

3 Wicket-Überblick

In den folgenden Abschnitten werden die Bausteine von Wicket vorgestellt, die das Wicket-Programmiermodell bilden.

Es wird gezeigt, welche Abstraktionen Wicket gegenüber der direkten Verwendung der Servlet-API vorsieht und wie die Softwareentwicklung davon profitiert.

Wir starten mit einem Streifzug durch die Bausteine einer Wicket-Anwendung und ihre Abhängigkeiten, um uns aus der Vogelperspektive einen Überblick zu verschaffen. Damit wollen wir erreichen, dass die Grundlagen von Wicket so weit erläutert werden, dass einfache Anwendungen entwickelt und verstanden werden können. Im Kapitel 7 werden wir dann die Feinheiten dieser Konzepte detailliert untersuchen.

3.1 Aufbau einer Wicket-Anwendung

Eine mit Wicket entwickelte Anwendung stellt eine zur Servlet-API 2.3 konforme Webanwendung dar. Wicket-Anwendungen lassen sich dadurch auf jedem Webcontainer betreiben, der mindestens diese Servlet-Spezifikation unterstützt.

*Konform zur
Servlet-API 2.3*

Eine Wicket-Anwendung ist definiert als eine ereignisgesteuerte Abfolge von dynamisch erzeugten Webseiten. Dies entspricht der Definition des klassischen Modells einer Webanwendung, bei dem nach jedem HTTP-Request eine neue Seite zurückgeliefert und im Browser gerendert wird (*Multi-Page-Modell*). Darüber hinaus wird auch das Ajax-basierte Modell unterstützt, bei dem Benutzeranfragen synchron oder asynchron mit JavaScript versendet und die Serverantworten über JavaScript verarbeitet werden. Serverantworten sind im einfachsten Fall HTML-Fragmente, die durch Manipulation des DOM-Baumes in die aktuelle Seite eingearbeitet werden. Man spricht vom *Single-Page-Modell*, wenn die Anwendung im Wesentlichen aus einer Seite besteht und jeder Request per Ajax ausgeführt wird.

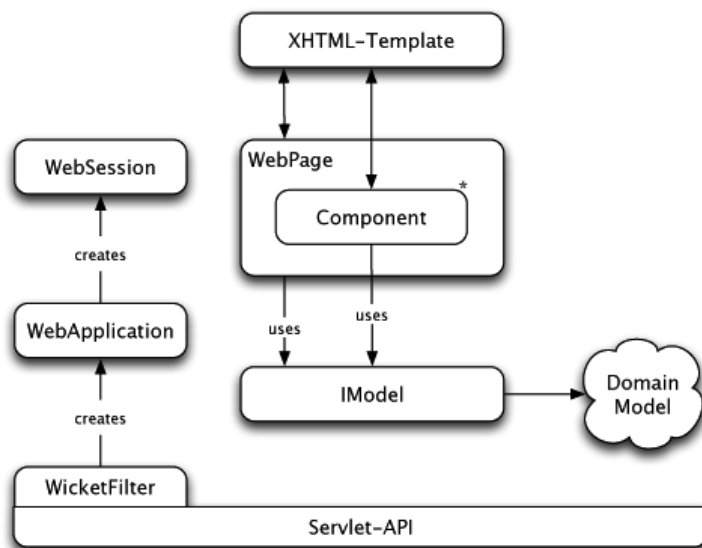
*Klassisches Webmodell
und Ajax*

Wicket versteckt die technischen Details beider Modelle hinter der Framework-API, was es dem Entwickler erlaubt, sehr schnell zwischen beiden Modellen zu wechseln bzw. diese in der Anwendung

zu kombinieren. Dabei ist hervorzuheben, dass Ajax-Handling und DOM-Manipulation in den meisten Fällen alleine mit Java-Mitteln erreicht werden können. Entsprechende JavaScript-zu-Java-Wrapper sind Bestandteil der Wicket-Distribution. Da wir der Meinung sind, dass Wicket bezüglich Ajax ein herausragendes Programmiermodell anbietet, haben wir dem Thema ein eigenes Kapitel gewidmet (siehe Kapitel 8).

