# Das Vaadin 7 Grundlagenbuch

## Einstieg in das Vaadin Framework für Rich Internet Applications

Roland Krüger

# Das Vaadin 7 Grundlagenbuch

## Einstieg in das Vaadin Framework für Rich Internet Applications

Roland Krüger

Dieses Buch wird verkauft, unter http://leanpub.com/vaadinbuch

Diese Version wurde veröffentlicht am 2016-09-18

# Twitter dieses Buch!

Bitte unterstütz Roland Krüger, indem du über dieses Buch auf Twitter weiterempfiehlst!

Vorschlag: Verwende den folgenden Hashtag, wenn du über dieses Buch twitterst: #vaadinbuch.

Was sagen Andere über dieses Buch? Klick hier, um nach diesem Hashtag auf Twitter zu suchen:

https://twitter.com/search?q=#vaadinbuch

# Inhaltsverzeichnis

# Über den Autor

Roland Krüger, Jahrgang 1978, hat Wirtschaftsinformatik an der Universität Mannheim studiert und arbeitet seit 2011 als Trainer, Berater und Entwickler bei der Orientation in Objects GmbH[1] in Mannheim. Dort beschäftigt er sich hauptsächlich mit der Konzeption und Entwicklung von professionellen Webanwendungen mit verschiedenen Java Web-Frameworks, mit Softwareentwicklungsmethoden, wie Continuous Delivery und DevOps, und berät Kunden zu den Produkten der Firma Atlassian.

Roland hat das Vaadin Framework im Jahr 2010 kennen und schätzen gelernt. Sein erstes Projekt mit Vaadin, die Schnäppchenjäger-Community Spar-Radar (die es heute leider nicht mehr gibt), konnte er dank des einfachen Programmiermodells in kurzer Zeit erfolgreich umsetzen und in Produktion bringen. Seine Erfahrungen mit dem Framework hat er bei der Orientation in Objects GmbH zu einem Seminar zusammengetragen, mit dem er deutschlandweit als Trainer unterwegs ist. In seiner Vaadin-Schulung vermittelt er Grundlagen und Profiwissen über das Vaadin Toolkit.

Roland ist glücklich verheiratet und hat eine kleine Tochter. In seiner Freizeit ist er am liebsten auf Wanderungen im Odenwald oder auf stromgitarrenlastigen Konzerten unterwegs. Dass er nebenher natürlich auch hobbymäßig an Vaadin-Anwendungen bastelt, braucht hier sicherlich nicht besonders hervorgehoben werden ;-).

Man erreicht Roland auf Twitter mit seinem Handle @Roland_Krueger, auf GitHub findet man seine Repositories unter https://github.com/rolandkrueger, auf Google+ erreicht man ihn unter +RolandKrüger[2], und seine Homepage ist unter der Adresse http://www.rolandkrueger.info zu erreichen.

Roland bloggt regelmäßig zu Vaadin-bezogenen und anderen Themen auf dem Firmenblog der OIO GmbH[3].

---

[1] http://www.oio.de

[2] https://www.google.com/+RolandKr%C3%BCger

[3] http://blog.oio.de

# Vorwort: Über dieses Buch

Ein deutschsprachiges Vaadin-Buch, herausgegeben im Selbstverlag und vertrieben über eine Online-Plattform, bei der ein Buch schon während seiner Entstehung freigegeben werden kann. Wie kam es zu diesem Vorhaben?

Die Idee zu diesem Buch entstand, als ich als Seminarleiter mit meiner Vaadin Grundlagenschulung bei Kunden unterwegs war. Während man mit einem solchen Einsteigerseminar ein recht solides Grundverständnis für dieses spannende Framework schaffen kann, bleiben naturgemäß bei den Teilnehmern auch nach der Schulung noch Detailfragen offen, oder es stellen sich bestimmte Fragen überhaupt erst zu einem späteren Zeitpunkt. Manche Themen können aufgrund der begrenzten Zeit nur oberflächlich behandelt oder, wenn überhaupt, nur grob angerissen werden. Besonders wenn man sich dann als Neuling das erste Mal auf eigene Faust mit dem Framework die Hände schmutzig macht, ergeben sich die meisten Fragen und Verständnisschwierigkeiten. Der Trainer ist dann schon längst wieder zu Hause und steht bei den alltäglichen Problemen, auf die man als Entwickler mit einer neuen Technologie zwangsweise stoßen wird, nicht mehr zur Verfügung.

Wenn man dann den Teilnehmern zusätzlich ein übersichtliches Grundlagenbuch anbieten könnte — das wäre eine feine Sache und würde das Seminar zusätzlich abrunden. Die Idee, ein Buch zu schreiben und meine Kenntnisse und Erfahrungen mit Vaadin darin festzuhalten, wurde daher immer konkreter. Die Anfrage eines Fachbuchverlages, ob ich nicht Interesse daran hätte, ein Vaadin-Buch für Fortgeschrittene zu schreiben, gab mir schließlich den letzten Impuls.

Dem Verlag habe ich nach reiflicher Überlegung abgesagt. Meine Vollzeitstelle bei der OIO GmbH ließ es für mich realistischerweise einfach nicht zu, mit einer bestimmten vorgegebenen Abgabefrist ein Vaadin-Buch mit fortgeschrittenen Themen zu verfassen — und das Ganze auch noch auf Englisch und nicht in meiner Muttersprache. Das war mir dann doch etwas zu heikel, und ich lehnte das Angebot dankend ab.

Die Idee ließ mich jetzt allerdings nicht mehr los. Ich musste eine andere Möglichkeit finden, meine Vaadin-Kenntnisse nutzbringend niederzuschreiben, ohne dabei jedoch dem Druck eines festen Veröffentlichungstermins zu unterliegen.

Leanpub[4] als Plattform für selbstverlegende Autoren kam mir da genau gelegen. Ich kann das Buch in meinem eigenen Tempo voranbringen und auch schon einen halbfertigen Stand veröffentlichen, sobald dieser meiner Leserschaft schon einen Mehrwert bietet. Aus diesem Grund ist das Buch bis zu seiner Fertigstellung ausschließlich unter https://leanpub.com/vaadinbuch zu beziehen.

## Warum dieses Buch?

Ein deutschsprachiges Vaadin-Buch, veröffentlicht ohne Verlag im Rücken, rein elektronisch über eine Online-Plattform vertrieben? Und daneben eine große Konkurrenz auf dem Buchmarkt, nicht zuletzt in Form des offiziellen und kostenlosen *Book of Vaadin* von Vaadin Ltd., dem Hersteller des Frameworks. Wieso mache ich das?

[4] https://leanpub.com/

Nun, der Hauptgrund ist schlicht: ich habe den Anspruch, es besser zu machen als die anderen Autoren. Zudem gibt es auf dem deutschsprachigen Buchmarkt noch keine Vaadin-Bücher, die über eine oberflächliche Einführung in das Framework hinaus gehen. Das will ich ändern.

Fast alle englischsprachigen Vaadin-Bücher, die es zurzeit (aktuell im Januar 2016) auf dem Markt gibt, wurden von Autoren geschrieben, die Englisch nicht als Muttersprache sprechen. Dementsprechend holprig lesen sich die jeweiligen Texte dann zum Teil auch (wohingegen die inhaltliche Qualität der Bücher meist hoch ist). Inhaltlich gehen die verfügbaren Bücher nicht allzu sehr in die Tiefe.

Ausgenommen davon ist das kostenlose *Book of Vaadin*, welches von Marko Grönroos, einem Mitarbeiter von Vaadin Ltd., geschrieben und gepflegt wird. Das Buch zeichnet sich durch eine recht detaillierte Beschreibung des Frameworks aus — was naheliegend ist, da das Buch ja direkt aus dem Hause des Herstellers von Vaadin stammt.

Dennoch gibt es auch für dieses Buch noch Verbesserungspotenzial. Obwohl das Buch recht umfangreich ist und ein sehr weites Themenspektrum abbildet, geht es auf viele Punkte nur relativ oberflächlich oder gar nicht ein (wie z. B. Ressourcen, Konverter oder JavaScript Components). Das ist natürlich auch der Tatsache geschuldet, dass das *Book of Vaadin* sehr viele Themen abdecken muss, die vielleicht nicht für jeden Leser von Relevanz sind. Schließlich besteht auch hier das Problem, dass das englischsprachige Buch nicht von einem Muttersprachler verfasst worden ist und sich damit stellenweise etwas sperrig liest.

Mit dem vorliegenden Vaadin Grundlagenbuch möchte ich der Entwicklergemeinde eine deutschsprachige Alternative zu der vorhandenen Literatur bieten.

