynamic database queries.

Electron: A framework of cross-platform desktop applications using web technologies, namely HTML, CSS, and JS.

Capacitor: Cross-platform framework for building mobile applications using web technologies but with native functionality access.

4.2 Features of The Framework

4.2.1 Embedded full text search

Quentity supplies an embedded full-text search with the Apache Lucene-based high-power functionality, enabling complex searches to be executed against enormous volumes of data with very high efficiency.

Benefits

High-Performance: Gets results quickly even when the amount of data is enormous.

Advanced Querying: Enables searching by keyword, phrase, filter range, including date and number.

Indexing: Data indexing automatically provides quick retrieval and scaling. It utilizes the indexing and querying in Lucene to provide both structured and unstructured data.

It allows searching in several fields: text, numeric, and date types. It can be integrated with database queries in order to combine structured search with full-text capability.

4.2.2 Multilingual Support.

Quentity is with native support for multilingual applications, meaning an application can serve its users in more than one language.
This functionality is an essential one when working with financial and management systems, or e-commerce platforms, targeting audiences spread around different parts of the world.

Benefits

User Experience: Enables users to interact with the application in their own preferred language.

Scalability: New languages are added very easily without changing the core application logic.

Implementation: Resource bundles store translations.
Dynamic switching of languages is supported out-of-the-box without application restarts. Adding a new language is needed with a restart for loading configurations, but it works dynamically while changing to an already present language.

Benefit: This will balance user flexibility with developer scalability.

Front-end Framework Integrations: The Vaadin framework among others automatically adapts UI to the language and makes it very easy to change LTR/RTL depending on the language.

4.2.3 Quentity

1. FldNumber Class

Description: The FldNumber class implements the numeric field of an entity, allowing the management and validation, or the handling of changes of numeric values. Allows integration into databases and full-text search.

The class represents a numeric field in an entity. It is used for managing, validation, or handling changes of numeric values, and it supports the integration of this into a database and full-text search indexing.

Key Features:

Numeric Value Management:

1. Stores and processes numeric values using a Double data type.
2. Provides getter and setter methods for managing the field's value.

Validation:

1. Ensures that numeric inputs meet predefined constraints (e.g., range validation, null checks).
2. Invalid inputs are flagged with error messages, giving users immediate feedback.

Real-Time Feedback:

1. Synchronizes validated values with the UI and backend models for consistent user experience.

Error Handling:

1. Marks the field as invalid if the value fails validation and displays the appropriate error messages.

Full-Text Search Integration:

1. The @IndexedEmbedded annotation allows the numeric field to be included in search operations.

Usage Context:

- Suitable for handling numeric data in applications, such as prices, quantities, or measurements.
- Ensures data integrity by validating inputs and providing error feedback.
- Integrates seamlessly into search and database systems for enhanced functionality.

2. SingleEntityReference<T> class

Description:class is a custom field class that allows developers to refer to one entity from another entity.

It provides:

1. **Entity Selection**: A dropdown (ComboBox) for selecting an entity.
2. **Customizable UI**: Integration with Vaadin for visual representation.
3. **Dynamic Behavior**: Field change listeners and validation.

Key Annotations

@DieTogether//Deleting the student will delete the mark public

SingleEntityReference<Student> student;

Fields

Field Name	Type	Description
entity	T	The referenced entity.
required	boolean	Whether the field is mandatory.
visibleField	boolean	Whether the field is visible.
editable	boolean	Whether the field is editable.

Key Features

1. UI Components

The class uses Vaadin components to create a user-friendly interface:

- **ComboBox<T>**: Displays a list of entities for selection.
- **Button:** Allows adding new entities directly from the UI.

2. Data Binding

- The field is bound to an entity using @ManyToOne.
- Developers can dynamically populate the ComboBox with entity data using the reflect() method

3. Validation

The validateValue() method provides for bespoke validation logic to be applied when ensuring the being set entity is valid.

4. Field change

The **onFieldChanged()** method enables attaching your own behaviour once the value of the field is changed

5. Entity Reflection

The **reflect()** method uses the entity class with which the binding is performed for the ComboBox will have at runtime.

6. Entity Representation

The **getEntityTitle()** method returns the human readable entity for presentation purposes.

7. Custom Rendering

The **createRenderer()** method is used to customize the appearance of entities in ComboBox using Vaadin's ComponentRenderer.

8. Dynamic Labeling

Labels for the ComboBox can be dynamically generated using LanguageUtil.

How to Use

Add a SingleEntityReference Field in Your Entity

Configure the Field Set properties like required, editable, and visible

Bind to an Entity Class Use the reflect() method to bind the field to an entity class

Handle Field Changes Add a listener for changes to the field

Summary

The SingleEntityReference class provides powerful tools for handling one-to-one relationships in the Quentity framework. By leveraging its dynamic UI, validation, and data-binding features, developers can create rich, interactive forms and entity management systems. This guide offers everything you need to understand and use this class effectively.

3. NSSingleEntityReference<T> class

Description: NSSingleEntityReference is a quite customizable and extensible class, thus very reusable inside the Quention framework to support functionality related to the management of single entity references. It is a subclass of InternalSingleEntityReference, which, together with the handling of entity references, enables the integration of back-end logic-saving of entities together with user-defined callbacks and context data.

Key Components

1. Core Properties

This class maintains several core properties to handle entity references effectively:

- entity (T): Represents the referenced entity.
- skipDefaultOnSave: A flag to skip the default saving mechanism for the entity.
- onSaveCallback: A consumer to execute additional logic when saving.
- contextData: A Map to store metadata or additional information related to the entity reference.
- doneEntitySave: A flag to prevent cyclic or redundant save operations.

2. Save Logic with Callbacks

The onSave method handles the saving of the referenced entity:

1. Validates the entity.
2. Executes a user-defined callback (onSaveCallback), if provided.
3. Handles default save logic unless skipDefaultOnSave is true.

3. Setting the Entity

The setEntity and setEntityWithContext methods allow assigning the referenced entity while optionally attaching contextual data or executing logic.

4. Context Data Management

You can manage and attach metadata or additional information using the contextData method, which accepts a Consumer for modifications.

5. Callback Management

You can assign a custom callback to execute additional logic during the save process.

6. Equality and HashCode

The equals and hashCode methods are overridden to ensure equality checks are based on the referenced entity.

How to use

1. Create a Single Entity Reference

Developers can use the NSSingleEntityReference class to reference any entity type that extends the Entity class.

2. Set and Manage Entities

3. Skip Default Save Logic

Developers can skip the default save logic by setting skipDefaultOnSave to true.

4. Attach Context Data

Developers can attach metadata to enhance the entity reference with additional information.

5. Ensure Equality

Equality is based on the referenced entity, making comparisons between references straightforward.

Summary

The NSSingleEntityReference class in the **Quentity framework** provides a flexible and powerful way to manage single entity references, offering features such as:

1. **Custom Callbacks**: Developers can define custom logic during the save process.
2. **Contextual Data**: Attach additional metadata or runtime context to the entity reference.
3. **Default Save Logic**: Automatically save entities with options to skip or customize.
4. **Entity Management**: Easily set, get, and manipulate referenced entities.
5. **Equality Support**: Simplified comparison of entity references.

4. Patterns

Description: A utility class containing predefined regex patterns for common validation tasks.

Patterns:

ALPHABETICAL: Matches alphabetical characters and spaces from any language.

EMAIL: Matches standard email addresses.

PASSWORD_STRONG: Matches strong passwords with at least one uppercase letter, one lowercase letter, one digit, one special character, and a minimum length of 8.

(Additional patterns are available in the Patterns class.)

Field Change Listeners

- **Description:** Listen to field value changes and execute custom logic based on the new values.

Example:

```
human.name.onFieldChanged((oldValue, newValue) -> { if
(Objects.equals(newValue, "example"))
{ human.age.setFieldValue("100"); } });
```

5. Entity class

Description: The Entity class serves as the foundation for all entities in the **Quentity framework**. It provides utilities for managing persistence, validation, soft deletion, and more. The class leverages JPA, QueryDSL, and Hibernate features to create a robust and flexible architecture for entity management.

Key Features:

· **Generic Base Class**:

The Entity<T extends Entity> class can be extended by any specific entity type.

Provides a shared implementation for common entity behaviors.

· **Soft Deletion**:

Includes a deleted flag to manage entities without physically removing them from the database.

· **Persistence and Validation**:

- Integrated with JPA for persistence.

- Includes custom validation logic for fields.

· **Query Support**:

Supports QueryDSL-based filtering and query editing.

· **Field Management**:

Dynamically initializes and validates entity fields using reflection.

· **Extensibility**:

Developers can define additional behavior by overriding abstract methods like define.

· **Lifecycle Events**:

Implements @PostLoad to initialize services after the entity is loaded from the database.

How to use

1. Creating a Custom Entity

To define a new entity, extend the Entity class and implement the define method.

2. Persisting an Entity

You can save a custom entity using the save() method.

3. Using Soft Deletion

Soft deletion ensures that an entity is marked as deleted but not removed from the database.

Example:

user.delete();

Querying Non-Deleted Entities:

- The @Filter and @Where annotations automatically exclude deleted entities in queries.

4. Query Editing and Filtering

The class supports dynamic query modification using addQueryEditor and addQueryFilter.

5. Dynamic Field Validation

The save() method validates fields before persisting them.

- **Example**:
  - If a field is of type Fld, its validateValue method is automatically called.
  - For SingleEntityReference, referenced entities are saved recursively.

