Entwurf

Dieses Entwurfsdokument stellt die interne Struktur der Anwendung dar. Einerseits wird beschrieben, welchen Entwurfsentscheidungen durch Wicket vorgegeben waren, andererseits werden die Datenhaltung, die Paketstruktur und einige besonderes interessante Aspekte der Implementierung erläutert.

1 Projektstruktur

Bevor auf die Architektur der Anwendung eingegangen wird, soll die Organisation der Dateien, Ordner und Klassen des Projektes kurz vorgestellt werden.

1.1 Grundlegende Organisation

Als grundlegende Ordnerstruktur des Projektes wurde die Standard-Struktur von Maven gewählt. Diese gestaltet sich wie folgt:

- src
  Das Quellverzeichnis des Projektes
  - main
    Die eigentliche Anwendung
    - java
      Der Java-Code
    - resources
      Die HTML-Dateien und Einstellungen für Hibernate und Log4j
    - webapp
      Das Root-Verzeichnis der laufenden Anwendung mit Bildern, statischen HTML-Dateien und CSS-Dateien
  - test
    Die JUnit-Tests
- target
  Das Zielverzeichnis, welches die kompilierten Class-Dateien und andere zur Laufzeit benötigten Dateien enthält
- pom.xml
  Die Konfigurationsdatei von Maven

Wir haben uns dazu entschieden, im Zuge des Wicket-Konzeptes der strikten Trennung von Logik und Design in Java-Code und HTML-Code, diese nicht nur in separaten Dateien zu halten, sondern die HTML-Dateien in ein eigenes Verzeichnis zu legen. Um dies zu erreichen nutzen wir das Feature von Maven, welches es erlaubt Dateien aus dem Quellverzeichnis an einen bestimmten Ort im Zielverzeichnis zu kopieren. Auf diese Weise werden die HTML-Dateien an die gleiche Stelle kopiert wie die Class-Dateien der dazugehörigen Wicket-Komponenten, so dass Wicket sie findet.

1.2 Paketstruktur

Der Code des Projektes ist in folgende Pakete unterteilt:

Paket	Inhalt
`usg.data`	Die Datenschicht mit den Entity-Klassen und Klassen für den Zugriff auf die Datenbank
`usg.lib`	Enthält die Hilfsklassen der Anwendung
`usg.links`	Beinhaltet Klassen für häufig benötigte Links zu bestimmten Seiten
`usg.pages`	Enthält alle Pages der Anwendung
`usg.panels`	Die Panels der Anwendung
`usg.validators`	Selbst entwickelte Validatoren für die Überprüfung der Eingaben nach einer Formularabsendung

Tabelle 1: Die Aufteilung der Klassen in Pakete

2 Architektur

Die Architektur der Anwendung wird im wesentlichen durch das verwendete Web-Framework Apache Wicket vorgegeben. In diesem Abschnitt soll daher auf die für das Verständnis unserer Anwendung nötigen Konzepte von Wicket eingegangen werden und geschildert werden wie diese eingesetzt wurden.

2.1 Grundlegende Funktionsweise von Wicket

Eine Wicket-Anwendung besteht aus einem Komponentenbaum und Markup (z.B. XHTML). Die Markup-Dateien enthalten dabei keine Logik. Stattdessen werden mittels wicketspezifischen XML-Attributen Elemente des Markups „markiert“, mit Hilfe dessen Komponenten sich auf diese beziehen, wobei zu jeder Komponente stets genau eine Markup-Datei existiert. Ein Auszug der Komponenten-Hierarchie ist in Abbildung 1 auf der nächsten Seite dargestellt. Wie die Abbildung zeigt, teilt Wicket die Komponenten in zwei Kategorien ein: `MarkupContainer`, welche andere Komponenten enthalten können und `WebComponents`, die einen „flachen“ Inhalt besitzen. Eine wichtige Komponente ist `Page`, da sie eine gesamte im Browser dargestellte Seite repräsentiert und somit die Wurzel eines Komponentenbaums bildet. Neben den vorgegebenen Komponenten können auch eigene entwickelt werden, welches eine gute Kapselung und Wiederverwendung von Code ermöglicht.

Wicket verwendet Modelle, um die Daten von Komponenten zu halten. Daher besitzt jede Komponente ein Modell und stellt Methoden für den Zugriff auf selbiges bereit. Modelle sind Implementierungen der Schnittstelle `IModel`, deren wichtigste Methoden `getObject` und `setObject` sind. Die Komponente kennt den Typ des im Modell enthaltenen Objekts nicht, wodurch in Wicket keine Modelle für die einzelnen Komponenten oder Geschäftsobjekte der Anwendungen entwickelt werden, sondern für die unterschiedlichen Quellen der Objekte. Typischerweise werden jedoch von Wicket vorgegebene Modell-Implementierungen wie `LoadableDetachableModel` oder `CompoundPropertyModel` eingesetzt, auf welche in Kürze noch genauer eingegangen wird.