## Was will dieses Buch vermitteln?

Dieses Buch verfolgt das Ziel, der Software-Entwicklerin und dem Software-Entwickler eine solide und überschaubare Einführung in das Vaadin Framework zu geben. Mein Hauptanliegen ist es dabei, gerade so viele Themen in dem Buch unterzubringen, wie unbedingt dazu notwendig sind, erfolgreich seine ersten einfachen Anwendungen mit Vaadin zu schreiben.

Das Buch will seine Leser daher nicht mit Informationen überfrachten. Insbesondere soll es nicht notwendig sein, sich erst durch einen mehrere hundert Seiten dicken Trümmer lesen zu müssen, bevor man eine erste einfache Vaadin-Anwendung schreiben kann.

Eines meiner mir gesetzten Ziele als Autor ist es, eine Auswahl an Themen zu treffen, die für den Vaadin-Einstieg unbedingt bekannt sein müssen, bevor eine vernünftige – das heißt wartbare und sauber strukturierte – Vaadin-Anwendung geschrieben werden kann. Genau das sind die Themen, die Sie in diesem Grundlagenbuch vorfinden werden. Themen und Informationen, die in der täglichen Arbeit mit Vaadin zumindest am Anfang noch keine allzu große Relevanz aufweisen und erst in späteren Projektphasen wichtig werden, werden daher ausgespart.

Das angestrebte Ziel des Buches ist es also, eine übersichtliche, aber dennoch nicht zu oberflächliche Einführung in das Thema zu geben. Nachdem Sie dieses Buch gelesen haben, kennen Sie das Vaadin Framework zwar nicht in all seinen Details. Sie wissen dafür aber wie das Framework im Inneren tickt und wie man es grundsätzlich anwendet. Durch diesen Ansatz soll die knappe Zeit, die einem als Entwickler und Entwicklerin zur Verfügung steht, nicht über Gebühr beansprucht werden.

# Was möchte dieses Buch nicht sein?

Dieses Grundlagenbuch will kein detailliertes Referenzwerk zu jedem Detail und jedem Nebenaspekt des Vaadin Frameworks sein. Es soll auch vermieden werden, dass wichtige Grundlagen, die man sich unbedingt aneignen muss, zwischen fortgeschrittenen und zu Beginn irrelevanten Spezialthemen verloren gehen.

Aber was ist, wenn man später, bei der täglichen Arbeit mit dem Framework, an dem Punkt ankommt, an dem die einfachen Grundlagenkenntnisse nicht mehr für das aktuell zu lösende Problem ausreichen? Wenn man wissen muss, wie man die Architektur einer Vaadin-Anwendung am besten gestaltet? Oder wenn man den inneren Aufbau des Frameworks noch besser verstehen muss, um bestimmte knifflige Problemstellungen zu lösen?

Dann wird man zu dem Begleitbuch greifen können, welches ich im Anschluss an dieses Grundlagenbuch plane, wenn dafür bei der Leserschaft Interesse besteht. Dies soll ein Buch für Fortgeschrittene werden, das sämtliche Themen aufgreift und vertieft, die in diesem Grundlagenbuch nicht oder nur einführend behandelt werden konnten. Dieses Buch, *Vaadin für Fortgeschrittene* (dies ist momentan nur der Arbeitstitel), will hinter die Kulissen von Vaadin blicken und die Dinge im Detail erklären. Es sollen Best Practices beschrieben und Ideen und Impulse zu Themen gegeben werden, die in einem etwas weitergefassten Kontext stehen als die reine Oberflächenentwicklung mit Vaadin. Dazu gehören Entwurfsmuster und Optimierungsmöglichkeiten. Daneben sollen Kernbereiche von Vaadin behandelt werden, die zwar wichtig für das tiefere Verständnis des Frameworks sind, die aber dennoch in einem Grundlagenbuch keinen Platz haben. Eines dieser Themen ist beispielsweise die Entwicklung eigener, clientseitiger Vaadin-Komponenten und Komponenten-Extensions mit GWT. Dies ist eines der typischen Ecken des Frameworks, die man zu Beginn nicht unbedingt kennen muss. Man kann problemlos eine ganze Weile sehr produktiv mit Vaadin arbeiten, bevor man in die Verlegenheit kommt, eine eigene UI-Komponente zu entwickeln — wenn das überhaupt geschieht.

# Aufbau des Buches

In diesem Abschnitt wird der Aufbau des Buches und seine Unterteilung in Unterkapitel beschrieben. Es wird grob aufgeführt, welche Kapitel die Leser in diesem Buch vorfinden und welchen Inhalt sie dort erwartet. Dieser Abschnitt wird nachträglich vervollständigt, wenn das Buch fertiggestellt wurde.

# Konventionen

Dieser Abschnitt beschreibt die in diesem Buch verwendeten Konventionen. Das sind die Konventionen für die Darstellung von Quelltext und für Hyperlinks. Auch dieser Abschnitt wird vervollständigt werden, kurz bevor das Buch fertiggestellt wird.

## Quelltext

Sämtliche Quelltexte werden in `nichtproportionaler Schrift` dargestellt. Java-Klassen, die das erste Mal erwähnt werden, werden immer mit ihrem vollqualifizierten Klassennamen ausgeschrieben. Später wird die Package-Information für diese Klassen weggelassen.

Gleiches gilt für Methoden: um zu verstehen, aus welcher Klasse eine angesprochene Methode stammt, wird diese zusammen mit der sie definierenden Klasse angegeben. Es wird dann die vollständige Signatur der Methode inklusive vollqualifiziertem Klassennamen und Typen der Methodenparameter angegeben. Die verwendete Schreibweise orientiert sich dabei an der Referenzierweise von JavaDoc-Kommentaren. Das heißt, eine statische oder nicht-statische Methode wird mit dem `#`-Zeichen von ihrer definierenden Klasse abgetrennt. Folgendes Beispiel soll dies verdeutlichen:

```
com.vaadin.ui.AbstractComponentContainer#addComponent(com.vaadin.ui.Component)
```

bezieht sich auf die Methode, mit der eine Vaadin Komponente auf ein Layout gesetzt werden kann. Im späteren Verlauf des Textes wird dann nur noch von `addComponent()` gesprochen.

Möglicherweise muss aufgrund des begrenzten Platzes auf der Seite eine solche längliche Methodenreferenz künstlich umbrochen werden.

Längere Code-Listings erhalten immer eine Zeilennummerierung. Die Zeilennummern werden im Text aber nicht verwendet, um auf bestimmte interessante Stellen im Code hinzuweisen. Stattdessen werden im betroffenen Quelltext an den Stellen, auf die sich die nachfolgenden Erläuterungen beziehen, nummerierte Kommentare eingefügt. Das sieht dann wie in dem folgenden Beispiel aus:

**Nummerierte Code-Stellen in Form von Kommentaren**

```
1  Label text = new Label();  // {1} //
2  text.setCaption("Beschriftung");
3  text.setValue("Text");     // {2} //
```

In den Erläuterungen zu dem Code-Beispiel wird dann einfach die Nummer der referenzierten Code-Stelle in den Text eingefügt: In dem Code-Beispiel wird ein Label erzeugt {1} und der dargestellte Text des Labels mit `setValue()` gesetzt {2}.

## Kommandozeilenaufrufe

Auch Beispiele von Befehlsaufrufen auf der Kommandozeile werden in `nichtproportionaler Schrift` gesetzt. Um deutlich zu machen, dass Beispielcode einen Aufruf auf der Kommandozeile darstellt, wird diesem für Windows-Befehle, analog zur Windows Command Shell, das Symbol `C:\>` vorangestellt. Für Befehle, die so auch auf unixoiden Systemen ausgeführt werden können, wird das Symbol für den Shell Prompt `$` vorangestellt.

## UML-Diagramme

Sämtliche UML-Diagramme in diesem Buch wurden mit dem Open Source Projekt PlantUML[5] erstellt. PlantUML erlaubt die Erstellung von verschiedenen UML-Diagrammtypen mithilfe einer einfachen, textbasierten Sprache. Die Vererbungshierarchie von `java.util.AbstractList` kann damit zum Beispiel wie folgt dargestellt werden:

[5] http://de.plantuml.com

**Beispiel für ein PlantUML Klassendiagramm**

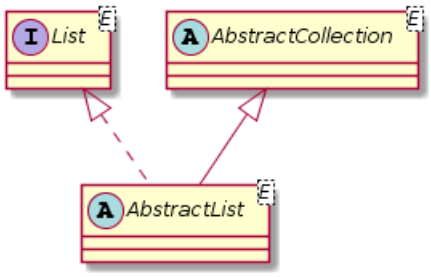
```
@startuml

interface List<E>
abstract class AbstractList<E>
abstract class AbstractCollection<E>

List <|.. AbstractList
AbstractCollection <|-- AbstractList

@enduml
```

Dieses Skript wird von PlantUML zu folgendem Diagramm gerendert:




**Beispiel für ein Klassendiagramm von java.util.AbstractList erzeugt mit PlantUML**

Was man an diesem Beispieldiagramm gut sehen kann, ist die etwas eigenwillige Darstellung von PlantUML für den generischen Typparameter einer Klasse: Der Parameter `E` wird oben rechts an der Klasse in einem weißen Kästchen dargestellt.