6. Creating a New Entity Instance

The newEntity method initializes a new entity instance with all its fields.

7. Lifecycle Management

The @PostLoad annotation ensures that the entityService is re-initialized after loading an entity from the database.

8. Detecting Changes

The isEntityEdited method checks if the current entity differs from the version in the database.

9. Using API-Specific Save Logic

The apiSave() method includes additional logic for handling field changes.

6. FLD Class

Description: The Fld class is an abstract generic class that represents a field within an entity. It provides functionalities for managing field properties, validation, and change tracking. The class extends Res<TYPE> which indecates that it got a resource (multilingualmultilingal label text) and implements Comparable for sorting purposes, as well as HasValue for value management.

class acts as a base field abstraction, encapsulating common behavior and properties for entity fields such as:

1. **Validation**: Enforces constraints on field values.
2. **Value Change Listeners**: Tracks changes to field values.
3. **Visibility and Editability**: Provides control over the field's visibility and modifiability.
4. **Labeling**: Dynamically updates field labels for better UI representation.

This makes it a **core component** for defining custom fields within the Quentity framework.

Attributes

Attribute	Type	Description
defaultValue	Object	The default value of the field.
required	boolean	Indicates whether the field is required.
visibleField	boolean	Determines if the field is visible in the UI.
editable	boolean	Specifies if the field can be edited.
hasLabel	HasLabel	UI label associated with the field.
fieldChangedCallback	FieldChanged<INNER_TYPE>	Callback function to handle value changes.

Key Components

1. Field Properties

1. The class defines key properties for managing fields:
2. defaultValue: A default value for the field.

3. required: Whether the field is mandatory.
4. visibleField: Controls the visibility of the field.
5. editable: Determines whether the field can be edited.

2. Field Labeling

- The hasLabel property, which implements HasLabel, allows developers to assign a label to a field for UI display.
- Labels can be updated dynamically with setFieldName.

3. Validation

· The abstract method validateValue enforces field-level validation rules.

· Developers must implement this method in subclasses to define how a field value should be validated.

4. Value Change Listener

- The onFieldChanged method enables developers to define custom logic that triggers whenever the field's value changes.

5. Comparison Logic

- Implements the Comparable interface, allowing fields to be compared based on their values.
- Uses the INNER_TYPE's compareTo method for comparison.

6. Value Change Listener Integration

- Provides an abstract method getValueChangeListener to handle value change events. Developers need to implement this in subclasses.

How to use

1. Create a New Field Type

To implement a custom field type, extend the class Fld and override the abstract methods.

2. Add the Field to an Entity

Create an instance of your field and use it inside your entity class configuration.

3. Validate the Field

Call validateValue method and have the value verified against set restrictions.

4. Add a Change Listener

Assign custom callback and control the changeable state of this field at any moment.

5. Customize Field Visibility and Editability

Hide-read-only certain view dynamically using bind conditionally field.

Summary

The Fld class provides a flexible and extensible basis for creating and manipulating the fields of an entity in general within the Quentity framework. This allows the developers to perform field-level validation, track changes in values, and manipulate field visibility, editability, and labeling.
Following the patterns and extending the Fld class, the developer creates robust, reusable fields for the application while maintaining clean and consistent design.

Any class whose name begins with Internal extends Res<TYPE>, and it is an abstract class - intended to be subclassed by an application to actually implement a particular persistence scheme.
It provides a framework to be extended and is intended to be flexible. The classes can include methods for dealing with **storable** and **nonstorable** objects and thereby provide distinction between the objects that can be persisted or stored and those that cannot.

7. InternalFldNumber class

Description: An abstract implementation for number fields that extends **Fld**.

It provides additional features such as suffixes, prefixes, and UI integration.

It encapsulates:
· **UI Integration**: Uses Vaadin's NumberField for numeric input in a web interface.
· **Validation**: Handles field-specific validation such as min/max values and required constraints.
· **User-Friendly Features**: Supports prefixes, suffixes, step increments, and more.

· **Dynamic Behavior**: Provides dynamic changes to the numeric field's properties (e.g., visibility, editability).

Key Components

1. Field Properties

The class extends Fld and adds specific properties for numeric fields:

- min **and** max: Minimum and maximum values.
- step: Increment value for stepping through numbers.
- prefix **and** suffix: Visual enhancements for the field.
- numericField: An instance of Vaadin's NumberField to handle numeric input in the UI.

2. Customizable Behavior

Setters for Field Properties

Methods like setStep, setMin, and setMax allow developers to configure field behavior dynamically:

Java
public InternalFldNumber setStep(Double step) { this.step = step; numericField.setStep(step); numericField.setStepButtonsVisible(step != 0); return this; }

3. Validation

The validateValue method enforces constraints, such as checking if the value is within the min/max range or if a required field is empty

4. UI Integration

Uses Vaadin's NumberField component for numeric input, providing seamless integration with the UI.

Dynamic Value Updates

Ensures the field's state in the UI reflects the underlying value.

5. Abstract Methods

The following methods must be implemented by subclasses to provide additional functionality:

- Double getNumericValue(): Retrieves the numeric value of the field.
- void setNumericValue(Double value): Updates the numeric value.

How to use

1. Define a Custom Numeric Field

Extend the InternalFldNumber class and implement the abstract methods.

2. Add the Field to an Entity

Integrate the custom field into an entity and configure its behavior.

3. Validate Field Values

Invoke the validateValue method to enforce field-specific rules.

4. Handle Value Changes

Attach a change listener to the field to respond to value updates dynamically.

5. Customize UI Appearance

Set prefixes, suffixes, and other UI properties for enhanced user experience.

Summary

The InternalFldNumber class provides a robust foundation for numeric field handling in the **Quentity framework**, combining:

- Validation
- Dynamic behavior
- UI integration with Vaadin

By extending this class and utilizing its features, developers can build powerful, reusable numeric fields for their applications while maintaining consistency and scalability.

8. InternalFldString class

Description: An implementation for string fields that extends Fld.

It supports additional constraints like masking, minimum, and maximum lengths.

Including:

· **Validation for values**.
· **Minimum and maximum length constraints**.

· **Pattern matching (Regex)**.
· **Control over visibility and editability**.
· **Seamless integration with the Vaadin UI framework**.

Key Features

1. Core Properties

The class provides essential properties to manage text fields effectively:

- textField: A Vaadin TextField used for UI representation.
- fieldName: A private field for storing the field's name.
- minLength: The minimum allowed length for the text.
- maxLength: The maximum allowed length for the text.
- mask: A regex pattern to validate the text input.

2. Initialization (Constructors)

The class provides both a default constructor and a parameterized one to set default values and constraints.

3. Value Validation

The class provides a validateValue method to check if the input is valid. The validation ensures:

1. The field is not empty if required.
2. The input matches the specified regex pattern.
3. The input length is within the defined range.

4. Managing Text Values

The class defines methods to manage the text value of the field:

- setFieldValue: Sets a new value for the field and invokes callbacks if the value changes.
- getFieldValue: Retrieves the current value of the field.
- getTextValue and setTextValue: Abstract methods to be implemented in subclasses to handle how the text value is retrieved and stored.

5. UI Integration

The class integrates seamlessly with Vaadin's UI components, allowing developers to configure and display text fields in a web application. For example:

- The textField is configured with constraints such as minimum/maximum length, patterns, and visibility.
- Developers can attach listeners to handle user input dynamically.

How to use

1. Creating a Subclass

To use this class, developers can create a concrete subclass and implement the abstract methods getTextValue and setTextValue.

2. Using the Field in a Model

Developers can use the UsernameField (or similar subclasses) to define text-based fields in their models.

3. Integrating with the UI

The class's textField can be directly added to a Vaadin view to allow user input.

Summary

The InternalFldString class provides a powerful and reusable solution for handling text fields in the **Quentity** framework. Its key features include:

1. **Validation**: Ensures inputs meet specific constraints (e.g., length, pattern).
2. **Customization**: Developers can create subclasses to suit specific use cases.
3. **UI Integration**: Built-in compatibility with Vaadin for creating interactive user interfaces.
4. **Reusability**: Simplifies the management of text fields across multiple forms and models.

9. InternalFldBool Class

Description: The InternalFldBool class represents a **boolean field** . It is designed to work with Vaadin's Checkbox component for boolean (true/false) inputs and ensures seamless UI integration, validation, and customization. It is an abstract class that provides the foundation for reusable boolean fields with configurable properties.

Key Components

1. Field Properties

This class extends Fld and adds properties specific to boolean fields:

- checkBox: A Vaadin Checkbox component for capturing boolean input.

- fieldName: An identifier for the field, with AccessLevel.NONE to prevent external modification.

2. Constructors for Initialization

Default Constructor:

Creates a boolean field with default configurations:

```
java
public InternalFldBool()
   { this(null, false, false, true, true);
}
```

Parameterized Constructor:

Allows full customization of the field's behavior, including:

- Default value
- Whether the field is required
- Visibility
- Editability

3. Validation

The validateValue method ensures the boolean field meets specific constraints:

- If the field is marked as required, it must have a value

4. Abstract Methods

The class defines abstract methods to be implemented by subclasses:

- Boolean getBoolValue(): Retrieves the field's value.
- void setBoolValue(Boolean value): Sets the field's value.

5. UI Integration

The checkBox component is synchronized with the field's configuration, ensuring:

- Visibility reflects the visible flag.
- Editability reflects the editable flag.
- Required fields show a required indicator.