Wicket definiert für klar voneinander abgrenzbare Bausteine einer Webanwendung entsprechende Abstraktionen. Abbildung 3-1 gibt einen Überblick über die wichtigsten Bausteine und ihre Beziehung untereinander. Nachfolgend wollen wir diese im Einzelnen betrachten.

Abb. 3-1
Bausteine einer
Wicket-Anwendung



WebPages, Components, XHTML-Templates und Wicket-Models (Interface IModel) bilden die Kernelemente einer Wicket-Anwendung. Wie diese Elemente in der laufenden Anwendung zusammenspielen, soll am Beispiel einer Benutzerinteraktion gezeigt werden. Zur Veranschaulichung dient Abbildung 3-2. Das Beispiel zeigt konzeptionell, was passiert, wenn der Benutzer den Inhalt eines Formulars an den Server schickt.

*Ablauf einer
Benutzeranfrage*

Der Browser sendet den Request an den Server, wo er von Wicket auf einen Methodenaufruf an eine Komponente weitergeleitet wird. Diese Komponente repräsentiert aus Sicht des MVC-Patterns sowohl *Controller* als auch Teile des *View*.

Die Funktion des Controllers übernehmen dabei Event-Handler-Methoden in der Komponente wie z. B. `onClick()` oder `onSubmit()`. Sie

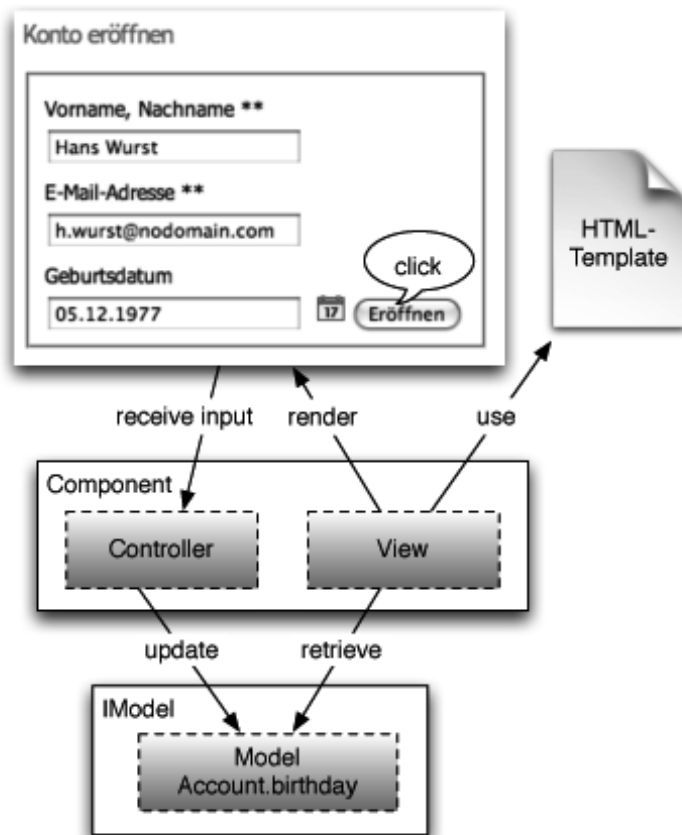


Abb. 3-2

Typischer Ablauf einer Benutzeranfrage

sind dafür zuständig, Benutzereingaben zu verarbeiten, das Modell zu aktualisieren und gegebenenfalls einen neuen View aufzurufen.

Das *Modell* wird durch Klassen der Anwendungsdomäne repräsentiert. Dort finden wir Klassen wie Kunde, Warenkorb, Projekt oder Aufgabe. Das IModel-Delegate-Interface ermöglicht der Komponente einen standardisierten Zugriff auf die Domänenobjekte und erlaubt es auch mehreren Komponenten, gemeinsam auf das gleiche Modell zuzugreifen, ohne voneinander zu wissen.

Nachdem die Verarbeitung im Controller, also in den Event-Methoden, abgeschlossen ist, rendert Wicket die Ergebnisseite, den *View*. Das ist standardmäßig die Seite, von der aus der Request kam, oder eine andere Seite, wenn der Controller dies bei Wicket mit `setResponsePage()` angefordert hat.