# Beispielcode

Der Inhalt dieses Buches wird von vielen, möglichst einfach gehaltenen und verständlichen Codebeispielen begleitet. Natürlich ist es schwierig bis unmöglich, in einem Buch vollständig lauffähige und selbsterklärende Codebeispiele unterzubringen. Es können immer nur Ausschnitte in den Text übernommen werden, die den Kern eines besprochenen Themas beleuchten.

Für jemanden, der sich gerade ganz neu in eine Programmierschnittstelle einarbeiten möchte, ist es jedoch oftmals sehr hilfreich, wenn man kleine, in sich abgeschlossene und lauffähige Beispielprojekte zur Hand hat, die ein bestimmtes, eng umfasstes Thema demonstrieren. Am besten sollte es dazu möglich sein, ein solches Beispielprojekt mit einem einzigen Befehl bauen und ausführen zu können, ohne dazu erst eine umständliche Setup-Prozedur durchführen zu müssen.

Aus diesem Grund wird jeder nicht-triviale Beispielcode aus diesem Buch durch ein isoliert bau- und lauffähiges Demoprojekt ergänzt, das man sich aus dem Netz herunterladen kann und das zu eigenen Experimenten einlädt. Dafür existiert ein GitHub-Projekt "*grundlagenbuch-vaadin7-bsp*"[6], das sämtlichen Beispielcode für dieses Buch enthält.

[6] https://github.com/rolandkrueger/grundlagenbuch-vaadin7-bsp

Die Beispielprojekte basieren auf Java 8 und machen auch Gebrauch von den neuen Features dieser Java-Version. Das heißt, es werden Lambda-Ausdrücke und Streams verwendet, um den Code kurz und bündig zu halten. Insbesondere für Vaadins Ereignisbehandlungsroutinen eignen sich die Lambda-Ausdrücke von Java 8 hervorragend.

Weiterhin verwenden die Projekte alle Apache Maven[7] als Build-Management Tool. Dadurch ist ein sehr einfaches Arbeiten mit den Demoprojekten möglich. Um dies tun zu können, sind keine tieferen Kenntnisse von Maven notwendig. Es genügt, wenn man Maven installiert hat und den notwendigen Befehl kennt, um ein solches Maven Projekt zu starten. Dies wird im Folgenden erläutert.

## Organisation der Beispielprojekte

Sämtliche Beispielprojekte (bis auf das allererste Beispiel, welches mit Eclipse erzeugt und mit Apache Ivy gebaut wird) sind in einem sogenannten *Maven Multi-Module Projekt* organisiert. Das heißt, es gibt ein gemeinsames Hauptverzeichnis, unter dem die einzelnen Demos organisiert sind, und einen gemeinsamen Build Deskriptor, der als Elterndeskriptor alle untergeordneten Projekte zusammenfasst (dieser Build Deskriptor heißt bei Maven *Project Object Model (POM)* und liegt in der Datei `pom.xml`). Über diesen Elterndeskriptor (*Parent POM*) lassen sich sämtliche untergeordnete Projekte mit einem einzigen Kommando bauen und paketieren.

Dennoch ist jedes Unterprojekt dabei für sich genommen selbständig lauffähig. Man kann also in den Verzeichnissen der einzelnen Beispielprojekte den entsprechenden Maven-Befehl zum Bauen und Starten des Projektes absetzen und anschließend die jeweilige Beispielanwendung in seinem Browser besuchen.

## Notwendige Voraussetzungen

Um die Beispielprojekte auszuprobieren, sind ein paar wenige Dinge als Voraussetzung notwendig. Es wird ein JDK ab Version 1.7 benötigt, und es muss das Build-Management Tool Maven in der aktuellen Version (mindestens ab Version 3.0.x) installiert sein (Eine Installationsanweisung für Maven finden Sie im nächsten Abschnitt). Die Installation eines Servlet Containers, wie z. B. Tomcat, ist nicht erforderlich.

Für das erstmalige Bauen der Beispielprojekte wird eine Internetverbindung benötigt. Maven lädt sich sämtliche Abhängigkeiten eines Projektes aus einem zentralen Verzeichnis im Internet (das sogenannte *Maven Central*) herunter und legt diese lokal in einem Cache ab. Zu diesen Abhängigkeiten gehören für unsere Beispiele auch die Vaadin Bibliotheken. Wenn alle notwendigen abhängigen Bibliotheken im Cache verfügbar sind, kann Maven das Projekt fortan auch offline bauen.

## Installation von Maven

Zur Verwendung von Maven muss man sich das für sein Betriebssystem passende Paket von der Maven Homepage[8] herunterladen und dieses lokal installieren.

Die Installation von Maven soll im Folgenden kurz beschrieben werden.

Zuerst laden Sie sich im Download-Bereich der Maven Homepage das Zip-Paket der aktuellen Maven-Version herunter. Diese Zip-Datei entpacken Sie lokal an einen beliebigen Ort,

[7] http://maven.apache.org
[8] http://maven.apache.org

z. B. nach `C:\Programme\apache-maven`. Unter Linux installieren Sie sich das entsprechende Paket über den jeweiligen Paketmanager Ihrer Linux-Distribution.

Anschließend muss eine Umgebungsvariable `M2_HOME` eingerichtet werden, die auf dieses Installationsverzeichnis verweist. Hierbei ist es wichtig zu beachten, dass Sie **nicht** das `\bin`-Verzeichnis der Maven-Installation für diese Umgebungsvariable angeben, sondern das Wurzelverzeichnis der Installation.

```
C:\> set M2_HOME=C:\Programme\apache-maven
```

Unter Linux:

```
$ export M2_HOME=/path/to/maven/installation
```

Anschließend können Sie die `PATH`-Variable des Systems um das Maven `\bin`-Verzeichnis erweitern:

```
C:\> set PATH=%PATH%;%M2_HOME%\bin
```

Unter Linux können Sie einen symbolischen Link auf das Maven Binary `mvn` in `/usr/bin` anlegen.

Wie Sie auf Ihrem System eine Umgebungsvariable fest einrichten, entnehmen Sie bitte der Anleitung Ihres Betriebssystems.

Haben Sie diese Schritte durchgeführt, können Sie in einer neu gestarteten Konsole Ihre Installation testen. Der Befehl

```
$ mvn -version
```

sollte dann zu einer Ausgabe ähnlich der Folgenden führen:

```
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T13:5\
7:37+02:00)
Maven home: C:\Tools\apache-maven-3.3.3
Java version: 1.8.0_11, vendor: Oracle Corporation
Java home: C:\java\jdk1.8.0_11\jre
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
```

Damit haben Sie Maven erfolgreich installiert.

## Bauen und starten der Beispielprojekte mit Maven

Um die Beispielprojekte verwenden zu können, laden Sie sich diese zuerst von GitHub herunter. Das geschieht entweder direkt über das Versionskontrollsystem Git[9] oder indem Sie sich direkt auf der Seite des Projekts auf GitHub (https://github.com/rolandkrueger/grundlagenbuch-vaadin7-bsp) den Quellcode als Zip-Datei herunterladen.

Das Projekt wird wie folgt mit dem Git Kommandozeilen-Client heruntergeladen (“geklont”):

[9] http://git-scm.com

```
$ git clone https://github.com/rolandkrueger/grundlagenbuch-vaadin7-bsp.git
```

> Neben dem offiziellen Git Kommandozeilen-Client gibt es auch eine Reihe grafischer Anwendungen, mit denen man fensterbasiert mit einem Git Repository arbeiten kann. Neben den Git-Integrationen, die alle modernen IDEs mit an Bord haben (z. B. das EGit Plugin für Eclipse), gibt es auch das kostenlose Programm *SourceTree*[a] von Atlassian, das unabhängig von einer bestimmten Entwicklungsumgebung arbeitet. SourceTree ermöglicht ein bequemes, grafisches Arbeiten mit Git und bietet einen übersichtlichen Blick auf sein Git Repository. Das Programm kann kostenfrei genutzt werden; man muss sich allerdings bei der ersten Verwendung mit einer Email-Adresse bei Atlassian registrieren.
>
> [a] https://www.sourcetreeapp.com/

Das Projekt *grundlagenbuch-vaadin7-bsp* steht übrigens unter einer Public Domain Lizenz. Das heißt, Sie dürfen mit dem Beispielcode anstellen, was Sie wollen, ohne mich vorher um Erlaubnis fragen zu müssen. Nutzen Sie den Code also gerne als Basis, Inspiration oder als Kopiervorlage in Ihren eigenen Projekten.