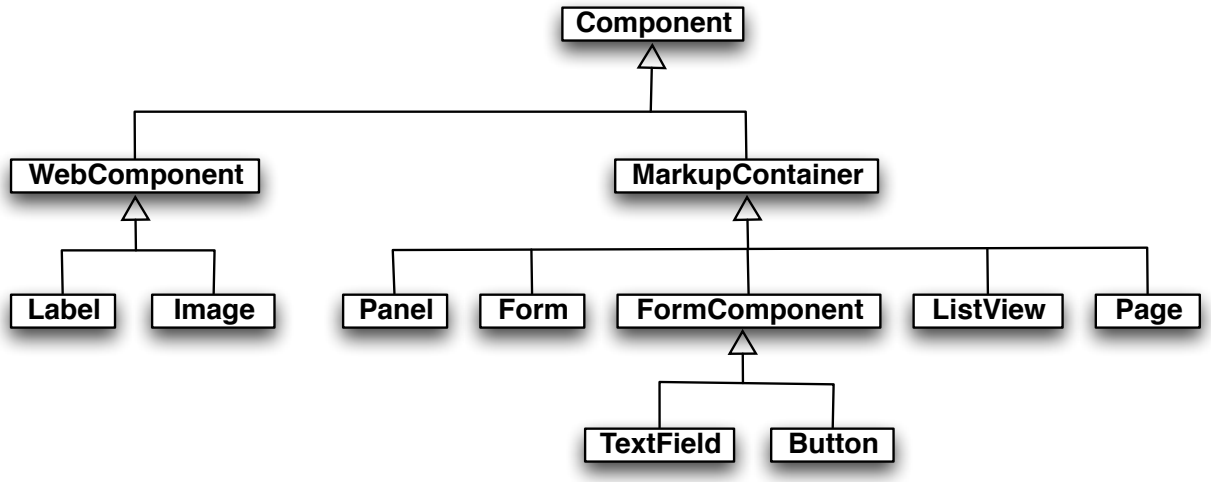

Abbildung 1: Komponenten-Hierarchie (Auszug)

Ein drittes wichtiges Konzept von Wicket sind Ereignisse. Wird ein Link angeklickt oder ein Formular abgesendet, wird ein Ereignis ausgelöst, auf welches man reagieren kann. Zu diesem Zweck bieten die entsprechenden Komponenten-Methoden wie `onClick` oder `onSubmit` an, die überschrieben werden müssen bzw. können, um die gewünschte Aktion bei Eintritt des Ereignisses auszuführen.

2.2 Struktur der Pages

Um die auf jeder Seite vorkommenden Elemente ohne Code-Duplizierung unterbringen zu können, haben wir uns dazu entschieden auf die sogenannte „Markup Inheritance“ zurückzugreifen. Dieses überträgt die Vererbbarkeit von Klassen auf Markup. So kann man von einer Komponenten-Klasse ableiten und hat somit zwei Markupdateien; eine für die Oberklasse und eine für die Unterklasse. In der Markupdatei der Oberklasse kann die Stelle, an die das Markup der Unterklasse eingefügt werden soll, mittels des Platzhalters `<wicket:child/>` festgelegt werden. Die Abbildung 2 auf der nächsten Seite verdeutlich den dadurch entstehenden Zusammenhang anhand der Registrierungsseite in der Rezepte-Anwendung. Auf der linken Seite ist die Ansicht im Browser dargestellt, welche durch die Markupdateien der einzelnen beteiligten Komponenten bestimmt wird. Der gelbe Bereich ist auf jeder Seite gleich, wohingegen der blaue Bereich von dem Ort des Besuchers abhängig ist; in diesem Fall befindet er sich auf der Registrierungsseite. Der mittlere Teil der Abbildung zeigt das dazugehörige Klassendiagramm. D.h. die Klasse `BasePage` leitet von der Wicket-Klasse `Page` ab und ist für den gelben Bereich der Seite verantwortlich. So legt sie das Markup für diesen Bereich fest und fügt die entsprechenden Komponenten (`SearchPanel`, `TagCloud`, ...) in den Komponenten-Baum ein. Davon abgeleitet existiert die Klasse `RegisterPage`, welche für den blauen Bereich zuständig ist. Die rechte Seite der Abbildung zeigt das Objektdiagramm, d.h. die Situation zur Laufzeit der Anwendung. Dies soll noch einmal verdeutlich, dass es zur Laufzeit nur ein Objekt der Klasse `RegisterPage` gibt, welches die gesamte Seite im Browser repräsentiert.