Handling Value Changes:

The getValueChangeListener method listens for UI changes and ensures the field remains valid.

6. Field Behavior

Generate and Present Values:

The generateModelValue and setPresentationValue methods ensure the field's value is synchronized between the UI and backend model

7. Error Handling

The isEmptyValue method is a utility to check if the field's value is empty or invalid.

How to use

1. Create a Custom Boolean Field

Developers can extend InternalFldBool to define a reusable boolean field.

2. Use the Field in an Entity

Developers can integrate the custom field in entities with configurable behavior.

4. Validate Values

4. Respond to Changes

Developers can attach a listener to react to changes in the field's value.

5. Customize UI Appearance

Summary

The InternalFldBool class is a flexible and reusable base for boolean fields in the **Quentity framework**, offering:

1. **Integration with Vaadin** Checkbox for seamless UI interaction.
2. **Validation** to ensure values adhere to business rules.
3. **Customizability** for visibility, editability, and behavior.
4. **Synchronization** between backend data and frontend UI.

10. InternalFldDate Class

Description: The InternalFldDate class is an abstract base class for managing date fields in a data-driven application. It integrates with Vaadin's DatePicker component to provide a flexible, UI-driven approach to handle date inputs with validation, formatting, and customization options.

Key Components

1. Field Properties

This class extends Fld and adds specific properties for date fields, including:

- datePicker: A Vaadin DatePicker for UI date input.
- minValue **and** maxValue: Define the acceptable range for dates.
- mask: A string to specify the expected date format.

2. Constructors for Initialization

The constructors allow developers to initialize the InternalFldDate with default values and configurations.

3. Validation

The validateValue method ensures that the selected date adheres to the required constraints:

- Checks if the date is required but missing.
- Ensures the date falls within the specified min and max range.

4. UI Integration

The DatePicker component is configured to reflect the field's behavior in the UI:

- **Dynamic Updates**: The field's value, visibility, and editability are synchronized with the UI.
- **Internationalization**: Supports multiple date formats for user input

5. Abstract Methods

The class defines abstract methods that must be implemented by subclasses:

- void setDateValue(LocalDate value): Sets the field's date value.
- LocalDate getDateValue(): Retrieves the field's current date value.

How to use

1. Create a Custom Date Field

To use this class, a developer needs to extend InternalFldDate and implement the abstract methods.

2. Configure the Field in an Entity

The custom field can then be used in entities, with specific configurations for validation and behavior.

3. Validate Field Values

Ensure the field's value adheres to the constraints using validateValue.

4. Respond to Value Changes

Attach a listener to handle changes to the field's value dynamically.

5. Customize the UI Appearance

Enhance the field's appearance and usability using prefixes, suffixes, or additional styles.

Summary

The InternalFldDate class is a powerful tool for handling date fields in the **Quentity framework**, combining:

- **Validation**: Ensures date values meet specific constraints.
- **UI Synchronization**: Provides seamless integration with Vaadin's DatePicker.
- **Customization**: Supports dynamic configuration of properties like min/max dates, formats, and required fields.

11. InternalMultiEntitiesReferences Class

Description: The class is an abstract implementation of a custom field that manages multiple entity references in a grid format. It integrates with Vaadin Flow components and provides a flexible framework for handling and displaying relational data within a web application.

it provides:

1. **Dynamic Reflection**: Automatically loads entity fields and generates a dynamic UI grid for displaying and managing them.
2. **Drag-and-Drop Support**: Allows reordering of entities in the grid using drag-and-drop.
3. **Validation and Callbacks**: Includes features like required field checks, field validation, and custom change handling.
4. **Vaadin Integration**: Leverages Vaadin components to create modern, interactive user interfaces.

Key Features

1. Reflect Entities Dynamically

The reflect method dynamically loads the fields of a given entity class and displays them in a grid.

Outcome:

- A dynamic grid is generated for MySubEntity.
- Columns for each field are automatically created and configured.
- Data binding and validation are handled automatically.

2. Add and Remove Items

The class allows developers to add or remove references to the entities through a grid and buttons.

3. Drag-and-Drop Support

The grid allows reordering of items via drag-and-drop. Items can be moved within the list or dropped onto other target items.

- **Custom Logic for Reordering**: The moveItem method defines how to reorder or rearrange items based on the drop location (BELOW or ABOVE).

4. Validation and Field Changes

Developers can define validation logic and handle field change events using callbacks.

Validation: Implement the validateValue method to add custom validation logic.

Field Change Listener:

```
java
references.onFieldChanged((oldValue, newValue) ->
   { System.out.println("Field changed from " + oldValue + " to " + newValue);
});
```

5. Vaadin UI Integration

This class works seamlessly with Vaadin components to create user-friendly interfaces.

Grid Configuration:

- Developers can customize the grid's behavior, appearance, and interaction using built-in methods.

Add to List Button:

Use the addToListButton to add entities dynamically.

6. Context Menu for Actions

The GridContextMenu allows developers to attach context menu options (e.g., edit, save).

7. Abstract Methods to Implement

Developers need to implement several abstract methods to tailor the class to their needs:

Key Components

1. **Grid**:
    1. Displays the list of entities.
    2. Configured dynamically based on the entity class.
    3. Supports drag-and-drop and context menus.

1. **Add Button (**addToListButton**)**:
    1. Adds a new entity to the list.
2. **Validation**:
    1. Ensures the list of entities meets business requirements.
3. **Reflection** :
    1. Automatically generates the UI and field configurations.

How to use

Step 1: Create a Subclass

Create a concrete subclass of InternalMultiEntitiesReferences for the specific entity type.

Step 2: Use the Subclass

Use the subclass in your application to manage multiple entity references.

Summary

The InternalMultiEntitiesReferences class is a powerful, reusable tool for managing multiple entity references in the Quentity framework. With its rich features like dynamic reflection, drag-and-drop support, and Vaadin integration, it simplifies the development of complex UIs while enforcing data consistency and validation.

12. InternalSingleEntityReference

Description: This document breaks down the InternalSingleEntityReference abstract class, explaining how developers can use, extend, and learn from it.

Purpose of InternalSingleEntityReference

The InternalSingleEntityReference is an abstract class designed to:

- Provide a single entity selection reference in a UI.
- Manage entity data dynamically using a ComboBox.
- Integrate with Quentity services for querying and handling entity objects.

This class enables developers to easily implement dropdown-like selection components with custom logic, validation, and presentation.

Key Features

Core Functionality

- Implements a ComboBox to select entities of type T (extending Entity).
- Supports filtering and pagination through Quentity services.
- Provides a "plus" button (Button) to add new entities.

Customizable Behavior

- Fields like required, visibleField, and editable control field behavior.
- Developers can override abstract methods like setEntity, getEntity, and onSave to implement specific logic.

Validation

- The validateValue method ensures constraints (e.g., required values) are enforced.
- Invalid inputs trigger error messages in the ComboBox UI.

Dynamic Rendering

- Uses ComponentRenderer to render items in the ComboBox with rich details.

Field Change Listener

- A FieldChanged callback notifies when the field value changes.

Reflection-Based Utilities

- Dynamically retrieves and displays entity fields using reflection (EntityFieldsFactory).

Integration with Quentity Services

- Uses ServiceFactory for entity-related operations like filtering and searching.
- Supports additional filters through addedFilters.

How to Use

1. Extend the Class

Since InternalSingleEntityReference is abstract, you must extend it and implement its abstract methods.

2. Add to a UI

You can use your new reference class in a Vaadin-based UI.

3. Filter and Customize the ComboBox

You can dynamically set filters for the ComboBox

Summary

The InternalSingleEntityReference is a powerful, flexible abstract class designed for entity selection and management within the Quentity framework. It demonstrates:

- How to integrate frontend components with backend services.
- Practical usage of reflection for dynamic rendering.
- Real-time validation and UI feedback.

13. MultiEntitiesReferences class

Description: Manages references to multiple entity objects in a many-to-many relationship. This class ensures efficient handling and saving of these references while avoiding cyclic dependencies.

Key Features

Entity Management:

- Allows managing multiple references to entities (List<T>).
- Uses JPA annotations to persist the relationships in a database.

Validation:

- Provides validation mechanisms for referenced entities.

Lifecycle Integration:

- Ensures that referenced entities are saved when the parent entity is saved.

Data Binding:

- Supports integration with Vaadin's ListDataProvider for UI-related data management.

Extensibility:

- Inherits from InternalMultiEntitiesReferences, making it a specialized implementation.

How It Works

JPA Annotations

@ManyToMany(fetch = FetchType.EAGER):

- Establishes a many-to-many relationship between the parent entity and its references.
- Fetches the related entities eagerly by default.

@OrderColumn:

- Maintains the order of the entities in the list.

@Embeddable:

- Marks this class as embeddable, meaning it can be embedded into other entity classes.

Summary

The MultiEntitiesReferences class is a powerful tool for managing relationships between multiple entities in the Quentity framework. Here's what developers need to know:

Features:

1. Manages many-to-many relationships.
2. Validates and saves referenced entities automatically.
3. Integrates with Vaadin for UI-based data management.

Usage:

1. Embed it in custom entity classes to manage lists of related entities.
2. Use methods like setEntities and onSave to handle data dynamically.

Best Practices:

1. Always validate data using validateValue() before saving.
2. Use onSave() to ensure referenced entities are saved correctly.