Die auf Maven basierenden Beispielprojekte für dieses Vaadin Grundlagenbuch finden Sie in dem geklonten Projekt unter `/Maven-Projekte`. Wechseln Sie in dieses Verzeichnis und führen Sie dort den folgenden Befehl aus:

```
$ mvn package
```

Alle Beispielprojekte werden dadurch gebaut und können anschließend mit Maven über den integrierten Jetty Server gestartet werden. Wechseln Sie dazu in ein beliebiges Unterverzeichnis unterhalb von `/Maven-Projekte` und führen Sie den folgenden Befehl aus:

```
$ mvn jetty:run
```

Das einzige Projekt, das nicht mit Maven gebaut werden kann, ist `/Kap02_HelloVaadin`. Dies ist ein reines Eclipse-Projekt, das wir im Kapitel “Hello Vaadin” erstellen werden.

Wenn Sie im vorigen Schritt die Demoprojekte erfolgreich gebaut haben, wird nun ein Jetty Server gestartet, auf dem das Projekt automatisch deployt wird. Ist der Jetty gestartet, können Sie das mit Ihrem Browser die Seite

```
http://localhost:8080
```

besuchen. Es öffnet sich daraufhin das jeweilige Demoprojekt in Ihrem Browser, und Sie können beginnen, damit zu experimentieren.

# Feedback

Ich habe mich für Leanpub als Veröffentlichungsplattform für dieses Buch entschieden, da man hier sein Buch schon für die Allgemeinheit freigeben kann, während man noch daran schreibt. Dieses Prinzip erlaubt es, schon frühzeitig auf Feedback, Rückmeldungen und Leserwünsche einzugehen, und diese in die Entstehung des Buches einfließen zu lassen.

Damit das für dieses Buch richtig funktioniert, bin ich auf Ihre Rückmeldung angewiesen. Ich freue mich also über jede Art von Kommentaren, Wünschen und Meinungen, die ich von Ihnen bekommen kann. Sie wünschen sich die bevorzugte oder vertiefendere Behandlung eines Themas, das für Sie besonders interessant ist? Schreiben Sie mir das, und ich werde versuchen, Ihre Wünsche in die Kapitelplanung einfließen zu lassen.

Sie können mich über verschiedene Kanäle erreichen: entweder über die oben genannten sozialen Netzwerke (Twitter oder Google+), per Email (mail@rolandkrueger.info) oder über die Kommentarspalte zu diesem Buch auf Leanpub.

# Copyrights und Bildnachweise



Die Abbildungen in diesem Buch wurden inspiriert durch die bikablo® Publikationen, www.bikablo.com[10].

[10] http://www.bikablo.com

# Teil 1: Einleitung und erste Schritte

In diesem Teil des Buches wollen wir unsere ersten Gehversuche mit dem Vaadin Framework unternehmen. Bevor wir uns das obligatorische *Hello World* Beispiel (oder vielmehr *Hello Vaadin*) anschauen, werfen wir zuerst einen kurzen Blick auf die Geschichte des Frameworks und auf dessen technologischen Hintergrund. Wir wollen erfahren, wo das Framework herkommt und welche Merkmale charakteristisch für das Framework sind.

Nachdem wir das Vaadin Framework technologisch eingeordnet haben, stürzen wir uns sogleich in die Anwendungsentwicklung selbst und schreiben die einfachst mögliche Vaadin-Applikation an, die man sich vorstellen kann (Gut, nicht ganz. Man kann sich eine noch einfachere Vaadin-Anwendung vorstellen, aber eine solche würde uns definitiv unterfordern): die allseits beliebte *Hello World* Anwendung. Wir werden dort die wichtigsten Bestandteile einer Vaadin-Anwendung und die dazugehörige Grundkonfiguration kennenlernen.

Am Ende des ersten Buchteils haben wir uns ein grundlegendes Verständnis dafür angeeignet, wie eine Vaadin-Applikation aufgebaut ist und welche Besonderheiten eine Vaadin-Anwendung auszeichnen. Aus den speziellen Eigenschaften von Vaadin ergeben sich eine Reihe von ganz bestimmten Vor- und Nachteilen. Diese werden wir kennenlernen, sodass wir eine informierte Entscheidung darüber treffen können, in welcher Situation das Vaadin Framework eine gute Wahl ist und wo eher nicht.

# 1. Was ist Vaadin?

Lassen Sie uns zum Einstieg zunächst einen Blick auf die geschichtlichen und technologischen Hintergründe des Frameworks werfen. Wir wollen erfahren, wo Vaadin herkommt und welche Besonderheiten das Framework auszeichnen.

Der Blick auf diese technologischen Besonderheiten gibt uns eine Idee davon, wie wir das Verhalten des Frameworks zur Laufzeit bewerten können. Anhand der speziellen Vor- und Nachteile, die sich daraus ergeben, können wir Szenarien ableiten, für die der Einsatz von Vaadin besonders geeignet, oder für die er eher weniger geeignet ist.

Falls Ihre Neugier auf das Vaadin Framework zu groß ist und Sie sich lieber gleich in die Programmierung mit Vaadin stürzen wollen, können Sie dieses Kapitel getrost überspringen und im nächsten Kapitel einsteigen. Es lohnt sich aber sicherlich, später noch einmal hierher zurückzukehren, um sich ein wenig mit der Herkunft von Vaadin zu beschäftigen.

## 1.1 Geschichtliches

Das Vaadin Framework ist unter diesem Namen seit dem Jahr 2009 auf dem Markt. Man möchte fast meinen, dass Vaadin damit ein noch recht junges und vielleicht noch nicht an jeder Stelle vollständig ausgereiftes Framework ist. Ich habe Ihnen aber nur die halbe Wahrheit erzählt.

In Wahrheit gibt es Vaadin, beziehungsweise dessen technologischen Vorgänger, schon seit dem Jahr 2000. Seit diesem Jahr wird nämlich von der finnischen Firma IT Mill Ltd. ein Webframework namens Millstone[1] entwickelt, das später einmal in unser Lieblingsframework Vaadin umgewandelt werden soll.

Die Firma IT Mill Ltd. gibt es heute unter diesem Namen nicht mehr. Sie wurde im Jahr 2009 in Vaadin Ltd. umbenannt – zeitgleich zu der Umbenennung des hauseigenen Frameworks nach Vaadin. Man wollte mit dieser Umbenennung den Wunsch unterstreichen, stärker die Community einzubeziehen.

Vaadin Ltd. ist in der südfinnischen Stadt Turku (oder auf Schwedisch Åbo) ansässig. Inzwischen gibt es aber auch schon Zweigstellen in Berlin und San Jose, USA. Firmengründer und Geschäftsführer ist Dr. Joonas Lehtinen, der mit seinem Team das Java Webframework entwickelt.

Vaadin ist ein Open Source Framework, das unter der freizügigen Apache License 2.0 steht. Die Firma Vaadin bietet neben dem kostenlosen Framework auch kommerzielle Dienstleistungen in den Bereichen Beratung, Projektunterstützung und Framework Schulungen an. Daneben gibt es einige kostenpflichtige Erweiterungen und Zusatztools für Vaadin, die das Framework um neue Funktionen und Komponenten erweitern. Beispielhaft aus diesem Bereich seien hier das Vaadin Touchkit zum Schreiben mobiler Anwendungen, der Vaadin Designer zur visuellen Gestaltung von Benutzeroberflächen oder die Vaadin Charts, eine Sammlung von Statistik- und Graph-Komponenten, erwähnt. Besuchen Sie für einen Überblick über diese Dienstleistungen und Angebote einfach die Vaadin Homepage http://www.vaadin.com[2].

[1] engl. für Mühlstein

[2] http://www.vaadin.com

Mit Vaadin lassen sich Webanwendungen mit rein serverseitigem Code erstellen. Das heißt, man programmiert eine Webanwendung vollständig mit Java und bewegt sich dabei komplett auf der Server-Seite. Der Teil, der den Benutzerinnen und Benutzern im Browser angezeigt wird, wird auch im Browser erzeugt. Dieser Prozess wird allerdings vom Server aus ferngesteuert. Anders als bei anderen Technologien, die z. B. auf HTML Templating aufsetzen, generieren wir hierbei HTML Markup nicht direkt. Dies wird für uns vom Framework übernommen. Wir stellen lediglich UI-Komponenten auf Layouts zusammen, die dann von Vaadin in das HTML Dokument "gezeichnet" werden. Man muss sich dadurch nicht mehr zwingend mit Technologien, wie HTML, CSS, Templating Engines oder JavaScript herumschlagen.