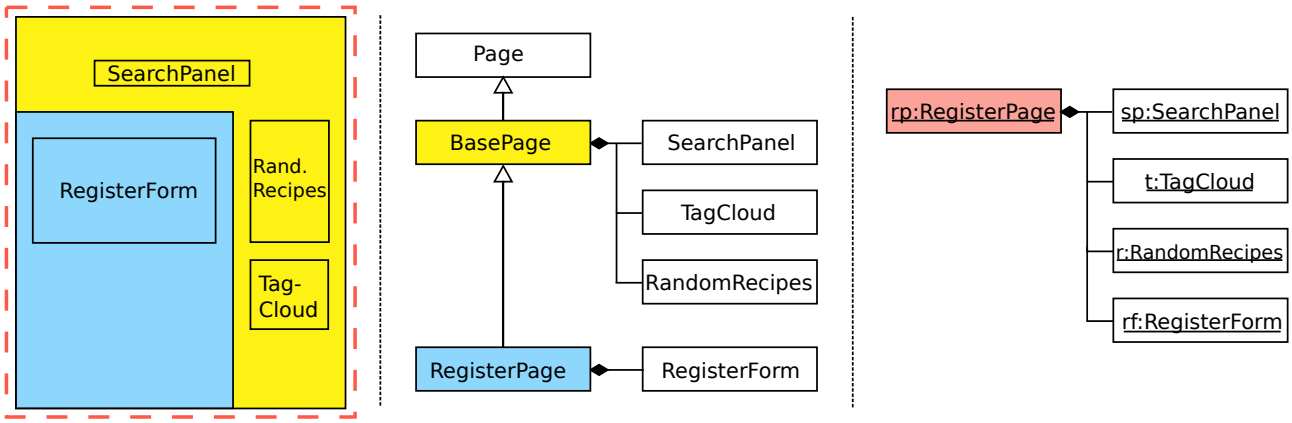

Abbildung 2: Markup Inheritance am Beispiel der Rezepte-Anwendung

2.3 Login und Berechtigungen

Die Authentifizierung und Authorisierung wird mit Hilfe der Wicket-Erweiterung „Auth Roles“ realisiert. Dafür ist es notwendig die `Application`-Klasse der Anwendung (repräsentiert die gesamte Anwendung und ist daher z.B. für grundlegende Einstellungen verantwortlich) von `AuthenticatedWebApplication` abzuleiten. Im Zuge dessen müssen Methoden implementiert werden, die der Erweiterung die gewünschte „Sign-in Page“ verraten und die eingesetzte `Session`-Klasse. Letztere muss analog zur `Application`-Klasse von `AuthenticatedWebSession` abgeleitet sein. Die Oberklasse kümmert sich bereits um den Status des Benutzers, d.h. Gast oder angemeldeter Benutzer, verlangt jedoch eine Implementierung der Methode `authenticate`, um die Gültigkeit von gegebenem Benutzernamen und Passwort zu ermitteln. Anschließend können die Methoden `signIn` und `signOut` von `AuthenticatedWebSession` verwendet werden, um einen Benutzer an- bzw. abzumelden.

Die Authorisierung wird durch Annotationen geregelt. Die Annotation `AuthorizeInstantiation` kann zu einer `Page` hinzugefügt werden und erhält als Parameter eine Liste von Rollen. Jeder Benutzer mit einer dieser Rollen darf somit diese Page aufrufen. Die Rollen werden wiederrum in der von `AuthenticatedWebSession` abgeleiteten Klasse durch Überschreibung der Methode `getRoles` festgelegt. Hat ein Benutzer keine dieser Rollen, wird die in der `Application`-Klasse festgelegte Sign-in Page aufgerufen. In der Rezepte-Anwendung wird die `AccessDeniedPage` für diesen Zweck eingesetzt. Diese hat lediglich die Aufgabe die `LoginPage` zu instanziieren, dieser eine entsprechende Fehlermeldung mitzugeben und zu dieser weiterzuleiten.

Sobald die Sign-in Page aufgerufen wird, wird zusätzlich die eigentlich angeforderte Seite gespeichert. Dafür wird der von Wicket vorgegebene Mechanismus der „Interception Page“ eingesetzt. Dieser erlaubt es bei jeder Komponente mit der Methode `redirectToInterceptPage` zu einer anderen Seite weiterzuleiten, speichert jedoch die ursprüngliche Seite und bietet die Möglichkeit später durch Aufruf von `continueToOriginalDestination` zu dieser zurückzukehren. Auf diese Weise kann die Anforderung, dass wenn ein Gast einen Kommentar abgeben möchte, dieser sich zuerst anmelden muss und anschließend direkt zur Kommentarabgabe zurückkehrt, sehr einfach realisiert werden.

2.4 Mobiles Theme

Wicket bietet von Haus aus ein Konzept für Lokalisierung der Anwendung und Benutzung verschiedener Styles. Das „Locale“ und der „Style“ werden dabei in der Session gespeichert. Wicket bezieht diese in die Suche nach Markupdateien und anderen Ressourcen mit ein, indem es bestimmte Dateinamen probiert. D.h. beispielsweise `<komponentenName>_<style>_<locale>.html`. Die Rezepte-Anwendung unterstützt derzeit lediglich die deutsche Sprache, so dass die Lokalisierung nicht benötigt wird. Um ein abgespecktes Theme für mobile Geräte anbieten zu können, werden jedoch unterschiedliche Styles verwendet. So gibt es für einige Komponenten sowohl eine Standard-Markupdatei (`<komponentenName>.html`) als auch eine für das mobile Theme (`<komponentenName>_mobile.html`).

An dieser Stelle tauchte ein Problem auf. Ohne weitere Maßnahmen verlangt Wicket, dass es zu jedem im Markup markiertem Element genau eine Komponente gibt und umgekehrt. Jedoch soll das mobile Style etwas abgespeckter sein, d.h. das eine oder andere Element soll gar nicht vorhanden sein. Um die Auslösung einer Exception bei nicht vorhandenem Element im Markup zu verhindern, existiert lediglich eine Debugging-Einstellung. Dies deutet darauf hin, dass Wicket den von uns gewählten Weg nicht vorgesehen hat. Dies ist teilweise auch nachvollziehbar, weil ohne Maßnahmen im Java-Code die Komponenten natürlich trotzdem erstellt werden, so dass Zeit und Speicher verschwendet wird. Uns erschien es jedoch äußert unelegant auf einen speziellen Style im Java-Code zu reagieren. Da der kleine Zeit- und Speicherverlust in diesem Fall nicht problematisch ist, haben wir uns für die Verwendung der Debugging-Einstellung entschieden.

2.5 Datenschicht

Die Daten der Anwendung werden in einer Datenbank gespeichert. Wir setzen eine PostgreSQL-Datenbank in der Version 9.0 ein und realisieren den Zugriff auf die Datenbank mit der Java Peristence API.

2.5.1 Repräsentation der Daten

Das Schema der Datenbank ist in Abbildung 3 auf der nächsten Seite zu sehen. Die unterstützten Datentypen decken sich weitgehend mit denen des SQL99-Standards. Eine Besonderheit ist das Vorgehen zur Erzeugung automatisch inkrementierender IDs. Dazu verwenden wir den Datentyp `BIGSERIAL`. Dieser bewirkt, dass automatisch eine Sequenz angelegt wird, deren Namen sich aus dem Tabellennamen, dem Name des zu erhöhenden Feldes und dem Zusatz „seq“ zusammensetzt. Auf diese Weise wird durch minimalen Konfigurationsaufwand erreicht, dass die betroffenen Spalte immer eindeutige Werte enthält.