Eine Seite generiert ihr Markup für den Browser aus dem zugehörigen Template sowie aus den Ausgaben der Komponenten,

HTML einer Seite erzeugen

die sie enthält. Jede Komponente stellt durch Implementierung von `renderComponent()` die Funktionalität zur Erzeugung ihres Markups bereit. Je nach Art der Komponente kommt dabei ein (XHTML-)Template zum Einsatz oder nicht. Einfache Komponenten wie `Label` und `TextField` benötigen dafür kein Template, da sie nur ein kurzes Stück Text oder z. B. ein HTML-Eingabeelement ausgeben. Andere Komponenten wie `Panel` erzeugen HTML-Fragmente, die sie aus einer eigenen Template-Datei lesen. Neben statischem Markup kann die Komponente auch Daten aus dem Modell beziehen, greift also lesend auf das Modell zu.

*Unterschied zu MVC im
Original*

Kenner des Original-MVC-Patterns werden sicher bemerkt haben, dass der Controller den View nicht direkt aufruft. Normalerweise würde der Controller den View benachrichtigen, falls Änderungen in den Benutzereingaben oder dem Modell erkannt werden und eine neue Ausgabe erfordern. Eine solche Verbindung wäre aber nur dann sinnvoll, wenn entsprechende Änderungen im View aktiv zum Client gesendet werden können. Das HTTP-Request-Response-Modell lässt das aber nicht zu. Letztendlich reicht es, wenn der Controller das Modell aktualisiert. Für die Antwort auf den HTTP-Request rendert Wicket die Ergebnisseite, die sich dann aus den aktuellen Modelldaten speist.

3.1.1 Front-Controller: WicketFilter

Direkt auf der Servlet-API operierend sitzt der `WicketFilter`, über den jeder Request an eine Wicket-Anwendung geschleust wird.

Front-Controller

Wicket setzt, wie viele Servlet-basierte Frameworks, auf das Front-Controller-Pattern zur Steuerung des Request-Zyklus¹. Der Front-Controller wird entweder in Form eines Filters (`WicketFilter`) oder eines Servlets (`WicketServlet`) im Deployment-Deskriptor `WEB-INF/web.xml` konfiguriert. Der Einsatz von `WicketFilter` wird für die meisten Zwecke empfohlen. Der nachfolgende Kasten erläutert, warum es zwei Lösungsvarianten gibt.

¹Unter dem Request-Zyklus versteht man den Ausführungspfad vom Absenden eines Requests durch den Benutzer über seine Verarbeitung auf dem Server bis zur Anzeige der Serverantwort im Browser.

WicketFilter oder WicketServlet?

Die Varianten Filter und Servlet bieten zunächst einmal die gleiche Grundfunktionalität. `WicketFilter` hat gegenüber `WicketServlet` den Vorteil, dass Pre- und Post-Verarbeitung über weitere Filter möglich ist (*Filter-Chain*) und nicht von der Anwendung behandelbare Ressourcen-URLs einfach an den Container weitergereicht werden können. Es kann z.B. aus Performance-Überlegungen gewünscht sein, dass statische Ressourcen direkt vom Webserver ausgeliefert werden. Würde man ein Servlet verwenden, so wäre dieses selbst für das Streamen solcher Ressourcen verantwortlich.

Es gibt aber auch Situationen, in denen nur `WicketServlet` eingesetzt werden kann. Das betrifft z.B. Webcontainer, die keine saubere Unterstützung für Filter bieten (z.B. ältere IBM Websphere- oder Bea Weblogic-Container), oder einige OSGI-Container, die lediglich eine Registrierung von Servlets erlauben. Die OSGI-Spezifikation 4.x sieht einen `HTTPService` vor, der es erlaubt, Servlets zu betreiben – allerdings auf Basis von Servlet 2.1. Filter gibt es aber erst seit Servlet 2.3.

Jeder Request an die Anwendung wird über den Front-Controller geleitet. Dieser untersucht den Request anhand der URL und der Request-Parameter. Daraus wird die Art des Requests ermittelt. In Wicket gibt es unterschiedliche Request-Arten, die sich im Wesentlichen in den Request-Zielen unterscheiden und damit die nötigen Prozessschritte zur Generierung der Serverantwort vorgeben. Typische Ziele sind Seiten, Komponenten oder Ressourcen. Wir werden in Abschnitt 7.2 ausführlich auf die einzelnen Request-Arten mit den zugehörigen Verarbeitungsschritten eingehen.