Das Vaadin Framework verwendete ursprünglich (als es noch Millstone Framework hieß) ein proprietäres, AJAX-basiertes Kommunikationsmodell zwischen der Client- und der Server-Seite. Die clientseitige Render Engine, der Teil also, der die UI Komponenten in den Browser zeichnet, und die Client-Server-Kommunikation waren Eigenentwicklungen. In dieser Version war es nur sehr schwer möglich, das Framework um eigene clientseitige Komponenten zu erweitern. Eine bessere Alternative für den proprietären Client-Teil musste gefunden werden.

## Integration des Google Web Toolkits

Im Jahr 2006 wurde von Google das Google Web Toolkit (GWT oder auch *Gwit* ausgesprochen) veröffentlicht. Das GWT ist ein Framework, das es Entwicklerinnen und Entwicklern ermöglicht, JavaScript-Anwendungen für den Browser rein mit Java zu entwickeln. Das funktioniert über einen sogenannten *Cross Compiler*. Das heißt, man schreibt mit Hilfe einer speziellen Programmierschnittstelle Java Code, der später mit dem GWT Compiler nach JavaScript übersetzt wird. Der GWT Cross Compiler arbeitet dabei direkt mit dem Java Source Code und nicht mit den kompilierten Class-Dateien. Am Ende wird der übersetzte JavaScript-Code im Browser ausgeführt.

Das Besondere am Google Web Toolkit ist, dass die komplette Webanwendung – sprich die gesamte Fachlogik – im Browser läuft. Es kann (muss aber nicht) mit Hilfe von *Remote Procedure Calls* (entfernten Prozeduraufrufen) mit einem Server-Backend kommuniziert werden, um bspw. Daten aus einer Datenbank nachzuladen oder diese dort zu persistieren.

Eine weitere Besonderheit ist, dass das GWT JavaScript-Code erzeugt, der genau auf die Eigenheiten der einzelnen Browser abgestimmt ist. Das bedeutet, dass für jede gewünschte Browser-Zielplattform genau eine spezielle JavaScript-Datei kompiliert wird, die nur von dem jeweiligen Browser geladen wird. Man kann damit sichergehen, dass seine Anwendung ohne gesonderte Anpassung auch in jedem Browser problemlos funktioniert.

Ein solches clientseitiges Framework, das browserabhängigen JavaScript-Code auf Basis von Java-Quellcode erstellen kann, ist natürlich der ideale Kandidat für den Client-Teil von Vaadin. GWT-Code kann komplett mit Java geschrieben werden, was gut zu einem Java-Framework wie Vaadin passt. Und über die Möglichkeit von GWT, auch per Remote Procedure Calls (RPC) mit einem Server-Backend kommunizieren zu können, kann eine GWT-Anwendung problemlos an den serverseitigen Teil von Vaadin – in dem ja die Fachlogik läuft – angebunden werden.

Im Jahr 2007 wurde der proprietäre clientseitige Teil von Vaadin über Bord geworfen und durch eine GWT-Implementierung ersetzt. Dies brachte einige große Vorteile mit sich:

- Das Vaadin-Team muss sich nun nicht mehr um die Weiterpflege einer eigenen clientseitigen Implementierung kümmern und kann sich voll und ganz auf die Verbesserung des eigentlichen Vaadin-Kerns konzentrieren.

- Vaadin wurde komplette browserunabhängig, ohne dass dafür ein gesonderter Aufwand betrieben werden musste.
- Die Weiterentwicklung des clientseitigen Basisframeworks (GWT) erfolgt durch das Team eines großen Herstellers, nämlich Google. Vaadin hat hier keine großen Aufwände mehr.
- Die Entwicklung clientseitiger Komponenten kann von jedermann durchgeführt werden, der sich mit dem Google Web Toolkit auskennt. Eine Einarbeitung in eine proprietäre Programmierschnittstelle ist damit nicht mehr notwendig. Vorhandenes Wissen kann also zum Einsatz kommen, sodass auf dem Arbeitsmarkt auch eine größere Menge an potentiellen Entwicklerinnen und Entwicklern für den Einsatz in Vaadin-Projekten verfügbar ist.

## 1.2 Technologischer Hintergrund

Seit 2007 basiert also der clientseitige Teil von Vaadin auf dem Google Web Toolkit. Was bedeutet das aus technischer Sicht?

Eine typische GWT-Anwendung besteht aus einer Reihe von Ansichten (*Views*), die über eine spezielle Fachlogik miteinander verbunden sind. Damit behandelt eine GWT-Anwendung immer ein ganz bestimmtes, klar umrissenes fachliches Thema, z. B. eine Kundendatenverwaltung oder einen Web Shop. Wie kann man dies nun mit einer Technologie wie Vaadin zusammenbringen, bei der wir eine Webanwendung rein serverseitig implementieren wollen? Wir wollen ja gerade nicht dazu gezwungen werden, zusätzlich noch clientseitigen Code schreiben zu müssen.

Die Lösung ist, dass Vaadin eine generische GWT-Anwendung für den Client-Teil mitbringt, mit der sich jegliche Anwendungslogik umsetzen lässt, ohne dass dieser clientseitige Teil von uns angepasst werden müsste. Diese generische GWT-Anwendung bildet dabei keine bestimmte, anwendungsfallbezogene Fachlogik ab, sondern stellt eine Schnittstelle dar, welche UI-Komponenten im Browser darstellen und mit dem Vaadin-Backend kommunizieren kann.

Diese generische GWT-Anwendung wird in der Vaadin-Terminologie *Widget Set* (oder auch *Client-Side Engine*) genannt. In diesem Widget Set ist die Menge aller verfügbaren Vaadin UI-Komponenten vorhanden. Zudem enthält es ein Kommunikationsmodul, mit dessen Hilfe dieses Widget Set mit der serverseitigen Anwendung kommunizieren kann.

Das Grundprinzip einer Vaadin-Anwendung sieht damit wie folgt aus. Wir stellen in unserem serverseitigen Code eine Benutzerschnittstelle zusammen, indem wir UI-Komponenten (Textfelder, Checkboxen, Tabellen etc.) auf Layout-Komponenten anordnen. Es ergibt sich dadurch eine baumartige Struktur von instanziierten UI-Objekten, die innerhalb der HTTP Session eines Benutzers (also im Speicherbereich des Servers) verwaltet wird. Vaadin kümmert sich nun für uns hinter den Kulissen darum, dass an den Browser spezielle Anweisungen geschickt werden, welche vom Widget Set interpretiert und ausgeführt werden können. Diese Anweisungen führen dazu, dass das Widget Set die UI-Komponenten, die wir auf Server-Seite zusammengestellt haben, in Form von HTML Markup in den DOM-Baum (*Document Object Model*) gezeichnet werden. Wir steuern also mit unserem serverseitigen Code den Vaadin-Teil im Browser indirekt fern.

Was geschieht als nächstes, wenn die Benutzerin oder der Benutzer mit der Vaadin-Anwendung im Browser interagiert? Interaktion bedeutet bei Vaadin immer, dass in irgendeiner Form ein Ereignis ausgelöst wird. Das kann geschehen, wenn auf eine Schaltfläche geklickt wird, oder wenn ein Textfeld den Eingabefokus verliert, oder wenn man die Sortierung einer

Tabelle ändert und so weiter. Das Widget Set sorgt nun dafür, dass derartige Ereignisse (*Events*) registriert und an den Server geschickt werden. Dort werden diese Ereignisse in Form von komponentenspezifischen Events (z. B. ein *Button.ClickEvent* wenn auf eine Schaltfläche geklickt wurde) von unserem Anwendungscode weiterverarbeitet.

Auf jedes Event folgt typischerweise eine anwendungsspezifische Reaktion. Zum Beispiel können Daten aus einer Datenbank nachgeladen werden, die in einer Tabelle oder einer Liste angezeigt werden sollen. Oder es werden Formulareingaben in der Datenbank persistiert, und auf der Benutzeroberfläche wird ein entsprechender Hinweistext angezeigt. In den allermeisten Fällen muss also in irgendeiner Weise die Benutzeroberfläche angepasst werden. Als Antwort auf ein solches Anwendungsevent manipulieren wir also in unserem serverseitigen Code die UI-Komponenten, auf die wir über die HTTP Session Zugriff haben. Beispielsweise ändern wir den Text, der von einem bestimmten Label angezeigt wird. Diese Änderungen an der dargestellten Komponentenhierarchie werden von Vaadin als Antwort auf ein Ereignis an das Widget Set im Browser zurückgeschickt. Dieses sorgt dafür, dass die clientseitig dargestellten UI-Komponenten entsprechend aktualisiert werden. Dafür werden einfach die jeweiligen HTML-Elemente im DOM-Baum angepasst, entfernt, oder es werden neue hinzugefügt.