2.5.2 Abbildung der Struktur auf Klassen

Innerhalb der Anwendung arbeiten wir mit Objekten, die die verschiedenen Datensätze der Datenbank widerspiegeln. Die Entityklassen enthalten jeweils die Attribute, die in der Datenbank gespeichert sind. Für jedes Attribut gibt es eine Getter- und Setter-Methode. Eine Übersicht über diese Klassen ist in dem Klassendiagramm in Abbildung 4 auf Seite 7 zu sehen. Um die Objekte in die Datenbank zu schreiben bzw. Datensätze aus der Datenbank in Objekte zu transformieren, verwenden wir die Hibernate-Implementierung der Java Persistence API.

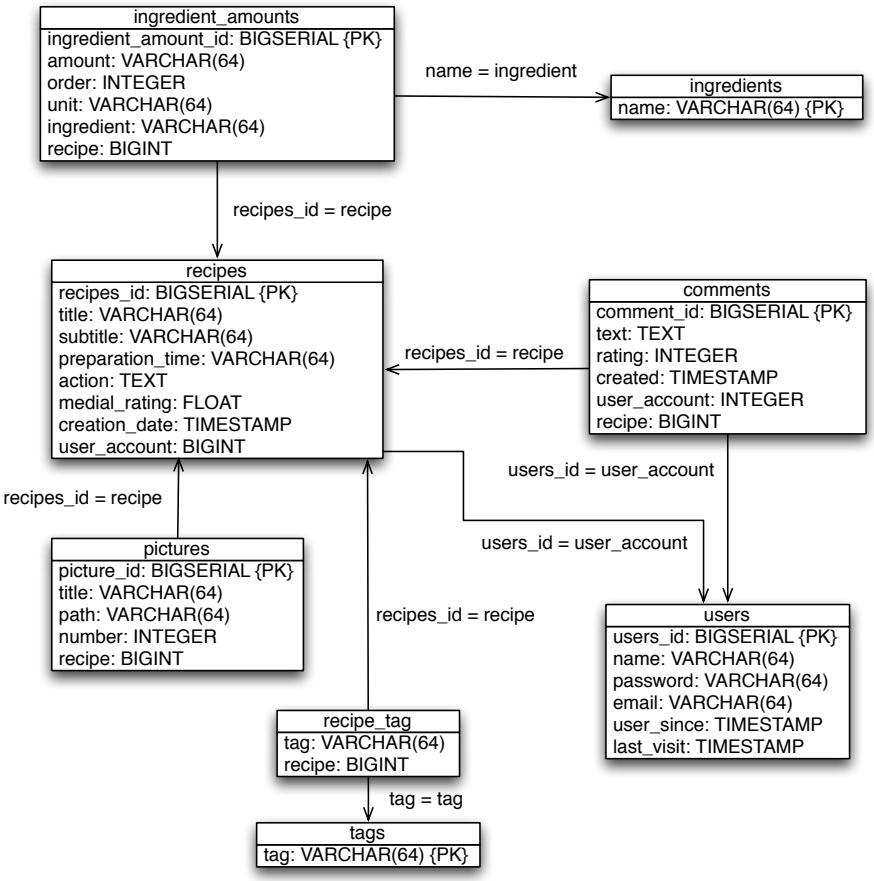

Abbildung 3: Diagramm des Datenbankschemas

Die Konfiguration des Zugriffs erfolgt über Annotationen in den Entityklassen. Die Zuordnung einer Klasse zu einer Tabelle geschieht durch die Annotation `@Table(name = "Tabellenname")`. Ist ein Attributname in einer Klasse mit dem dazugehörigen Spaltennamen in der Datenbank identisch, ist es nicht nötig den Namen per Annotation explizit bekannt zu machen.

Fremdschlüsselbeziehungen werden durch die Schlüsselworte `@ManyToOne`, `@OneToMany` bzw. `@ManyToMany` gekennzeichnet. Um ein Objekt nur durch das Persistieren dieses Objektes in „voller Tiefe“ in die Datenbank speichern zu können, greifen wir auf die Möglichkeit des Kaskadierens zurück. Beispielsweise kann so ein Rezept inkl. der dazugehörigen Bilder durch Persistieren des Rezeptes vollständig gespeichert werden. Um dies zu erreichen, muss die Richtung der Kaskadierung festgelegt werden, d.h. ob ein Speichern des Bildes das Rezept ebenfalls speichert oder umgekehrt. Da in unserer Anwendung das Rezept das „zentrale Entity“ bildet und die meisten anderen Entities zu einem Rezept zugeordnet sind (Bilder, Kommentare, Tags, Zutaten), haben wir uns dazu entschieden bei Speicherung eines Rezeptes die dazu zugeordneten Entities mitspeichern zu lassen. Dies wird durch die Angabe von `cascade = CascadeType.PERSIST, CascadeType.REMOVE` bei den `@OneToMany`-Annotationen erreicht. Auf diese Weise wird dafür gesorgt, dass im Falle von Persistierungen und Löschungen diese Operation an die durch diese Fremdschlüsselbeziehung zugeordneten Objekte weitergereicht wird. Das `CascadeType.REMOVE` ist dabei lediglich für die Unittests