14. NSMultiEntitiesReferences

Description: The NSMultiEntitiesReferences class provides a robust way to manage relationships with multiple entities in the Quentity framework. It is especially useful for:

1. Managing references to multiple entities.
2. Handling operations such as saving, validation, and updating relationships.
3. Providing advanced data binding and lifecycle hooks like callbacks during saving.

Key Features

1. Entity Management

- **Set and Get Entities**:
  - Manage multiple entities using ListDataProvider<T> for efficient data handling.
  - Methods like setEntity and setEntities allow developers to update the list of referenced entities dynamically.

2. Data Context and Callbacks

Context Data:

- The contextData map allows developers to attach additional metadata or context to the references.
- Developers can use setEntitiesWithContext to set entities along with custom context data.

On-Save Callback:

- A Consumer<Map<String, Object>> callback (onSaveCallback) can be executed during the save operation, allowing developers to add custom logic.

3. Validation

- The validateValue method enforces that the field is required if marked as such and throws an IllegalArgumentException if validation fails.

4. Saving Behavior

- The onSave method performs the following:
  - Validates the entities before saving.
  - Executes the onSaveCallback if defined.

- Handles cyclic references to prevent saving the same entity multiple times.
- Saves each entity by calling the save() method on them.

5. Equality and Hashing

- The equals and hashCode methods are overridden to compare entity lists and ensure proper behavior in collections or when performing equality checks.

6. Extendability

- Developers can extend the class to add further functionality or override methods for specific use cases.

How to Use

1. Creating an Instance

To initialize an NSMultiEntitiesReferences instance:

```
java
NSMultiEntitiesReferences<MyEntity> references = new
NSMultiEntitiesReferences<>();
```

2. Setting Entities

You can set entities either directly as a list or using a ListDataProvider:

```
java
// Setting entities using a list
List<MyEntity> myEntities = fetchEntities(); // Fetch or create entities
references.setEntities(myEntities);
// Setting entities using a ListDataProvider
ListDataProvider<MyEntity> dataProvider = new
ListDataProvider<>(myEntities);
references.setEntity(dataProvider);
```

3. Adding Context Data

Attach additional context to the references for use in callbacks or other operations:

```
java
CopyEdit
references.setEntitiesWithContext(dataProvider, context ->
   { context.put("source", "UserImport");
   context.put("timestamp", System.currentTimeMillis());
});
```

4. Defining an On-Save Callback

Add a custom callback to execute logic during the save operation:

```
java
CopyEdit
references.setOnSaveCallback(context  ->
   { System.out.println("Saving with context: " + context);
   // Add custom logic here
});
```

5. Saving Entities

Call onSave to trigger the save process for all entities:

```
java
CopyEdit
references.onSave();
```

This method will:

- Validate the entities.
- Execute the onSaveCallback.
- Save each entity while avoiding cyclic references.

6. Validation

Ensure that the field is valid before saving:

```
java
CopyEdit
try {
   references.validateValue(references.getEntity());
} catch (IllegalArgumentException e)
   { System.err.println("Validation failed: " + e.getMessage());
}
```

7. Comparing References

Use equals and hashCode to compare two NSMultiEntitiesReferences objects or use them in hash-based collections:

```
java
NSMultiEntitiesReferences<MyEntity> references1 = new
NSMultiEntitiesReferences<>();
NSMultiEntitiesReferences<MyEntity> references2 = new
NSMultiEntitiesReferences<>();
if (references1.equals(references2))
   { System.out.println("Both references are equal.");
}
```

Summary

The NSMultiEntitiesReferences class provides a flexible, customizable, and efficient way to manage references to multiple entities in the Quentity framework. It offers developers powerful tools for validation, data binding, lifecycle management, and UI integration

15. ServiceFactory class

Description: The ServiceFactory provides services to manage entities and interact with databases or search engines.

- is a central utility in the Quentity framework, designed to simplify interaction with entity services and provide access to shared utilities like the ObjectMapper. It supports efficient caching and dynamic method invocation for entity-specific behavior.

Key Features

Entity Service Management:

- Dynamically retrieves and caches EntityService instances for entity classes.

Object Mapper Access:

- Provides a shared instance of ObjectMapper for JSON serialization and deserialization.

Dynamic Entity Method Invocation:

- Uses MethodHandles to invoke the define method dynamically for any entity class.

Spring Integration:

- Leverages Spring's ApplicationContext to access beans by name and type.

Thread-Safe Caching:

- Uses ConcurrentHashMap to store cached services and method handles.

How to Use

1. Retrieve an Entity Service

You can retrieve an EntityService for any entity class using either the class or its name.

2. Access the Shared ObjectMapper

Obtain the shared ObjectMapper instance for JSON

3. Dynamically Define an Entity

Invoke the define method on an entity dynamically without explicitly knowing its implementation

Summary

The ServiceFactory class simplifies the Quentity framework's usage by centralizing service management, caching, and dynamic method invocation. It ensures efficient and consistent access to EntityService instances and shared utilities like ObjectMapper.

Handling Validation and Errors

- **Description:** Use setInvalid() to mark a field as invalid and add an error message when incorrect values are entered.

Example:

try { validateValue(eValue); } catch (IllegalArgumentException ex)
{ textField.setInvalid(true); textField.setErrorMessage(ex.getMessage()); }

14. EntityService<E> Class

Description:A generic service class for managing entities in the application.

-is a generic service layer designed to provide developers with common functionality for managing entities in the **Quentity framework**. Below, you'll find everything a developer needs to understand, extend, and use the EntityService class effectively.

Key Features:

- CRUD operations:Perform basic operations such as create, read, update, and delete.
- **Search and Pagination**:Provides full-text search and pagination capabilities using **Hibernate Search**.

- **Relationship Management**:Dynamically manages relationships between entities, including clearing references on delete.
- **Dynamic Queries**:Supports creating advanced, filterable queries using QueryDSL.
- **Custom Query Support**:Dynamically builds and executes native queries.

Methods:

1. CRUD Methods
2. Pagination and Sorting
3. Full-Text Search
4. Relationship Management
5. Advanced Query Building

How to Use

1. Create Your Custom Service

To use EntityService, create a custom service by extending it for your entity

2. Perform CRUD Operations

3. Search Entities
4. Handle Relationships

Summary

The EntityService class in the Quentity framework is a powerful utility for managing entities with minimal boilerplate code. It combines the strengths of multiple technologies like Spring, Hibernate, QueryDSL, and Vaadin to provide a robust and flexible service layer.

14. AppLayout class

Description: This class is part of the Vaadin framework and provides a layout component for creating app-like user interfaces. It allows you to manage primary and secondary navigation areas, content display, and layout behaviors.

Overview

The code provided includes a class named AppLayout (from Vaadin framework) and focuses on creating and managing layouts in Vaadin applications, such as Navbar and Drawer. Additionally, there's reference to an EntityService (from an earlier snippet) that handles database operations using **Hibernate** and **QueryDSL**. Below is a detailed breakdown of what developers can learn from this and how they can utilize the framework effectively.

15. MainLayout Class

Description: The MainLayout class extends Vaadin's AppLayout to provide a customizable, top-level layout for a Vaadin-based application. It manages the application's navigation structure, header, footer, and dynamic content areas.

Key Components

1. Main Purpose

The MainLayout class acts as the central layout for the application. It combines:

- A **navigation drawer** for side menu items.
- A **header bar** for global actions like language selection and user account management.
- Tabs and content management using Vaadin's TabSheet for dynamic content display.

2. Core Features

A. Layout Management

Drawer and Navbar:

- Add navigational items to the drawer (createNavigation) and set up the header using addHeaderContent.
- Drawer behavior adjusts automatically based on screen width (handleScreenWidth).

Dynamic Tab Management:

- Tabs are dynamically updated (setContent) to reflect the current view or selected entity.
- Each tab is associated with a specific view/component.

B. Navigation

Side Navigation:

- Automatically populates navigation items for all subclasses of Entity using reflection (addEntitySubclassesToNav).

- Ensures access control using AccessAnnotationChecker.

Custom Navigation Items:

- Navigation items link to specific entity views or actions using CustomSideNavItem.

C. Security

Authenticated User Management:

- The layout checks the logged-in user (AuthenticatedUser) to manage:
  - User-specific settings (e.g., language preference).
  - Roles and access control (e.g., only admins see certain entities).

Logout Functionality:

- Users can sign out using a menu item in the footer.

D. Localization

- **Language Selector:**
  - The application supports multilingual capabilities via LanguageUtil.
  - Language is dynamically set based on the authenticated user's preferences.

E. Entity-Based Reflection

- **Dynamic Entity Loading:**
  - The system scans for all subclasses of Entity using Reflector.getEntities.
  - Dynamically creates navigation items for these entities, enabling seamless integration of new models.

F. UI Responsiveness

- **Screen Size Handling:**
  - Adjusts between Drawer and Navbar modes based on screen width for better usability on different devices.

3. How to Use

A. Steps to Build an Application Using Quentity

Define Your Entities:

- Create subclasses of Entity for the data you want to manage.
- Annotate these with appropriate Quentity annotations.

Example:

```
java
CopyEdit
@Entitypublic class Product extends Entity
   { private String name;
   private double price;
}
```

Create Views for Each Entity:

- Define views for entities using MainEntityViewHandler.
- Use LanguageUtil for localized labels.

Add Custom Navigation Items (Optional):

- If needed, add custom navigation or actions by extending CustomSideNavItem.