Die Ereignisbehandlung geschieht bei Vaadin über AJAX-ähnliche Aufrufe. Die Abkürzung AJAX steht, wie Sie sicherlich wissen, für *Asynchronous JavaScript and XML*, also asynchrone JavaScript-Verarbeitung, bei der Nachrichten im XML-Format verschickt werden. Die Idee hinter AJAX-basierten Webanwendungen ist, dass Daten zwischen Server und Browser ausgetauscht und der Inhalt einer aktuell angezeigten Seite verändert werden kann, ohne dass die Seite komplett neu geladen werden muss. Vaadin arbeitet nach demselben Prinzip: Es wird eine einzige Seite geladen (die sogenannte *Host Page*), die als erstes den JavaScript-Code des Widget Sets lädt und ausführt. Anschließend werden nur noch Teile der Seite ereignisgetrieben abgeändert oder ausgetauscht. Die Kommunikation mit dem Server geschieht dabei im Hintergrund. Es werden hierbei nur die Daten zwischen Client und Server ausgetauscht, die zur Bearbeitung des aktuellen Events notwendig sind. Ein Neuladen der Seite findet nicht statt.

Es gibt bei Vaadin zwei wichtige Unterschiede zum klassischen AJAX: erstens ist die Kommunikation von Vaadin nicht asynchron. Wenn ein Ereignis zum Server geschickt wurde, wartet das Widget Set zuerst auf eine Antwort, bevor die Anwendung weiter benutzt werden kann. Zumeist fällt das gar nicht weiter auf, da die Ereignisbehandlung so schnell abläuft, dass die Benutzeroberfläche nur für den Bruchteil einer Sekunde blockiert ist — so schnell kann man gar nicht klicken. Wenn die Ereignisbehandlung allerdings besonders lange dauert, etwa weil eine langlaufende Berechnung angestoßen wurde, dann wird dies für die Anwenderin oder den Anwender spürbar.

Zweitens verwendet Vaadin keine XML Nachrichten für die Kommunikation mit dem Server. Stattdessen setzt das Framework auf JSON[3]-kodierte Nachrichten, welche durch JavaScript-Code wesentlich besser interpretierbar und verarbeitbar ist als XML.

Zusammenfassend können wir festhalten, dass eine Vaadin-Anwendung einmalig über eine Host Page geladen und dann nur noch über leichtgewichtige Serveranfragen gesteuert wird. Die gesamte Anwendung wird dabei auf einer einzigen Seite betrieben — nämlich die Host Page, die beim ersten Aufruf der Vaadin-Anwendung geladen wurde. Aus diesem Grund spricht man bei einer solchen Art von Anwendung auch von einer *Single-Page Application*: es wird eine Seite geladen, die den notwendigen JavaScript-Code nachlädt, der dann im Folgenden dynamisch die

[3] JavaScript Object Notation

Inhalte dieser Seite austauscht.

Durch diese Eigenschaften und durch die damit möglichen dynamischen UI-Komponenten (z. B. Menüleisten, Drop-Down Vorschlagslisten, Dialogfenster, Möglichkeiten für Drag-and-Drop-Verarbeitung, Baumkomponenten usw.) zeigen derartige Anwendungen ein ähnliches Verhalten, wie herkömmliche Desktop-Applikationen. Sie zeigen ein *reichhaltiges* Verhalten mit Bezug auf die Interaktionsmöglichkeiten, und das Ganze findet im Browser über das Internet statt. Aus diesem Grund werden derartige Anwendungen auch *Rich Internet Applications* genannt.

Damit haben wir die Grundeigenschaften von Vaadin herausgearbeitet: Vaadin ist ein Framework für Rich Internet Applications, welches nicht auf einer Browser-Plugin-Infrastruktur aufsetzt, sondern nach dem Prinzip einer JavaScript Single-Page Application funktioniert.

# 2. Hello Vaadin!

Lassen Sie uns nun auf unsere Reise durch die Welt von Vaadin begeben. Wie es in der Literatur für Computerthemen so üblich ist, wollen auch wir hier mit einem einfachen *Hello World* Beispiel beginnen. Mit dessen Hilfe werden wir die grundlegendsten Konzepte des Frameworks kennenlernen.

In diesem Kapitel werden wir eine erste, sehr einfache Anwendung schreiben, die schon die wesentlichen Grundbestandteile einer jeden Vaadin-Applikation enthalten wird. Wir begnügen uns hierbei nicht damit, einfach nur ein simples Label mit dem Text "*Hello World*" auf den Bildschirm zu bringen. Wir wollen uns gleich anständig mit Namen begrüßen lassen und sehen deshalb die Eingabe unseres Namens in ein Textfeld vor. Diese Eingabe müssen wir über eine Schaltfläche bestätigen. Anschließend werden wir auf eine Seite umgeleitet, auf der wir mit Namen begrüßt werden.

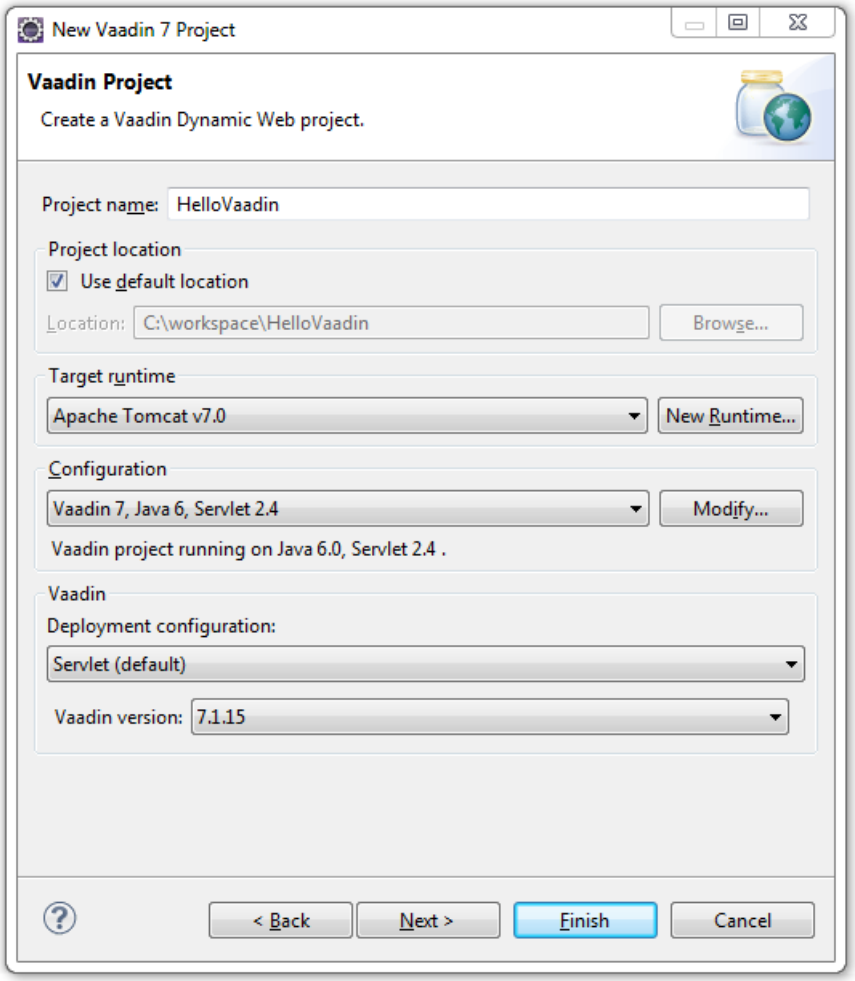
Damit lernen wir die drei wichtigsten Grundelemente der Programmierung von Vaadin-Anwendungen kennen: die validierte Eingabe von Daten, die Ereignisbehandlung und die Seitennavigation innerhalb einer Anwendung.

Anschließend verschaffen wir uns einen Überblick über die wichtigsten Eigenschaften von *Rich Internet Applications*, zu denen ja mit Vaadin geschriebene Anwendungen gehören, wie wir im letzten Kapitel erfahren haben.

## 2.1 Anlegen eines Projekts

Wir beginnen mit einem neuen Eclipse-Projekt, das wir `HelloVaadin` nennen wollen. Für dieses Beispiel verwenden wir das Vaadin Plugin für Eclipse, das wir zuvor über den *Eclipse Marketplace* installiert haben.

Um ein neues Vaadin Projekt anzulegen, verwenden wir den Wizard für Vaadin Projekte, den wir unter *File* → *New* → *Other...* → *Vaadin* → *Vaadin 7 Project* finden. Der erste Dialog des Wizards erlaubt einige grundlegende Einstellungen.




**Anlegen eines neuen Vaadin 7 Projektes mit dem Vaadin Eclipse Plugin**

Zunächst können wir den Namen unseres neuen Projektes festlegen. Wir geben hier `HelloVaadin` ein. Unter *Project location* wählen wir den Zielort der Projektdateien. Wir lassen das Projekt einfach in unseren Eclipse Workspace legen.