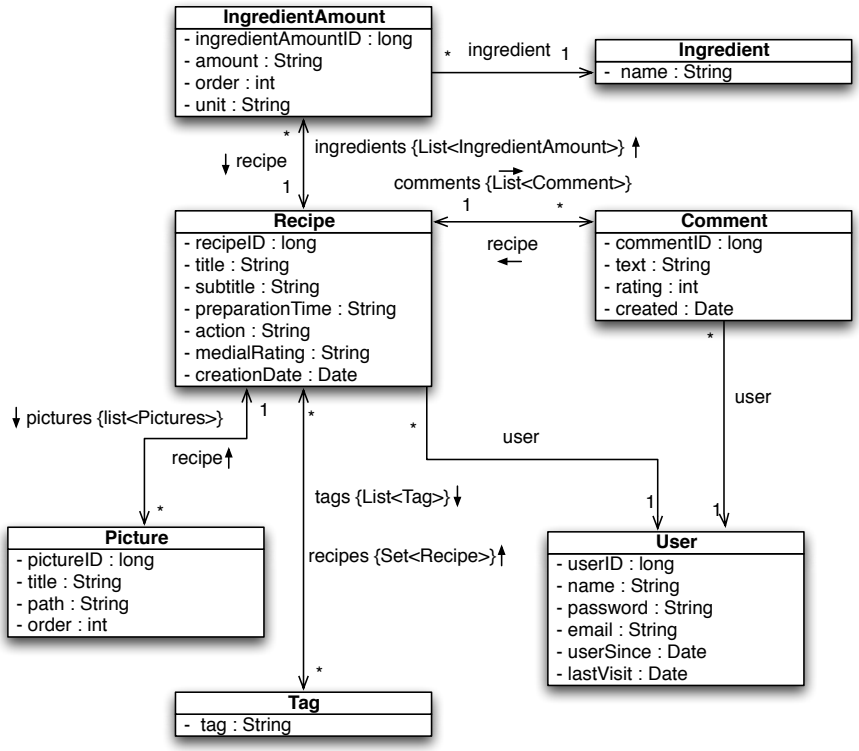

Abbildung 4: Klassenmodell

notwendig.

Um auf Eingaben des Benutzers, die länger als das dazugehörige Feld in der Datenbank sind, angemessen zu reagieren, d.h. keinen Laufzeitfehler zu verursachen, sondern dem Benutzer eine entsprechende Meldung anzuzeigen, wird die Eigenschaft `length` der Annotation `Column` verwendet. Diese wird von Hibernate zur Erzeugung des Datenbankschemas aus den Entity-Klassen eingesetzt, nicht jedoch für die Überprüfung der Länge der zu speichernden Werte. Aus dem Grund muss dies manuell sichergestellt werden. Zu diesem Zweck existiert in der Rezepte-Anwendung in der Klasse `EntityUtil` die Methode `getColumnLength`. Diese erhält ein Klassen-Objekt und den Namen eines Feldes dieser Klasse, sucht mit Hilfe der Reflection-API das Feld mit diesem Namen heraus inkl. der `Column` Annotation, sofern vorhanden, und gibt den Wert der Eigenschaft `length` zurück. Diese Methode wird bei einem Längen-Validator für Wicket eingesetzt, um zu lange Eingaben vor dem Zugriff auf die Datenbank abzufangen und somit derartige Fehler zu vermeiden.

2.5.3 Zugriff

Für den Zugriff auf die Datenbank gibt es die als Singleton realisierte Klasse `DBCon`. In dieser wird die Verbindung zur Datenbank aufgebaut und es werden Methoden zur Transaktionssteuerung (`beginTransaction`, `commitTransaction` und `rollbackTransaction`), zur Persistierung sowie Löschung von Objekten und für das Holen von Datensätzen aus der Datenbank (`getRow`, `getRows`, ...) angeboten.

Das Holen von Datensätzen wird in der Rezepte-Anwendung über Named Queries realisiert. Diese sind in den jeweiligen Entity-Klassen hinterlegt um sie nicht über den gesamten Code zu verteilen, sondern zentral in der Datenschicht zu halten.

2.6 Pages

Die Pages bilden gewissermaßen die Einstiegspunkte in die Anwendung, weil diese die Seiten im Browser repräsentieren und somit durch den Benutzer aufgerufen werden. Wird eine Page vom Benutzer angefordert, übernimmt Wicket die Instanziierung der Page-Klassen. Die Pages verwenden die Datenschicht, selbst entwickelte Panels und Hilfsklassen. Auf einige interessante Panels und Hilfsklassen wird in den nächsten Abschnitten eingegangen.

Um einen Eindruck von dem Aufbau der Pages zu vermitteln, soll hier exemplarisch die Klasse `RegisterPage` vorgestellt und erläutert werden. Wie der Name vermuten lässt, beinhaltet diese Seite ein Formular, mit Hilfe dessen sich der Benutzer registrieren kann. Die Implementierung sieht, etwas vereinfacht, folgendermaßen aus:

```
1  public final class RegisterPage extends BasePage {
2    public RegisterPage() {
3      final User u = new User();
4
5      Form<User> form = new Form<User>("form",new CompoundPropertyModel<User>(u)) {
6        @Override
7        protected void onSubmit() {
8          u.setUserSince(Calendar.getInstance().getTime());
9          u.setPassword(Util.getHash(u.getPassword()));
10         DBCon.get().beginTransaction();
11         DBCon.get().persist(u);
12         DBCon.get().commitTransaction();
13
14         StatusPage page = new StatusPage();
15         page.info("Ihr Benutzerkonto wurde erfolgreich angelegt. Sie können sich" +
16           " jetzt mit diesem Konto anmelden.");
17         setResponsePage(page);
18       }
19     };
20
21     form.add(
22       new ValidatedRequiredTextField<String>("name")
23         .add(StringValidator.minimumLength(3))
24         .add(StringValidator.maximumLength(EntityUtil.getColumnLength(User.class, "name")))
25         .add(new DupUsernameValidator())
26     );
27
28     FormComponent<String> pw = new PasswordTextField("password")
29       .add(StringValidator.minimumLength(5));
30     pw.add(new ValidationAttributeModifier(pw));
31     form.add(pw);
32
33     form.add(
34       new ValidatedRequiredTextField<String>("email")
35         .add(StringValidator.maximumLength(EntityUtil.getColumnLength(User.class, "email")))
36         .add(EmailAddressValidator.getInstance())
37         .add(new DupEmailValidator())
38     );
39
40     add(form);
41   }
42 }
```