Example:

```
java
nav.add(new CustomSideNavItem("Custom Action",
VaadinIcon.EDIT.create(), event -> {
   // Custom logic here
}));
```

Manage Security:

- Use AccessAnnotationChecker to define roles and permissions for views or entities.

Deploy and Test:

- Build and run the application to ensure all entities and navigation are displayed correctly.

Key Methods Developers Should Understand

A. addEntitySubclassesToNav

- **Purpose:** Dynamically adds navigation items for all accessible Entity subclasses.
- **How It Works:**

- Scans for Entity subclasses using Reflector.
- Checks user access using AccessAnnotationChecker.
- Creates navigation items with icons (IconHandler) and localized names (LanguageUtil).

B. setContent

- **Purpose:** Dynamically updates the main content area with either a new view or a tab.
- **How It Works:**
  - If the content is an instance of Main, it sets it directly.
  - Otherwise, it adds a new tab for the component.

C. addHeaderContent

- **Purpose:** Sets up the application's header with a drawer toggle, title, and language selector.

D. addDrawerContent

- **Purpose:** Adds search functionality and navigation items to the drawer.

16. Component Class

Aspect	Details
Fields	- **idDescriptor**: Descriptor for the optional ID attribute. - **element**: Encapsulated DOM element. - **eventBus**: Manages event listeners.
Constructors	- **Component()**: Creates a component using the @Tag annotation. - **Component(Element element)**: Maps the component to an existing element.
Core Methods	- **getElement()**: Returns the DOM element. - **getParent():** Retrieves the parent component. - **setId(String id)**: Sets the component's ID.
Event Handling	-**addListener(Class, Listener)**: Registers event listeners. - **fireEvent(ComponentEvent)**: Triggers a component event.
Localization	- **getTranslation(String key)**: Retrieves localized text. - **getLocale()**: Returns the current locale.
Utilities	- **scrollIntoView()**: Brings the component into view. - **removeFromParent()**: Detaches the component from its parent.
Nested Class	

17. AccessAnnotationChecker class

Description: The AccessAnnotationChecker class is a utility for checking access permissions based on Java security annotations (@DenyAll, @PermitAll, @RolesAllowed, etc.) in a Vaadin Flow application. It evaluates access to classes and methods using the annotations and the current user's role and authentication status.

Features:

- Determines access to methods or classes based on security annotations.
- Supports annotation-based security, including role checking.
- Can handle both authenticated and unauthenticated users.

Key Utility Methods:

static AnnotatedElement securityTarget(Class<?> cls)
Determines the class to evaluate for security annotations.

private boolean hasAccess(AnnotatedElement annotatedElement, Principal principal, Function<String, Boolean> roleChecker)
Main method for evaluating access based on annotations.

private static boolean hasSecurityAnnotation(AnnotatedElement method)
Checks if a method or class has security annotations.

19. AuthenticatedUser class

Description: The AuthenticatedUser class is a utility for managing the currently authenticated user in a Spring Security-based Vaadin Flow application. It integrates with the AuthenticationContext to provide access to user details and manage authentication state.

Features:

- Fetches the currently authenticated user.
- Updates user data in the repository.
- Provides functionality to log out the authenticated user.

Dependencies:

- UserRepository: For accessing and updating user data in the database.
- AuthenticationContext: For interacting with the Spring Security context.

Methods:

Optional<User> get()
Retrieves the currently authenticated user as an Optional.

- Maps the UserDetails from the authentication context to a User entity in the repository.

User update(User user)
Updates the user's information in the repository.

void logout()
Logs out the currently authenticated user by clearing the authentication context.

Annotations:

- @Component: Marks the class as a Spring component for dependency injection.
- @Transactional: Ensures that database operations are executed within a transactional context.

20. FldDate class

Description: The FldDate class is an embeddable field type used for managing date-based data. It extends InternalFldDate and provides additional functionality for validation and user interaction.

class is a reusable component for date fields, with advanced validation and customizable configurations.

It includes an overview of its functionality, the features it provides, and how developers can effectively use and extend it.

Features of the FldDate Class

1. Core Attributes

The class provides several customizable attributes:

- dateValue: The actual date value stored in the field.
- minValue and maxValue: Optional constraints on the valid range of dates.
- mask: A date formatting mask (if needed for display/input purposes).
- defaultValue: A default date value assigned when the field is initialized.
- required: Specifies if the field must have a value.
- unique: Indicates if the field's value must be unique.
- visible: Determines if the field is visible in the UI.
- editable: Indicates if the field can be edited.

2. Validation

- The validateValue method ensures the date value meets constraints like minValue and maxValue. If the value is invalid, an error message is shown in the associated date picker UI component.

3. Value Change Handling

- The getValueChangeListener method listens for changes to the field's value, validates the input, and updates the dateValue accordingly.
- It also provides callback functionality (fieldChangedCallback) to notify other parts of the system when the value changes.

4. Integration with UI Components

- The class works with a date picker component to provide a seamless user experience. It handles UI feedback like marking the field invalid and displaying error messages when necessary.

How to Use

1. Adding a Date Field to an Entity

To use the FldDate class, define it as a field in your entity class. Use the builder to configure its properties.

2. Customizing Field Behavior

You can customize the behavior of the date field by setting its attributes:

- **Required Field:** Set required(true) to ensure a value is always provided.
- **Date Constraints:** Use minValue and maxValue to restrict valid date ranges.
- **Default Value:** Set defaultValue(LocalDate.of(2025, 1, 1)) to provide a predefined value.

3. Handling Value Changes

If you want to perform additional actions when the date value changes, define a fieldChangedCallback

Summary

The FldDate class in the Quentity framework provides a powerful and flexible way to handle date fields in entity models. It supports:

- Validation and constraints.
- Integration with UI components for user-friendly interaction.
- Customizable behavior for specific application needs.

21. NSFldDate

Description: The NSFldDate class is a concrete implementation of the abstract InternalFldDate class, designed to manage and validate date fields in a structured and customizable manner. It is specifically tailored for applications that require date input with support for validation, constraints, and event handling.

It also describes how to create and use custom field types and provides insights into the class design for learning purposes.

Features of NSFldDate

Core Attributes

1. dateValue: The actual value of the date.
2. minValue and maxValue: Constraints to define the valid date range.
3. defaultValue: Default date value assigned to the field.
4. mask: Formatting mask for input/output display.
5. required: Indicates whether the field must have a value.
6. unique: Specifies if the value must be unique across entities.
7. visible: Determines if the field is shown in the UI.
8. editable: Specifies if the field is modifiable.

Validation

1. The validateValue method ensures the provided value respects minValue and maxValue.
2. Invalid input triggers an error message in the UI, and the field is marked as invalid.

Event Handling

1. A ValueChangeEvent listener is implemented via the getValueChangeListener method, which validates changes and updates the field.
2. The fieldChangedCallback notifies other system components when the field value changes.

UI Feedback

1. Integration with a date picker component allows real-time validation and error display.

Embeddable Field

1. The class is annotated with @Embeddable, making it suitable for embedding in JPA entities.

Transient Field

1. The dateValue is marked as @Transient, meaning it is not persisted in the database but can be used for runtime operations.

How to Use

1. Adding a Date Field to an Entity

The NSFldDate class can be embedded in an entity to handle date values.

2. Customizing Field Attributes

You can configure the field with constraints and behavior using the builder.

3. Handling Value Changes

Implement the fieldChangedCallback to perform custom logic when the field value changes.

4. Validation

The validateValue method automatically enforces constraints. For example:

- If the date is outside the allowed range, an exception is thrown, and the UI reflects the error.

Summary

The NSFldDate class is a powerful and flexible date field implementation within the Quentity framework. It combines backend validation, user-friendly frontend integration, and extensibility. Whether you're building a project with Quentity or learning how to design reusable components, this class offers valuable patterns and ideas.

22. FldBool

Overview of FldBool

The FldBool class is a specialized implementation for handling **boolean values** within the Quentity framework. It extends InternalFldBool, inheriting base behavior for boolean field management, while allowing for further customization and integration.

Annotations:

- @Embeddable: Marks this class as embeddable, meaning it can be used as part of another JPA entity.
- @Accessors(chain = true): Enables method chaining for setters, making code more concise.
- @EqualsAndHashCode: Generates equals and hashCode methods to compare instances effectively.

Key Features

Boolean Value Management:

1. Manages a boolean field (textValue) using getter and setter methods.
2. Integrates seamlessly with entities and the framework for handling boolean states.

Customizable Behavior:

1. The getBoolValue() and setBoolValue(Boolean value) methods can be overridden to extend or modify behavior.

Framework Integration:

1. Embeddable in JPA entities to represent boolean fields.
2. Usable directly in business logic or integrated with UI components.

Method Chaining:

1. Provides a fluent API for setting values, improving code readability and efficiency.

How to Use

1. Embedding in JPA Entities

The FldBool class can be used as part of a JPA entity to represent boolean fields.

Explanation:

- The isActive field uses FldBool to handle its boolean state.
- The FldBool instance encapsulates all behavior related to boolean value management.

2. Managing Boolean States

Developers can use the getter and setter methods to retrieve or update the boolean value.

3. Extending Behavior

If you need additional functionality, extend FldBool and override its methods to implement custom behavior.

4. UI Integration

You can integrate FldBool into UI components, such as checkboxes in a Vaadin application.

Explanation:

- The FldBool instance (isActive) holds the boolean state.
- The Checkbox component updates the state dynamically.