3.1.2 WebApplication

Jede Wicket-Anwendung muss eine Anwendungsklasse als Erweiterung zu `WebApplication` bereitstellen. Die Anwendungsklasse bestimmt, welche Seite als Startseite der Anwendung dient. Darüber hinaus ist sie zentraler Anlaufpunkt für die Anwendungsconfiguration und für seitenübergreifend gemeinsame Ressourcen. Registriert wird die Anwendungsklasse im Rahmen der Deklaration von `WicketFilter` oder `WicketServlet` in `web.xml` mithilfe des Parameters `applicationClassName`. In Abschnitt 7.1.1 werden wir die Aufgaben der Anwendungsklasse im Einzelnen erläutern.

3.1.3 Seiten und Komponenten

Eine Wicket-Seite

Kommen wir nun zu den Bausteinen, mit denen sich der Anwendungsentwickler am häufigsten auseinandersetzen wird. In Wicket bilden Seiten und Komponenten die zentralen Abstraktionen. Eine Wicket-Anwendung ist eine Ansammlung von Seiten. Eine Wicket-Seite ist zunächst eine Klasse, die von `WebPage` abgeleitet wird. Seiten werden entweder aus der Anwendung heraus über einen der Konstruktoren erzeugt oder unter Angabe einer Seitenklasse zur Erzeugung an das Framework übergeben.

Komponenten

Jede Wicket-Seite besteht aus einer Hierarchie von Komponenten. Alle Komponenten erben von der Klasse `Component`, wie auch `WebPage` selbst. Komponenten werden im Konstruktor der Seite mit `new` erzeugt und mit `add(..)` hinzugefügt. Komponenten sind User-Interface-Elemente wie Labels, Textfelder, Buttons, Panels, Menüs oder Tabellen. Sogar `WebPage` selbst ist eine Komponente, da sie von `Component` erbt. Die Hierarchie kann beliebig viele Stufen aufweisen. Das bedeutet, Komponenten können wiederum Komponenten enthalten. Man spricht dann von Containerkomponenten. Die Komponentenhierarchie einer Seite wird also durch einen Baum beschrieben, dessen Wurzel die Seitenklasse ist, dessen Knoten aus Containerkomponenten wie Tabellen, Menüs oder Panels bestehen und dessen Blätter durch einfache Komponenten wie Labels und Eingabefelder repräsentiert werden.

Ein Teil der Komponenten ist in der Lage, Benutzerereignisse zu verarbeiten. Typische Ereignisse sind Klicks auf Links und Buttons oder das Selektieren eines Eintrags in einer Select-Box. Die korrespondierenden Komponenten wie `Button`, `Link` und `DropDownChoice` beinhalten Event-Listener-Methoden, die durch das Framework bei Auftreten eines Events aufgerufen werden. Die Anwendung kann eigenen Code durch Implementieren von Event-Callback-Methoden beisteuern, die von den Event-Listnern aufgerufen werden. Listing 3.1 zeigt im Ansatz, wie eine Komponente mit Event-Callback eingesetzt wird.

Listing 3.1
Callback-Methoden

```
Button next = new Button("nextButton") {
    public void onSubmit()
    {
        // Verarbeitung der Eingabewerte,
        // z.B. Aufruf der Businesslogik hier
    }
};
```

Logik in Java

In Wicket existiert eine strikte Trennung von Logik und Design. Seiten- und Komponentenklassen sind Ausgangspunkt für die Umsetzung der User-Interface- und Anwendungslogik (Businesslogik). User-Interface-

Logik (UI-Logik) umfasst beispielsweise das Iterieren von Tabellenzeilen oder das Einblenden zusätzlicher Eingabefelder, abhängig von der bisher getätigten Eingabe, oder die Auswahl der nächsten Anwendungsseite. Die Anwendungslogik dagegen ist in der Regel unabhängig vom User-Interface.

Das Design von Seiten und Komponenten wird in HTML-Templates ausgelagert. Die Vorlagen basieren auf dem XHTML-Standard und definieren vor allem den statischen Inhalt einer Seite. Des Weiteren sind sie dafür zuständig, die Verbindung zu den in der Seitenklasse instanziierten Komponenten herzustellen. Durch die Vergabe von speziellen Wicket-Attributen im Template, den Wicket-IDs (Attribut `wicket:id`), und der gleichzeitigen Verwendung dieser IDs beim Erzeugen der Komponenten im Konstruktor der Seitenklasse wird diese Verbindung hergestellt. Das Beispiel in Abbildung 3-3 veranschaulicht diesen Zusammenhang.