Zunächst soll das in der Anwendung oft verwendete `CompoundPropertyModel` erläutert werden. Konzeptionell gesehen dient es dazu für einen Container festzulegen, dass er ein bestimmtes Objekt repräsentiert. Beispielsweise erhält hier das Registrierungs-Formular dieses Modell, um auszudrücken, dass es gewissermaßen einen Benutzer (das Objekt `u`) repräsentiert bzw. damit ein Benutzer erstellt wird. D.h. die einzelnen Formularelemente entsprechen den Eigenschaften des Benutzers und das gesamte Formular somit dem Benutzer.

Um zu verstehen wie es funktioniert, muss man zunächst wissen, dass jede Komponente eine ID besitzt, über welche es an das im Markup ebenfalls durch diese ID gekennzeichnete Element gebunden wird. Besitzt ein Container (z.B. ein Formular, ein Panel oder eine Page) als Modell das `CompoundPropertyModel`, so müssen dessen Kindkomponenten nicht explizit ein Modell angeben. Stattdessen wird das Modell des Containers und die ID der Kindkomponente dafür herangezogen. Die ID dient dabei zur Spezifikation einer Eigenschaft des Objektes im Container. D.h. beispielsweise das im Code verwendete `ValidatedRequiredTextField` mit der ID „name“ bezieht sich auf die Eigenschaft „name“ des User-Objektes `u`. Konkret wird die Methode `u.getName()` aufgerufen. Es kann jedoch nicht nur auf „flache“ Eigenschaften zugegriffen werden, sondern auch auf verschachtelte. So führt die Angabe von „address.zipCode“ zum Aufruf von `u.getAddress().getZipCode()`.

Dieses Konzept dient nicht nur dazu, an den Wert von Eigenschaften zu gelangen, sondern es wird auch verwendet, um diese zu ändern. D.h. Wicket füllt in diesem Beispiel das Objekt `u` mit Daten (via `u.setName()`, `u.setPassword()` und `u.setEmail()`) nachdem das Formular abgesendet wurde, so dass in der Implementierung von `onSubmit` des Formulars das gefüllte Objekt mehr oder wenig direkt in die Datenbank gespeichert werden kann. An dieser Stelle muss lediglich noch ein Hash des Passworts erstellt werden und das Datum der Registrierung hinzugefügt werden, da dies durch das Formular nicht abgedeckt wird.

Ein ebenfalls viel genutztes Konzept sind die Validatoren. Diese können, wie z.B. in Zeile 23 des Code-Listings sichtbar, einfach zu Formularfelder hinzugefügt werden. Die Validierungen werden noch vor Aufruf von `onSubmit` von Wicket durchgeführt und sollte ein Fehler aufgetreten sein, wird diese Methode gar nicht aufgerufen, sondern es wird dem Benutzer eine (oder mehrere) entsprechende Fehlermeldung angezeigt.

2.7 Panels

Panels sind Container, die eine eigene Markup-Datei besitzen. Durch sie können somit mehrere Elemente, die häufig gemeinsam verwendet werden, zusammengefasst werden. Die Abbildung 5 auf der nächsten Seite zeigt einige von uns entwickelte Panels. Diese dienten teilweise zur Kapselung (z.B. die Tag-Cloud, die ausgewählten Rezepte oder die durchschaltbaren Bilder bei der Rezeptanzeige), da diese Panels nur einmal verwendet werden. In anderen Fällen wurden Elemente zu Panels zwecks Wiederverwendung zusammengefasst, d.h. um Code-Duplizierung zu vermeiden (z.B. die mittels Sternen dargestellte Bewertung oder der Bildlink zu einem Rezept).

2.8 Hilfs-Klassen

In diesem Abschnitt soll auf einige interessante Punkte der Hilfsklassen unserer Rezepte-Anwendung eingegangen werden.