· **Core Purpose**: FldBool is designed for managing boolean fields with minimal effort while maintaining flexibility.
· **Integration**: It works seamlessly with the Quentity framework, JPA entities, and Vaadin components.
· **Customization**: Developers can extend the class to add validations or modify behavior as required.
· **Method Chaining**: Enhances readability and simplifies setting multiple values in sequence.

23. NSFldBool

Description:the NSFldBool class in the **Quentity framework**, how it works, and how to build and extend functionality using the framework. The document also provides insights for anyone reading the code to understand it and learn from it.

Key Features:

1. **Embeddable**:
    1. Marked with @Embeddable, making it usable as part of larger JPA entities.
2. **Transient Value**:

    1. The @Transient annotation indicates that the textValue field is not persisted directly in the database. Instead, it can be managed dynamically during runtime.

3. **Method Chaining**:

1. The @Accessors(chain = true) annotation allows for fluent API usage, improving code readability.

Equals and HashCode:

1. The @EqualsAndHashCode annotation generates equals() and hashCode() implementations automatically for better comparison and hashing.

How to use

1. Embedding in JPA Entities

You can embed the NSFldBool class into JPA entities to represent boolean fields.

2. Managing Boolean State

You can set and retrieve the boolean value using the getBoolValue() and setBoolValue(Boolean value) methods.

3. Extending or Customizing Behavior

If you need more functionality, you can create a custom class extending NSFldBool and override its methods.

4. Integration with UI Components

NSFldBool can be integrated into UI frameworks like Vaadin to manage boolean states dynamically in the interface.

5. Handling Non-Persistent States

Since the textValue field is marked as @Transient, it can be used for temporary computations or states that don't require database persistence.

Annotations:

1. @Accessors(chain = true): Enables method chaining for setters.
2. @Embeddable: Indicates this class can be embedded in JPA entities.
3. @EqualsAndHashCode: Provides default implementations for equals() and hashCode().

Fields:

1. @Transient private Boolean textValue: Represents the boolean value but is not persisted in the database.

Methods:

1. getBoolValue(): Returns the current boolean value.
2. setBoolValue(Boolean value): Sets the boolean value.

Purpose:

1. NSFldBool is designed for managing boolean fields efficiently within the Quentity framework.

Features:

1. Transient fields for non-persistent states.
2. Embeddable in JPA entities.
3. Easy-to-extend functionality.

Use Cases:

1. Representing boolean fields in entities.
2. Integrating with UI components.
3. Managing temporary states.

Summary:
the NSFldBool class, developers can effectively manage boolean fields in their applications while leveraging the flexibility and power of the Quentity framework.

24. FldString

Description: The FldString class is a special text field for use in entities. It manages text values, including validation logic, database constraints, and integration with full-text search systems. This class will provide seamless interaction between the front-end and back-end while maintaining data integrity and user experience.

Key Features:

Validation Mechanisms:

1. Integrity of data is ensured by the validation of input through the following constraints:
2. **Mask:** Regex-based both at the front-end and back-end. The Patterns class provides out-of-the-box regex patterns.

3. **MinLength:** Minimum character length enforced at the time of input and also during validation at front-end and back-end.
4. **MaxLength:** Maximum character length limits not only the input but also enforces database constraints.

Real-Time User Feedback:

1. Immediate feedback during the user input is provided. In the case of some value being invalid, an error message appears, and the field is marked as invalid to let users correct input errors immediately.

Database and Search Integration:

1. The class integrates with the database via Hibernate and makes the text field searchable by allowing compatibility with full-text search systems through the @IndexedEmbedded annotation.

Customizability:

1. Other configurations supported are required, unique, editable, and visible for various use cases.

User Interaction:

1. The user types or changes the text in the field, either via a form or programmatically.

Validation Workflow:

1. On focus change or during the validation of the field:
2. The regex mask checks whether the value matches the pattern required.
3. The minLength and maxLength constraints check the length of the input.

Error Handling:

1. If the input fails validation, the system provides immediate feedback (e.g., error messages), ensuring the field cannot proceed with invalid data.

Storage :

1. Valid data is saved to the textValue property thus to database, ensuring it is updated in the UI and model.

Search Capability:

Text values entered in the field are indexed to allow full-text search capability for efficient querying.

EXAMPLE:
FldString name = new FldString();
name.setMaxLength(30).setMask(Patterns.ALPHABETICAL);

Summary:

The FldString class fills the gap in user input, back-end validation, and database integration for ease of the user without even a single glitch. It is a very robust solution to managing text fields within entity-based systems because of real-time validation and feedback, with full compatibility with full-text search systems.

25. NSFldString

Description: The NSFldString class is a concrete implementation of the abstract InternalFldString class for managing string-based fields in a structured and data-driven application. This class includes extended functionality for validation, event handling, and customization of string inputs.

The NSFldString class represents a custom field type for managing string data in the Quentity framework. It is a subclass of InternalFldString and is designed for:

1. Handling string input with built-in validation and customization.
2. Providing flexible configuration for UI components like text fields.
3. Supporting advanced field properties such as validation, masking, visibility, and editability.

Key Features

1. Field Configuration

Field Properties:

- required: Specifies if the field is mandatory.
- unique: Ensures the value is unique (if applicable).
- visible: Controls whether the field is visible in the UI.
- editable: Defines if the field is editable by the user.
- minLength & maxLength: Specifies the length constraints for the string value.

- mask: Allows for applying input formatting or masking.
- defaultValue: Defines a default value for the field.

Builder Constructor: Developers can initialize the field with all these properties using the @Builder constructor.

2. Validation

The validateValue method (inherited from InternalFldString) is used to:

- Enforce length constraints (minLength, maxLength).
- Apply masks or other custom validation logic.
- Throw an IllegalArgumentException if validation fails.

3. Real-Time Value Updates

- The getValueChangeListener method listens for changes in the field's value, validates it, and updates the model.
- If validation fails, the field is marked as invalid, and an error message is displayed.

4. Data Binding

- The setModelValue and setPresentationValue methods synchronize the field's backend value with its frontend representation.

5. Customization and Extendability

Developers can override methods like setTextValue or getTextValue for custom behaviors.

How to Use

1. Create an Instance

You can create an instance of NSFldString and configure its properties using the builder or default constructor:

```
java
NSFldString nameField = NSFldString.builder()
   .value("Default Name")
   .minLength(3)
   .maxLength(50)
   .required(true)
   .unique(true)
   .visible(true)
   .editable(true)
   .build();
```

2. Integrate with UI

Add the field to a UI layout:

java
VerticalLayout layout = new VerticalLayout();
layout.add(nameField.getComponent()); // Assuming `getComponent` returns the field's Vaadin component.

3. Field Change Listener

To listen for value changes and react to them:

java
nameField.setFieldChangedCallback((oldValue, newValue) ->
    { System.out.println("Value changed from: " + oldValue + " to: " +
    newValue);
});

4. Validation

Ensure proper handling of invalid input:

java
try {
    nameField.setTextValue("Short"); // This will throw an exception if minLength > 5
} catch (IllegalArgumentException e)
    { System.err.println("Invalid value: " + e.getMessage());
}

Summary

The NSFldString class is a powerful tool in the Quentity framework, enabling developers to manage text fields with built-in validation, dynamic properties, and seamless data binding. By leveraging its features, developers can create robust and user-friendly applications

26. NSFldNumber Class

Description:NSFldNumber class, how it integrates with the **Quentity framework**, and how developers can utilize it effectively to create numeric fields in their applications.

Overview of NSFldNumber

The NSFldNumber class extends the InternalFldNumber class, inheriting its functionality and specializing it for numeric fields. It is designed to handle **numeric values** and allows chaining setters using Lombok's @Accessors.

- **Annotations**:
  - @Embeddable: Marks the class as embeddable for use in JPA entities.
  - @Transient: Prevents certain fields from being persisted in the database (in this case, value).
  - @Accessors(chain = true): Allows method chaining for setter methods.
  - @EqualsAndHashCode: Generates equals and hashCode methods for comparing instances.

Key Features

1. **Numeric Value Management**:

   Provides getter and setter methods for a numeric field (value), making it easy to retrieve or update the field's value.

1. **Integration with the Quentity Framework**:

   Works seamlessly with other Quentity components to represent numeric fields as part of entities or UI elements.

2. **Chaining Support**:

   Developers can set multiple properties of NSFldNumber in a single statement using method chaining.

How to use

1. Define Numeric Fields in Entities

Developers can use NSFldNumber as part of their JPA entities to represent numeric fields.

2. Set and Retrieve Numeric Values

Use the provided methods to set or get numeric values in your application.

3. Customize Behavior by Extending NSFldNumber

If additional functionality is needed, you can extend NSFldNumber and override its methods.

4. Use in UI Components

In a **Vaadin** application, developers can integrate NSFldNumber with UI components for displaying and editing numeric values.

· **Core Purpose**: The NSFldNumber class is a simple yet flexible component for managing numeric values in the Quentity framework.
· **Integration**: It can be embedded in JPA entities or used directly in business logic and UI components.
· **Customization**: Developers can extend the class to add custom validation or behavior for numeric fields.

27. AuthenticationContext Class

Description: The AuthenticationContext class provides a utility layer for managing security and authentication contexts in a Vaadin Flow application integrated with Spring Security. It simplifies interactions with the Spring Security framework, such as retrieving user details, managing roles and authorities, and handling logout functionality.

Purpose:

The AuthenticationContext class handles credentials and authentication data required to access repositories or proxies securely in a Maven-based build system.