*Design über
XHTML-Templates*

Verzicht auf JavaServer Pages (JSP)

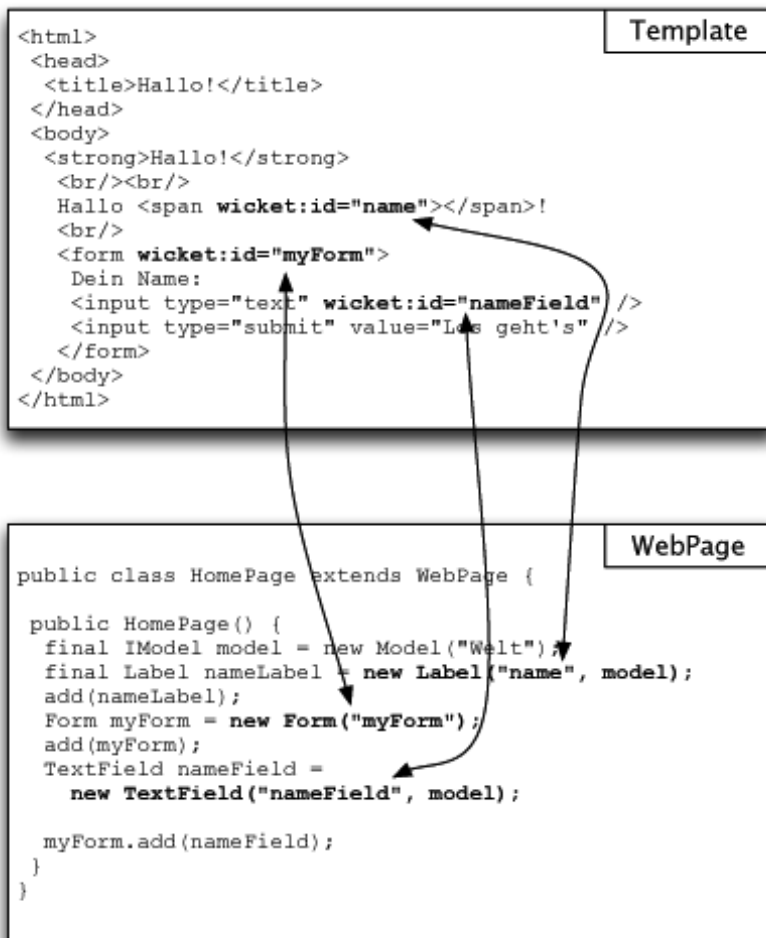
Auf den ersten Blick überrascht es, dass Wicket auf den Einsatz von JavaServer Pages (JSP) gänzlich verzichtet. Ein zentraler Punkt im Wicket-Design ist es, UI-Logik komplett in Java außerhalb des Templates zu behandeln, ganz im Gegensatz zu JavaServer Pages, wo UI-Logik und Design im gleichen Artefakt gepflegt werden. Des Weiteren steht aus Sicht vieler Experten außer Frage, dass aufwendig zu erstellende Taglibs und die begrenzten Möglichkeiten einer Expression-Language (JSP-EL) als Mittel zur Umsetzung von UI-Logik in JavaServer Pages nur eingeschränkt tauglich sind.

Zusammenfassend kann man sagen, dass Seiten und Komponenten nahezu alle Aufgaben übernehmen, die im Zusammenhang mit der Darstellung einer Anwendungsseite, der Verarbeitung von Benutzerereignissen und der Auswahl einer Folgeseite stehen. Man spricht von einem seitenzentrierten Framework. Im Gegensatz dazu verfolgt ein Model-2-Framework wie Struts einen Controller-zentrierten Ansatz. Hier steckt hinter jedem Formular und jedem Link eine eigene Klasse als Controller (im Struts-Jargon: Action). In Wicket ist die gesamte Controller-Logik entweder in einer Klasse (WebPage) vereint oder wird von ihr aus angestoßen. Das trägt vor allem zu einer besseren Übersichtlichkeit und Wartbarkeit des Codes bei und führt dazu, dass Controller-Logik direkten Zugriff auf alle Eigenschaften der Seite erhält.