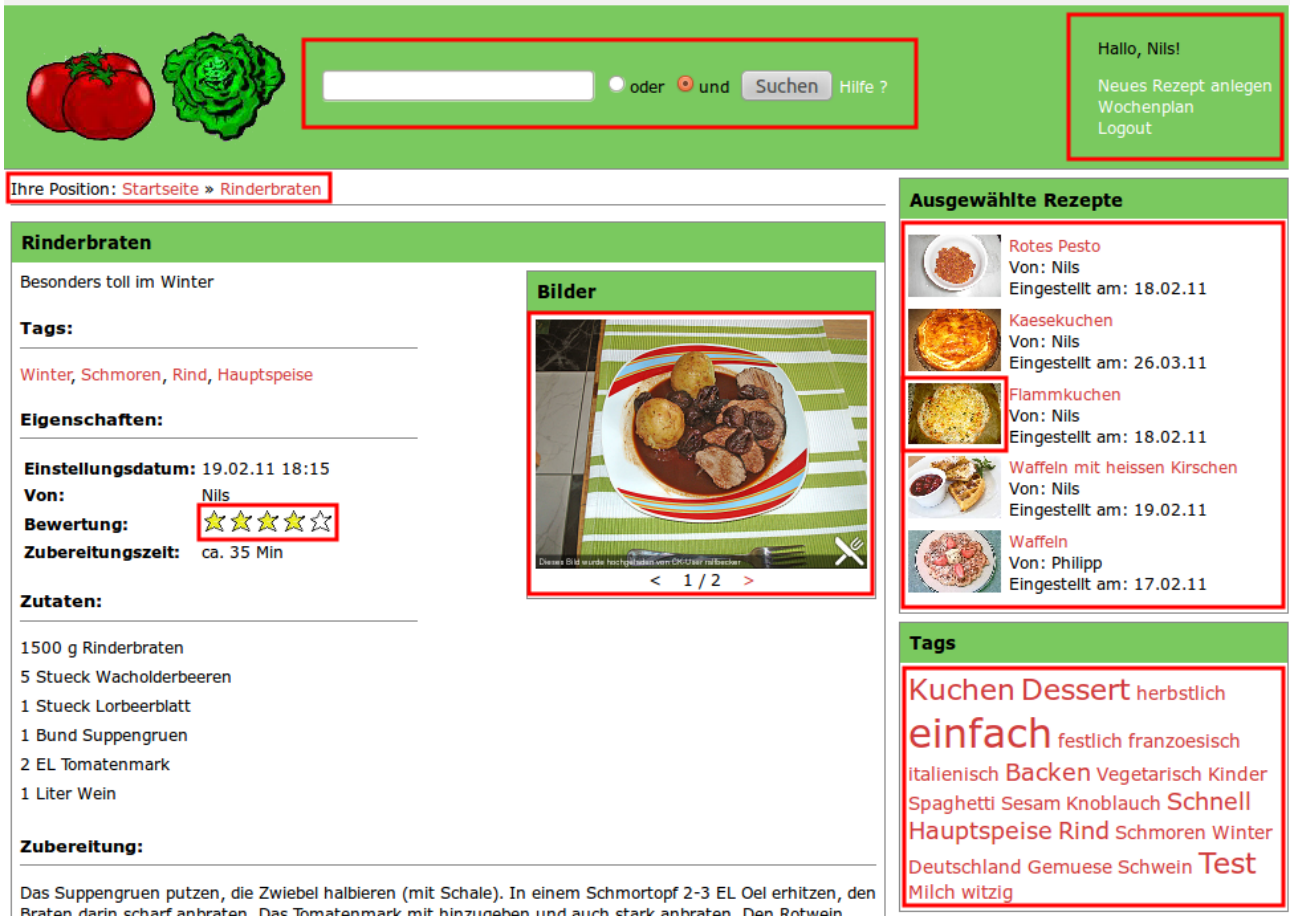

Abbildung 5: Beispiele für eigene Panels

2.8.1 Listen-Editor

Das Formular zur Anlegung neuer Rezepte beinhaltet mehrere Elemente, die nach dem gleichen Muster funktionieren sollen: Die Zutaten, die Tags und die Bilder. Bei diesen kann es jeweils eine beliebige Anzahl geben, so dass der Benutzer Möglichkeiten benötigt weitere Formularelemente hinzuzufügen und bestehende wieder zu entfernen. Zudem soll teilweise die Reihenfolge festlegbar sein. Aus diesem Grund haben wir uns dazu entschiedene eine eigene Komponente zu entwickeln, die bei den Zutaten, Tags und Bilder zum Einsatz kommen kann.

Diese Komponente basiert auf einem Artikel aus dem „Wicket in Action“-Blog[1] und wurde lediglich um die Möglichkeit erweitert, die Reihenfolge der Listenelemente ändern zu können. Das Klassendiagramm in Abbildung 6 auf der nächsten Seite vermittelt einen Überblick über die Struktur des „Listen-Editors“. Das Grundprinzip ist, dass `ListEditor` die Elemente der Liste hält und die von `EditorButton` abgeleiteten Klassen dem Benutzer erlauben, die Liste zu manipulieren. Dabei existieren zwei verschiedene Listen: Eine Liste der Komponenten und eine Liste der Geschäftsobjekte (z.B. Zutaten). Erstere werden von `RepeatingView` verwaltet, letztere von `ListEditor` (`items`). Wird ein Element hinzugefügt oder