Initialization:

1. forRepository: Creates an authentication context for a repository. This function is used when the user wants to authenticate access to a remote repository.
2. forProxy: Creates an authentication context for a proxy. This is used for accessing resources via an intermediate proxy server.

User Actions:

Retrieve Data (get method): The user can retrieve stored authentication data (e.g., username, password, or custom authentication keys) by providing the key name.

Example:

tring username =authContext.get(AuthenticationContext.USERNAME);

Store Data (put **method)**: The user can manually add or update authentication data to the context.
Example:

authContext.put(AuthenticationContext.PASSWORD, "your-secure-password");

Automatic Handling:

1. When authentication data is not already cached, the system dynamically invokes the fill method of the Authentication object to populate the required fields.
2. Sensitive data (e.g., passwords) are securely managed and cleared from memory when the context is closed.

Secure Closure:

authContext.close()

Helper Functions:

1. The convert method internally ensures type safety when retrieving stored authentication data, such as converting File paths to String or vice versa.

Benefits:

Dynamic Management: Automatically retrieves or caches authentication data when needed.

Secure Data Handling: Sensitive data like passwords are securely managed in memory and erased when no longer required.

Extensibility: Users can customize or extend the authentication mechanism by providing their implementations of the Authentication interface.

Error Prevention: Strict validation for key names and proper type handling ensures consistent and error-free usage.

4.3 Implementation

4.3.1 ResGenPlug

Description: This code defines a **Maven plugin** that scans Java source files for classes inheriting from specific types (like Entity or Res). It extracts their

field names, types, and relationships, then updates .properties files with this information.

Key Features:

1. **Scans Classes**: Identifies classes that extend Entity or similar.
2. **Extracts Fields**: Collects field names, types, and generic details.
3. **Updates Configuration**: Adds discovered classes and fields to .properties files for external use.
4. **Integrated with Maven**: Runs during the generate-resources phase of the Maven lifecycle.

Purpose:

This code is a **custom Maven plugin** that:

1. Scans Java source files for classes inheriting from specific superclasses or interfaces (Entity, Res, etc.).
2. Extracts details about fields in those classes, including generics and relationships.
3. Updates property files (.properties) in the resourcesDirectory with the extracted class and field data for use in external configurations.

How to Use

1. Add to Your Maven Project

2. Customize Configuration

By default, the plugin:

- Looks for Java files in src/main/java.
- Updates .properties files in src/main/resources/strings

3. Run the Plugin

Execute the Maven command:

mvn generate-resources

How work:

1. Class Hierarchy Traversal

The framework identifies relationships between classes using a ClassVisitor:

- Tracks which classes extend or implement specific types (Entity, Res, etc.).
- Stores these relationships in a classHierarchy map.

2. Field Extraction

For every identified class:

- Fields are analyzed to check if they belong to specific types or contain generics.
- Field names, generic types, and relationships are logged and stored.

3. Properties File Updates

For each discovered class and field:

- Updates a .properties file in the resources directory.
- Adds:
    - The fully qualified class name.
    - The package name.
    - Field names (e.g., com.example.Entity.someField).

This framework teaches several programming and architectural concepts:

A. Building a Custom Maven Plugin

Annotations (@Mojo**,** @Parameter**)**:

1. Use @Mojo to define the plugin's name and execution phase.
2. Use @Parameter to inject configurable parameters from pom.xml.

Integration with Maven Lifecycle:

1. Hooks into the generate-resources phase to generate resources dynamically.

B. Source Code Analysis with JavaParser

Parsing Java Files:

1. Uses SourceRoot to load and parse all source files in a directory.

Visiting Classes and Fields:

1. Implements VoidVisitorAdapter to traverse and process the AST.

Working with Generics:

1. Extracts generic types for fields (e.g., List<String>).

C. Properties File Management

1. **Reading and Writing Properties**:
    1. Uses java.util.Properties to load and save key-value pairs in .properties files.
2. **Dynamic Updates**:

    1. Adds class and field data to existing property files during execution.

4.3.2 RefGenPlug

Description: This code demonstrates how to build a framework based on a **Java Maven Plugin** to scan classes and objects in a Java project, process the relevant data of classes that inherit from a superclass (like Entity or Res), and generate reflection files in JSON format.

1. the purpose

- Scans all Java files in a specified directory.
- Detects classes that extend a base class (like Entity or Res).
- Extracts fields and their generics (if any) for each class.
- Generates JSON files (reflection files) representing these entities for use in further processes.

2. How does work

Configuration:

- The source directory for Java files is ${project.basedir}/src/main/java by default.
- The reflection JSON files are saved in ${project.basedir}/src/main/resources/reflection by default.

Execution:

- The plugin runs during the GENERATE_RESOURCES phase of the Maven lifecycle.
- It uses the JavaParser library to analyze source code and extract class and field information.

What we Learn:

Using the JavaParser library:

- How to parse source code files and extract class structures, fields, and generics.
- How to use visitors (like ClassVisitor) to traverse and process Java ASTs (Abstract Syntax Trees).

Generating structured output:

- Representing extracted data in plain old Java objects (POJOs) such as EntityPojo and FieldPojo.
- Writing structured data (like JSON) for further processing.

Working with Maven Plugins:

- Creating a custom Maven plugin and integrating it with Maven's lifecycle.
- Using annotations like @Mojo and @Parameter to define plugin behavior and configuration.

3. How to use

Integrate with your Maven project:

1. Add this plugin to your pom.xml.

Customize configuration if needed:

1. Specify the source directory with sourceDirectory.
2. Specify the output directory for JSON files with resourcesDirectory.

Run the plugin:

mvn generate-resources

Key components:

ClassVisitor:

1. Traverses all classes in the source code to build a class hierarchy (inheritance tree).

Entity Processing:

1. Identifies classes that extend the Entity superclass.
2. Extracts fields and their generics for each class and stores them in an EntityPojo.

Reflection File Writing:

1. Writes the extracted data into JSON files.

Customizable elements:

- The base classes (Entity, Res).
- Input and output directory paths.

Benefits:
· Automates the process of analyzing source code.
· Reduces manual errors when creating reflection files.
· Improves scalability by allowing easy addition of new classes and structures.

4.3.3 RepoServPlug

Description: This class is that automates the generation of repository, service, and constructor code for entity classes extending a base entity class in a Quentity-based framework. Below is a detailed breakdown of what developers need to know about this code and how to use it.

You need to know:

Purpose of the Plugin

- Automates boilerplate code generation for entity management in a **Quentity-based framework**.
- Reduces manual effort in creating repository and service classes for entities.
- Ensures entities have both default and service-based constructors to meet framework requirements.

Features

1. **Repository Class Generation**: Creates repository interfaces extending EntityRepository.
2. **Service Class Generation**: Creates service classes with CRUD operations, leveraging EntityService.
3. **Constructor Injection**: Adds service-based and default constructors to entity classes.
4. **Imports Management**: Ensures necessary imports are added to the entity classes.

Dependencies

- Relies on **Spring Framework** for dependency injection (@Autowired, @Repository, @Service).
- Uses **JavaParser** for analyzing and modifying Java source files.
- Integrates with **Maven's build lifecycle** to generate code during the generate-sources phase.

How It Works

1. Parsing Source Files

- findJavaFiles: Recursively scans the source directory for .java files.
- JavaParser: Parses each Java file into an abstract syntax tree (AST).

2. Identifying Target Classes

- isSubclassOfEntity: Determines if a class extends the specified base entity class.

3. Modifying Entity Classes

- addServiceConstructor: Adds a constructor with EntityService as a parameter.
- addDefaultConstructor: Adds a no-argument constructor.

4. Generating Repository and Service Classes

- generateRepositoryClass: Creates a Spring Data repository interface for the entity.
- generateServiceClass: Creates a service class extending EntityService with dependency injection.

5. Writing Changes

- Updates the entity file with new constructors and imports.
- Writes the generated repository and service files to the output directory.

How to Use

1. Setting Up the Plugin

2. Create Your Entity Classes

Entities must extend the base class (e.g., Entity). For example:

java
```
package com.yourcompany.models;
import com.yourcompany.entity.Entity;
public class Student extends Entity<Student>
   { private String name;
   private int age;
}
```

3. Run Maven Build

Run the following command to trigger the plugin:

bash
CopyEdit
```
mvn generate-sources
```

4. Generated Code

The plugin will:

- **Add missing constructors** to Student if not already present.
- **Generate repository**: StudentRepository.
- **Generate service**: StudentService.

Developer Notes and Best Practices

Extendability:

1. The plugin is easily extendable to generate more components (e.g., controllers, DTOs).

Error Handling:

1. Logs errors using Maven's logging API (getLog()).

Customization:

1. Configure basePackage, sourceDirectory, and outputDirectory in pom.xml for flexibility.

Code Generation Principles:

1. Ensures consistent naming conventions (e.g., StudentRepository, StudentService).
2. Avoids overwriting existing classes.

Integration with IDE:

1. Add the outputDirectory to your IDE's source paths to see generated code.

4.3.4 appGenPlug

Overview

The AppGenPlug Maven plugin is part of the Quentity framework, designed to simplify the creation of cross-platform mobile and desktop applications from web projects. The plugin has two primary goals:

1. make-app: Initializes and generates Capacitor projects for Android and iOS, as well as Electron projects for Windows, Mac, and Linux with native access.
2. sync: Synchronizes updates from your web project to the generated mobile and desktop applications.