Wicket-Seiten und -Komponenten bestehen normalerweise nicht nur aus Java-Klassen und XHTML-Templates. Sie beinhalten auch externe Ressourcen wie CSS-Dateien, Images, JavaScript-Dateien oder Resource-Bundles zum Zweck der Internationalisierung. Für ein ein-

Ressourcen

Abb. 3-3
Zusammenhang von
Seitenvorlage und
-implementierung



faches Bündeln und Wiederverwenden werden alle zu einer Seite bzw. Komponente gehörenden Artefakte über den Klassenpfad referenziert. Damit können Komponenten direkt per JAR-Datei verteilt oder verwendet werden. Wir werden in den Abschnitten 11.2 zu Ressourcen und 11.1 zur Lokalisierung tiefer in das Thema einsteigen.

Behaviors

Das Verhalten sowohl der Controller- wie auch der View-Aspekte einer Komponente kann auf verschiedene Weisen beeinflusst werden. Zum einen kann man Unterklassen schreiben und damit, wie in der Java-Entwicklung gewohnt, das Verhalten verändern. Zum anderen kann man auch sogenannte *Behaviors* hinzufügen. Ein Behavior ist ein verkapseltes Verhalten, mit dem eine Komponente erweitert werden kann und das bei bestimmten Events während des Lebenszyklus

aufgerufen wird, um z. B. CSS-Attribute oder einen JavaScript-Aufruf hinzuzufügen. Sie können aber auch komplexes Verhalten erzeugen. So kann z. B. das DatePicker-Behavior an ein Texteingabefeld, das für ein Datum bestimmt wird, ein komplettes JavaScript-basiertes Kalender-Widget anhängen, wie es in der Einleitung auf Seite 9 gezeigt wurde. Behaviors sind ein nützlicher Mechanismus, um Code wiederverwendbar zu machen, vor allem wenn unterschiedliche Komponenten nicht von der gleichen Superklasse erben können. Während in Kapitel 5 über Komponenten bereits einige nützliche Behaviors erwähnt werden, wollen wir sie in Abschnitt 7.1.4 detailliert beschreiben.

3.1.4 Wicket-Model

Ein letzter, aber dennoch sehr wichtiger Baustein entbehrt noch einer Beschreibung. Sie haben sich sicher schon gefragt, wie die Komponenten mit den Anwendungsdaten in Verbindung gebracht werden. Die Antwort darauf liefert das Wicket-Model-Konzept.

Anwendungsarchitekturen mit klarer Schichtentrennung definieren für den Zugriff der Präsentationsschicht auf persistente Anwendungsdaten prinzipiell zwei Varianten:

- Über objektrelationales Mapping wird eine objektorientierte Sicht auf die benötigten relational verwalteten Anwendungsdaten hergestellt. Die Anwendung nutzt die Möglichkeiten des eingesetzten OR-Mapping-Frameworks wie z. B. JBoss Hibernate, um CRUD-Operationen² oder mengenbasierte Operationen durchzuführen. Die erzeugte objektorientierte Sicht bildet das Domänenmodell der Anwendung, bestehend aus Domänenklassen.
- Service-Fassaden regeln den erlaubten Zugriff auf die Daten entweder durch einfache CRUD-Operationen oder durch komplexere Businesslogik-Service-Aufrufe. Die Anwendungsdaten werden über Data-Transfer-Objekte (DTO) der benötigten Granularität als In/Out-Parameter der Services verwendet. Häufig wird OR-Mapping für CRUD- und mengenbasierte Operationen herangezogen, wenn die Daten in einer relationalen Datenbank verwaltet werden, sodass beide Modelle miteinander kombiniert werden.

Je nachdem welche Variante zur Umsetzung der Businesslogik herangezogen wird, müssen wir entweder Domänenobjekte oder Data-Transfer-Objekte mit Wicket-Komponenten verknüpfen. Aus der Perspektive von Wicket ist diese Unterscheidung nicht von Bedeutung. Wicket führt nämlich mit dem Interface `IModel` eine Abstraktion ein,

²Create, Read, Update, Delete