[1] `http://wicketinaction.com/2008/10/building-a-listeditor-form-component/`

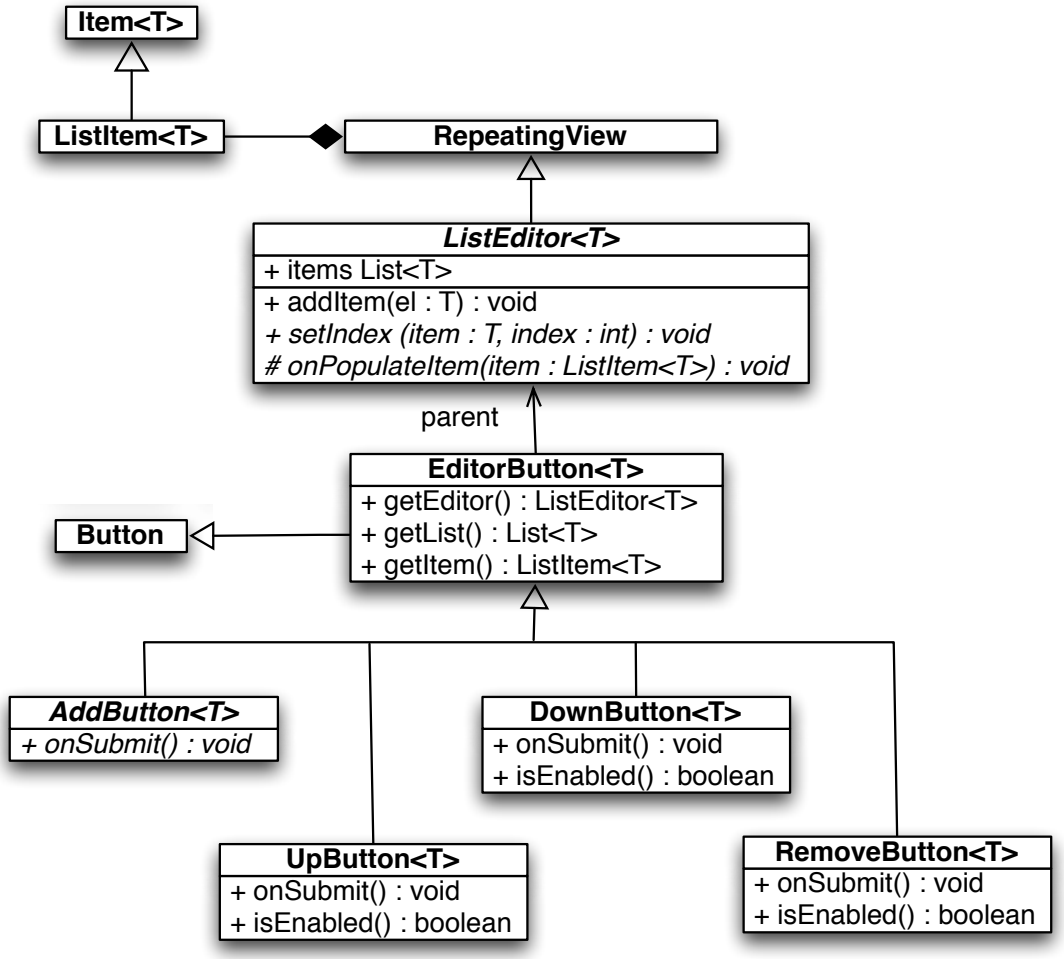

Abbildung 6: Klassenmodell des ListEditor

entfernt oder die Reihenfolge verändert, muss diese Änderung also stets an beiden Listen vorgenommen werden.

Wie das Klassendiagramm zeigt, ist die Klasse `ListEditor` abstrakt, da die Methoden `setIndex` und `onPopulateItem` noch implementiert werden müssen. Erstere dient dazu die Sortierung bei Änderung der Reihenfolge in den Geschäftsobjekten zu speichern (wie diese dort gespeichert wird ist natürlich unterschiedlich, so dass der ListEditor dies dem Verwender überlässt). Letztere muss überschrieben werden um ein Listen-Element zu füllen, d.h. um beispielsweise Formularelemente für das Geschäftsobjekt oder Instanzen von `UpButton` oder `DownButton` zur Änderung der Reihenfolge hinzuzufügen. Ein `AddButton` ist dafür gedacht außerhalb der Liste angebracht zu werden um die Liste zu erweitern. Dafür muss die Methode `onSubmit` überschrieben werden, welche `ListEditor.addItem` verwenden kann um ein Element hinzuzufügen.

Die Abbildung 7 auf der nächsten Seite zeigt ein Verwendungsbeispiel. Der Bereich 1 enthält dabei die Formularelemente für ein Geschäftsobjekt „Zutat“, Bereich 2 einen `UpButton`, `DownButton` und `RemoveButton`,

wohingegen Bereich 3 einen `AddButton` beinhaltet.

Abbildung 7: Beispielhafte Verwendung des Listen-Editors

2.8.2 E-Mails verschicken

Das Versenden von E-Mails wird in der Rezepte-Anwendung von der JavaMail-API übernommen. Um die Verwendung zu vereinfachen und von der Geschäftslogik der Anwendung zu trennen, wird die Klasse `Mailer` bereitgestellt. Diese erhält ein Properties-Objekt und - sofern eine Authentifizierung erforderlich ist - Benutzername und Passwort. Die Methode `send` bietet die Möglichkeit an eine E-Mail von einem bestimmten Absender an einen bestimmten Empfänger zu schicken. Sie übernimmt sowohl die Authentifizierung als auch das eigentliche Verschicken der E-Mail.

Der Verwender der Klasse liest das Properties-Objekt aus einer Property-Datei, welche sowohl JavaMail-API-spezifische Attribute enthält als auch Attribute, die von dem Verwender der Klasse `Mailer` verarbeitet werden. Letztere geben an ob eine Authentifizierung notwendig ist und wenn ja, den Benutzernamen und das Passwort, welche dann wiederum `Mailer` übergeben werden können. Auf diese Weise können die Authentifizierungsdaten geändert und verschiedene Mail-Protokolle verwendet werden ohne das eine Anpassung des Codes erforderlich ist.

2.8.3 Tag-Cloud

Die Tag-Cloud zeigt alle vorhandenen Tags an und verdeutlicht mit Hilfe variierender Schriftgröße die Anzahl der jeweils dazu zugeordneten Rezepte. Zur Berechnung der Schriftgröße wird folgende Formel verwendet:

$$fontSize = base + range * (recipeCount/maxCount)$$

Die Variable $base$ ist dabei die Mindestgröße und $range$ ein skalierender Faktor, der angibt auf welche Größe es maximal wächst ($base + range$). Die Variable $recipeCount$ ist die Anzahl der Rezepte, die zu einem bestimmten Tag existieren und $maxCount$ die maximale Anzahl dessen.

Diese Berechnung wird von der Klasse `TagWeighter` übernommen, welche die Liste aller Tags und die Parameter $base$ und $range$ im Konstruktor erhält und eine Methode anbietet um das „Gewicht“ eines bestimmten Tags zu berechnen. Beispielsweise könnte $base$ auf 100 und $range$ auf 200 gesetzt werden. Angenommen $maxCount$ wäre 10, dann würde ein Tag mit einem Rezept das Gewicht 120 erhalten, eines mit zwei 140 und eines mit zehn 300.