Prerequisites

Before using the plugin, ensure the following are installed and properly configured:

- Node.js and npm (required for Capacitor and Electron dependencies)
- A valid application.properties file in your project's src/main/resources directory, containing the required configurations:
  spring.application.name=YourAppName
  quentity.app.id=com.example.yourapp
  quentity.app.url=http://your-app-url (hosting server domain name, schema://ip:port)

Goals

1. make-app

The make-app goal initializes and sets up the mobile and desktop applications for your project.

Execution Phase:

- Default Phase: package

Steps:

1. Reads application properties (application.properties).
2. Initializes a Capacitor project with the provided app name and ID.
3. Installs required Capacitor dependencies.
4. Adds Android and iOS platforms.
5. Configures the Capacitor capacitor.config.json file.
6. Installs and configures Electron dependencies.
7. Synchronizes frontend files into the native projects.
8. Sets up symbolic links for Node.js dependencies.

How to Use:

Run the following Maven command:

mvn quentity:make-app

2. sync

The sync goal synchronizes changes from your web project to the mobile and desktop applications.

Execution Phase:

- Default Phase: package

Steps:

1. Updates the Capacitor capacitor.config.json file with the latest configurations.
2. Copies updated frontend files to the native project directories.
3. Synchronizes Capacitor and Electron dependencies.

How to Use:

Run the following Maven command:
mvn quentity:sync

Detailed Configuration

1. Properties File

The application.properties file is critical for the plugin's functionality. Below are the
mandatory properties:

- spring.application.name: The name of the application.
- quentity.app.id: A unique identifier for the app (e.g., com.example.myapp).
- quentity.app.url: The base URL for your web application.

2. Directory Structure

The plugin uses the following directory structure:

- natives/apps/: Contains the Capacitor and Electron projects.
- src/main/frontend: Your web project's frontend source files.

Troubleshooting

Common Errors and Solutions

1. Error: 'npm' is not installed or not available in the system PATH
   - Ensure Node.js and npm are installed and accessible from the command line.
2. Error: application.properties file not found
   - Verify that the application.properties file exists in the src/main/resources directory.
3. Error: capacitor.config.json not found
   - Ensure the make-app goal has been executed before running sync.

Symbolic Link Errors

- On Windows, ensure you run the Maven command with administrator privileges.

Additional Information

Native Access

The applications generated by AppGenPlug offer native access capabilities through Capacitor and Electron, enabling features like file system access, notifications, and hardware interaction.

Extensibility

The generated Capacitor and Electron projects are fully customizable.
You can extend their functionality by modifying the generated files directly.

Example Workflow

1. Run the make-app goal to initialize mobile and desktop projects:
   a. mvn quentity:make-app
2. Make changes to your web application.
3. Synchronize updates to the native projects:

a. mvn quentity:sync

4.4 Application class

Description: The Application class is the entry point for a Spring Boot-based server application. It integrates Vaadin as a front-end framework, JPA for data persistence, and provides support for progressive web applications (PWAs).

Aspect	Details
Features	-**Spring Boot Entry Point**: Handles application startup. -**PWA Support**: Configures the app as a Progressive Web App (PWA). -**Push Support**: Enables real-time server-push functionality. -**Theming**: Applies a custom theme to the Vaadin application. -**PA Integration**: Supports nested repositories for database operations. -**Database Initialization**: Custom logic to initialize the database only if it's empty. -**Localization**: Provides basic localization through properties files stored in a HashMap.
Annotations	-**@SpringBootApplication**: Marks the class as a Spring Boot application. -**@EnableJpaRepositories**: Enables JPA repository support with nested repository handling -**@Theme**: Sets a custom Vaadin theme ("server"). -**@PWA**: Configures the app as a PWA with offline resources -**@Push**: Enables server-push functionality.
Fields	-**LOCAL**: A constant string for the default locale ("en"). -**LOCAL_PROPERTIES**: A HashMap storing localization properties files. -**REFLECTOR**: An instance of Reflector for runtime reflection.
Key Methods	- **main(String[] args)**: Starts the application, initializes components and services, and handles startup configuration. -**dataSourceScriptDatabaseInitializer**: Customizes database initialization to avoid redundant setups.
Key Features	-**PWA Configuration**: Installable app with offline capabilities -**Custom Database Initialization**: Database is initialized only if it's empty. -**Vaadin Integration**: Modern web UI framework with theming and server-push. -**Localization**: Centralized localization management using a HashMap
Example Usage	**-Run the Application**: java -jar application.jar **-Custom Database Initialization**: Only initializes when no user records exist.

4.5 Database and API Server

Example	KeyMethods/Endpoints	Description	Component
QueryBuilder.buildQueryWithFilters(User.class, ...filters, "AND") → SELECT * FROM user	buildQueryWithFilters	Dynamically constructs SQL queries based on filters and logic provided by the use	**QueryBuilder**
genericService.getEntitiesWithDynamicQuery(User.class,filters, "AND", 0, 10)	getEntitiesWithDynamicQuery, saveEntities,updateEntities, deleteEntities	provides dynamic query interactions and CRUD operations with the database	GenericService
GET /api/user/getAll? status=active&logic=AND&start=0&pageSize=10	GET /api/{entityName}/getAll, POST /api/{entityName}/add, PUT /api/{entityName}/update, DELETE /api/{entityName}/delete	RESTful endpoints to interact with GenericService	GenericApiController
Not Applicable	Not Applicable	Configuration class for QueryDSL that creates a JPAQueryFactory bean	QueryDSLConfig
Not Applicable	Not Applicable	Helper class encapsulating dynamic query parameters such as filters, logic,and pagination	QueryRequest
GET /api/user/getAll? status=active&age>25&logic=AND&start=0&pageSize=10	Configure QueryDSL, Define Entities, Use REST API, Handle Exceptions	Steps to use the API effectively	How to Use
{"error": "Entity class not found: userx" }	invalid entity/field name errors.	Provides detailed error messages for invalid requests or operations	Error Handling
POST /api/user/add { "name": "deema", "age": 22,"status": "active" }	Not Applicable	Fetch active users above 25 or add a new user via REST AP	Example Use Case
Not Applicable	Not Applicable	The API simplifies database operations with dynamic query capabilities	Conclusion

summary:this API will be made to provide flexible and dynamic answers to queries on database entities. It will contain features that will enable developers to perform complex database operations like create, read, update, and delete entities dynamically.

And I used Spring Boot in the back-end of my project because it is simple and efficient to handle Java-based applications. For testing the

For testing my API endpoints, I have used Postman, which gave me a very good way to check requests and responses. In this case, the database used was XAMPP, a local server solution for managing databases, even though it is often associated with PHP. To interact with the database, I implemented JPA, or Java Persistence API, to map Java objects to database tables and perform CRUD operations.

Chapter 5

Conclusion & Future Work

5.1 Conclusion

The Quantity framework means a leap in paradigm because it automates the development of the database, front-end, and back-end. The invention provides an effective workflow that is seamless, hence greatly reducing the complexity of development and raising productivity for developers. The fact that the framework can generate feature-rich, scalable, and responsive web applications with no coding demonstrates enormous potential to positively impact the technology industry. The implementation of a dynamic website builder has achieved the main goals of our project and opened new opportunities for simpler and less complex development solutions.

5.2 Future Work

1. Better performance of complex data structure on large applications.
2. Integrate AI-based tools for wiser entity definitions and automated optimization.

3. Integrate real-time collaboration features inside the website builder for team-based development.

4. Extend multilingual support by adding AI-driven translation and localization features.

5. It provides an intuitive drag-and-drop interface for visual entity and schema design.

Chapter 6

References

[1] Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[2] Bloch, Joshua. Effective Java. 3rd Edition, Addison-Wesley, 2018.

[3] Fowler, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

[4] Alur, Deepak, et al. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall, 2003.

[5] Sun Microsystems. Java BluePrints. https://www.oracle.com/technetwork/java/javaee/blueprints/index.html. Accessed: 2025.

[6] Baeldung. "Design Patterns in the Spring Framework." https://www.baeldung.com/spring-framework-design-patterns. Accessed: 2025.

[7] Refactoring.Guru. "Design Patterns in Java." https://refactoring.guru/design-patterns/java. Accessed: 2025.

[8] GeeksforGeeks. "Java Design Patterns Tutorial." https://www.geeksforgeeks.org/java-design-patterns/. Accessed: 2025.

[9] Wikipedia. "Factory Method Pattern." https://en.wikipedia.org/wiki/Factory_method_pattern. Accessed: 2025.

[10] Wikipedia. "Composite Pattern." https://en.wikipedia.org/wiki/Composite_pattern. Accessed: 2025.

[11] Hibernate ORM. "Official Documentation." https://hibernate.org/. Accessed: 2025.

[12] Vaadin. "Vaadin Documentation." https://vaadin.com/docs. Accessed: 2025.

[13] Apache Lucene. "Lucene Documentation." https://lucene.apache.org/core/. Accessed: 2025.

[14] Capacitor Community. "Capacitor Documentation." https://capacitorjs.com/docs. Accessed: 2025.

[15] Electron Team. "Electron Documentation." https://www.electronjs.org/docs. Accessed: 2025.

[16] Oracle. "Jakarta EE Documentation." https://jakarta.ee/. Accessed: 2025.

[17] QueryDSL. "QueryDSL Documentation." http://www.querydsl.com/. Accessed: 2025.

[18] MySQL. "MySQL Documentation." https://dev.mysql.com/doc/. Accessed: 2025.