Als nächstes wählen wir bei der *Target runtime* den Servlet Container aus, auf dem wir unser Projekt laufen lassen wollen. Wir müssen in Eclipse unter *Window* → *Preferences* → *Server* → *Runtime Environment* mindestens eine Server-Laufzeitumgebung, z. B. einen Tomcat Server, eingerichtet haben. Wir können hier eine dieser Server-Konfigurationen auswählen. Sollte in diesem Dialog noch kein Server auswählbar sein, können wir die Option auch einfach überspringen. Unser Projekt lässt sich ohne Probleme auch später noch auf einer Server-Umgebung ausführen.

Im nächsten Feld wählen wir die Konfiguration unseres Projekts. Wir haben hier die Möglichkeit, die Version der verwendeten Servlet-Spezifikation zu wählen. Welche Version wir auswählen hängt im Wesentlichen davon ab, welche Features der Servlet-Spezifikation wir benötigen und welche Servlet-Version unsere Server-Zielplattform unterstützt. Für unser einfaches *Hello World*-Beispiel genügt uns die Version 2.4 der Servlet-Spezifikation. Über die Schaltfläche *Modify...* können wir die vorhandenen Konfigurationsmöglichkeiten anpassen. Beispielsweise lässt sich hier die verwendete Java Version ändern. Wir definieren damit die aktiven Eclipse Projekt-Facetten unseres Projekts.

Zu guter Letzt legen wir die *Deployment configuration* und die verwendete Vaadin-Version für unser Projekt fest. Mit der *Deployment configuration* definieren wir, in welcher Umgebung unsere Vaadin-Anwendung später einmal betrieben werden soll. Es gibt dazu drei Möglichkeiten, aus denen man wählen kann:

- **Servlet**: Dies ist die Standardkonfiguration. Man wählt diese, um seine Vaadin-Anwendung als herkömmliche Java Webanwendung in einem Servlet Container, wie Tomcat, Jetty oder GlassFish zu betreiben.
- **Google App Engine servlet**: Alternativ kann man seine Anwendung auch in der Google App Engine[1] installieren. Die App Engine ist ein PaaS-Dienst (*Platform-as-a-Service*) von Google. Man kann damit seine Vaadin-Anwendung sehr einfach in der Cloud betreiben.
- **Generic portlet (Portlet 2.0)**: Schließlich ist es auch möglich, Vaadin als Portlet in einem Portal Server laufen zu lassen. Damit bietet es sich z. B. an, Portlets für das Liferay Portal[2] mit Hilfe von Vaadin zu implementieren.

Die Konfiguration, die wir hier wählen, legt fest, welches konkrete Vaadin-Servlet in unserem Projekt eingesetzt wird. Abhängig von der Laufzeitumgebung unserer Anwendung muss ein spezielles Servlet verwendet werden. So gibt es eine eigene Servlet-Implementierung für die Google App Engine und für Portal Server.

Wir können uns später jedoch jederzeit für eine andere Laufzeitumgebung entscheiden. Vaadin ist so konzipiert, dass der Programmcode einer Anwendung vollständig von der konkreten Umgebung abstrahiert ist. Egal ob wir unsere Anwendung in die Google App Engine oder in einem herkömmlichen Servlet Container installieren — unser Anwendungscode bleibt davon unberührt (zumindest der Vaadin-Teil). Einzig das darunterliegende Servlet muss angepasst werden.

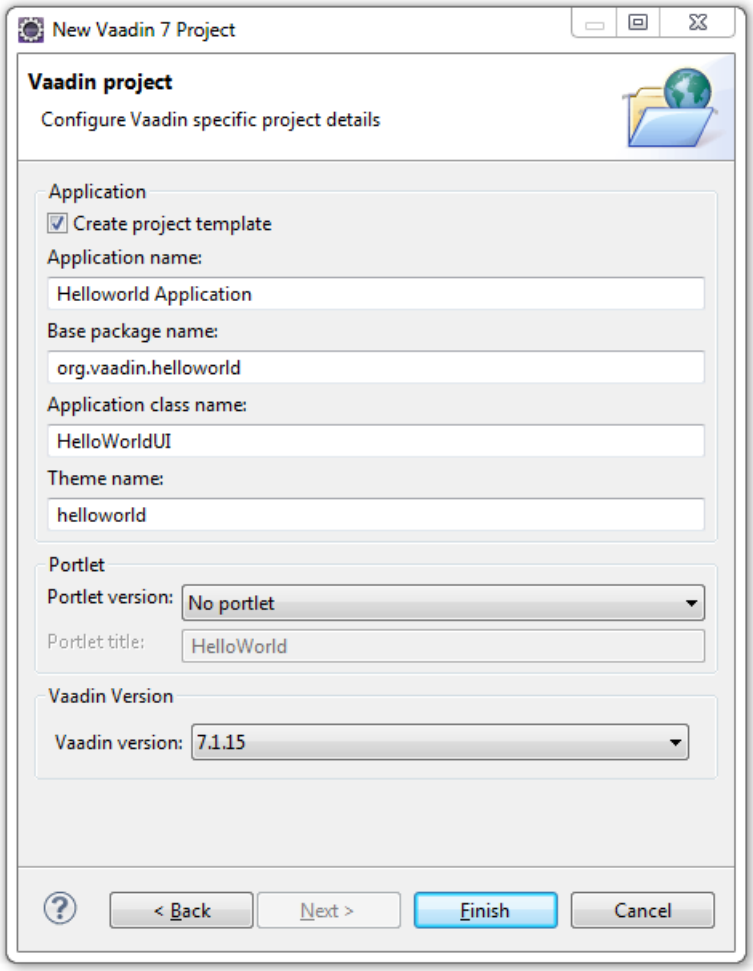
Nachdem wir nun die Grundkonfiguration für unser Vaadin-Projekt festgelegt haben, können wir auf *Next* klicken, um auf dem nächsten Dialog das Quellcode- und Ausgabeverzeichnis festzulegen. Hier können wir die Standardeinstellungen beibehalten.

Ein weiterer Klick auf *Next* bringt uns auf den Konfigurationsdialog für das Web-Modul. Hier legen wir den *Context Root* unserer Anwendung fest. Das ist der Pfad in der URL zu unserer Anwendung. Wenn wir hier also die Voreinstellung `HelloWorld` beibehalten, wird unsere Anwendung später lokal unter der URL `http://localhost:8080/HelloWorld` erreichbar sein.

Das *Content Directory*, das wir als nächstes festlegen können, bezeichnet dasjenige Verzeichnis in unserem Projekt, das sämtliche Web-Ressourcen enthalten wird. Unter diesem Pfad befindet sich das `WEB-INF`-Verzeichnis und die `web.xml` (falls vorhanden). Den vorgegebenen Namen `WebContent` können wir auch hier beibehalten. Für unser erstes Vaadin Projekt setzen wir den Haken bei *Generate web.xml deployment descriptor*. Damit wird für uns von dem Eclipse Plugin die Grundkonfiguration des *Deployment Descriptors* generiert. Wir können diesen später nach unseren eigenen Wünschen anpassen. Falls wir auf der ersten Seite des New Project Wizards eine Servlet-Version größer gleich 3.0 angegeben haben, können wir auch auf die `web.xml` verzichten und das Servlet im Code rein über Annotationen konfigurieren.

Mit einem Klick auf *Next* kommen wir auf die letzte Dialogseite des Wizards.

[1] https://appengine.google.com
[2] https://www.liferay.com




**Letzte Dialogseite des Wizards für neue Vaadin 7 Projekte**

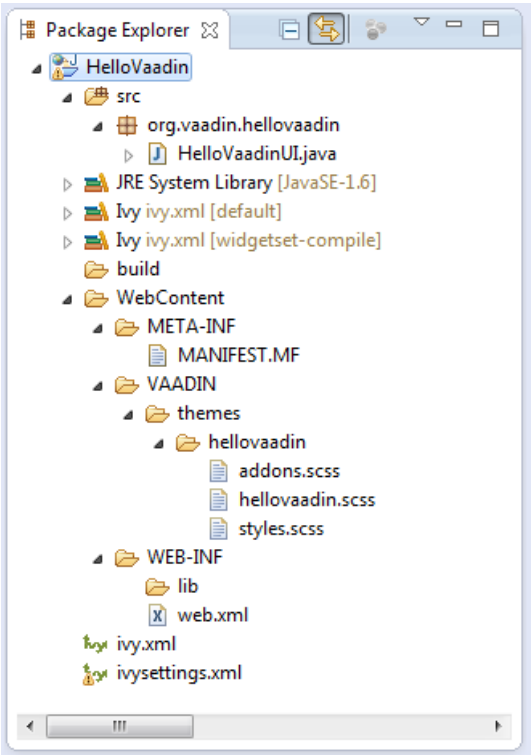
Dort haken wir die Option *Create project template* an. Damit wird für uns die Package-Struktur des Projekts und die Vaadin Hauptklasse erzeugt. Als nächstes geben wir die folgenden Informationen an:

- *Application name*: Name der Anwendung. Dieser Wert wird später in der `web.xml` als Servlet-Name für das Vaadin Servlet verwendet.
- *Base package name*: Name des Basis-Packages, unterhalb dessen unsere Anwendung liegt.
- *Application class name*: Klassenname für die Hauptklasse der Anwendung. Typischerweise heißt diese Klasse wie unsere Anwendung mit angehängtem `UI`, also z. B. `HelloWorldUI`. Wenn wir den Haken bei `Create project template` gesetzt haben, wird diese Klasse für uns automatisch angelegt.
- *Theme name*: Name des *Themes*, das von unserer Anwendung verwendet werden soll. Ein Theme definiert das optische Erscheinungsbild einer Vaadin-Applikation. Wir können hier einen Namen vorgeben, der zu unserer Anwendung passt. Was es mit Themes auf sich hat, werden wir später erfahren. Wir können es an dieser Stelle erst einmal so hinnehmen, dass für uns ein eigenes Theme erzeugt wird.

Schließlich können wir noch die Version der verwendeten Portlet-Spezifikation wählen, falls unsere Vaadin-Anwendung als Portlet betrieben werden soll. Diesen Punkt benötigen wir für

unser Beispiel nicht. Als letzten Konfigurationswert legen wir die Vaadin-Version fest, die für unser Projekt verwendet werden soll. Hier können wir einfach die jeweils aktuelle Vaadin-Version wählen.

Als letztes drücken wir nun den *Finish*-Knopf und lassen Eclipse unser bis hierher konfiguriertes Projekt erzeugen. Es wird ein Eclipse-Projekt generiert, das Ivy als Build Tool verwendet. Die Eclipse Ivy Integration sorgt anschließend dafür, dass alle benötigten Abhängigkeiten (d. h. die Vaadin Bibliotheken und deren abhängigen Libraries) aufgelöst und heruntergeladen werden. Mit den vorhandenen Abhängigkeiten kann das Projekt von Eclipse fehlerfrei kompiliert werden. Als Ergebnis haben wir jetzt ein neues Projekt namens *HelloVaadin* in unserem Workspace. Schauen Sie sich gerne ein wenig in den Projektdateien um.




**Das Projektlayout unseres brandneuen Hello Vaadin Projekts**

Im Folgenden wollen wir einen kurzen Blick auf die erzeugten Projektdateien werfen. Wir werden dabei die wichtigsten Grundbestandteile einer Vaadin-Anwendung umreißen.

## Die Projektdateien

Schauen wir uns einige der generierten Dateien etwas genauer an.

### HelloVaadin/src/org/vaadin/hellovaadin/HelloVaadinUI.java

Diese Datei beherbergt die Klassendefinition der Hauptklasse unseres Projekts. Die Klasse `HelloVaadinUI` erbt von der kurz und knapp benannten Vaadin Klasse `com.vaadin.ui.UI`. Dies ist die Einstiegsklasse für eine Vaadin-Applikation. In ihr muss vor allem die abstrakte Methode `UI#init(VaadinRequest)` implementiert werden. Über dieser Methode wird eine Vaadin-Anwendung initialisiert. Sie wird immer dann aufgerufen, wenn eine Benutzerin oder ein Benutzer unsere Anwendung mit dem Browser besucht.

Man kann sich die `init()`-Methode ungefähr wie die `main()`-Methode eines herkömmlichen Java-Programms vorstellen. Hier wird der Einstiegspunkt einer Anwendung definiert. Die Hauptaufgabe dieser Methode ist es also, die Benutzeroberfläche der Anwendung zu initialisieren und die initialen Ereignisbehandlungsroutinen für die Verarbeitung der Benutzeraktionen zu registrieren.

## HelloVaadin/WebContent/VAADIN/themes/hellovaadin/*

In diesem Verzeichnis befindet sich das sogenannte *Theme* einer Vaadin-Anwendung. Ein Theme legt das äußere Erscheinungsbild der Anwendung fest. Hier können wir das *Look & Feel* für unser Programm definieren. Im Wesentlichen besteht das aus speziell angepassten *Cascading Stylesheets* (CSS) und weiteren Ressourcendateien, wie z. B. Bilder und Icons.

Das Vaadin Eclipse Plugin hat schon ein solches Theme für uns angelegt. Dieses dient allerdings vorerst nur als Vorlage für unsere eigenen Anpassungen, d. h. das Theme ist zu Beginn leer. Belassen wir die Theme-Dateien so wie sie im Moment sind, verhält sich unsere Vaadin-Anwendung genauso, als hätten wir kein eigenes Theme definiert. Wir brauchen uns also vorerst gar nicht um diese Dateien zu kümmern.

## HelloVaadin/WebContent/WEB-INF/web.xml

Die `web.xml` ist der Deployment Descriptor unserer Webanwendung, der zumindest bei Verwendung von älteren Servlet-Spezifikationen benötigt wird. In ihr wird im einfachsten Fall ein spezielles Vaadin-Servlet konfiguriert, das wir für unsere Deployment-Umgebung benötigen. Wollen wir unsere Anwendung in der Google App Engine betreiben, müssen wir eine andere Servlet-Implementierung verwenden, als für ein Deployment als Portlet in einem Portal Server.

Neben der Festlegung der verwendeten Servlet-Klasse gibt es noch einige weitere Initialisierungs- und Kontextparameter, die wir in der `web.xml` konfigurieren können. Dazu gehören unter anderem die Festlegung der Hauptklasse, die das Servlet verwenden soll und die Konfiguration, in welchem Modus die Vaadin-Anwendung betrieben werden soll.

Es gibt zwei Modi, unter denen eine Vaadin-Anwendung laufen kann: der Produktionsmodus und ein Debug-Modus. Gesteuert wird dieser Modus über den Kontextparameter `productionMode`.

```
<context-param>
        <description>Vaadin production mode</description>
        <param-name>productionMode</param-name>
        <param-value>false</param-value>
</context-param>
```

Belegt man diesen Wert wie im Beispiel mit `false`, so wird die Anwendung im Debug-Modus betrieben. Wir haben dann Zugriff auf einige zusätzliche Features, die uns das Leben als Entwickler etwas erleichtern. Dazu gehört unter anderem ein spezielles Debug-Fenster, das wir zur Laufzeit auf unserer Anwendung anzeigen können. Über dieses Fenster können wir detaillierte Informationen über den clientseitigen Zustand der Applikation erhalten. Wir werden uns diesen Debug-Modus später an anderer Stelle noch etwas genauer anschauen.

Wichtig ist, dass wir in der Produktionsumgebung diesen Modus deaktivieren. Dazu belegen wir den Wert des Kontextparameters `productionMode` mit `true`. Es sind dann sämtliche Debug-Funktionalitäten des Vaadin-Servlets deaktiviert. Das schützt die Interna unserer Anwendung vor allzu neugierigen Blicken und kommt zudem der Anwendungs-Performance zugute.

Der zweite wichtige Wert, den wir in der `web.xml` konfigurieren müssen, ist die `UI`-Klasse, die vom Vaadin-Servlet verwendet werden soll. Dazu definieren wir den Wert des Servlet Init-Parameters `UI`. Mit diesem Parameter geben wir dem Vaadin-Servlet unsere eigene Subklasse von `com.vaadin.ui.UI` bekannt, welche ja die Haupteinstiegsklasse unserer Vaadin-Anwendung darstellt.

```
<init-param>
        <description>Vaadin UI class to use</description>
        <param-name>UI</param-name>
        <param-value>org.vaadin.hellovaadin.HelloVaadinUI</param-value>
</init-param>
```

Das Vaadin-Servlet muss, sobald eine Anwenderin oder ein Anwender die Vaadin-Applikation mit dem Browser besucht, eine neue Instanz unserer UI-Klasse erzeugen und diese Instanz in die HTTP Session stellen. Dieses UI-Objekt enthält unter anderem alle UI-Komponenten, die aktuell im Browser des Benutzers dargestellt werden, und deren jeweiligen Zustand. Damit das Vaadin-Servlet neue Instanzen unserer UI-Klasse erstellen kann, muss man ihm den voll qualifizierten Klassennamen dieser Klasse mitteilen. Dies geschieht mit dem `UI`-Parameter. Das Servlet kann damit bei Bedarf per Reflection neue Instanzen erstellen.

## Es geht los: Deployment der Anwendung

Das Vaadin Eclipse Plugin hat uns mit diesen Dateien eine voll funktionsfähige Vaadin-Anwendung generiert. Wir können diese jetzt so wie sie ist in einem Servlet Container deployen und starten. Lassen Sie uns das an dieser Stelle einfach einmal tun.

Wir müssen dazu vorher eine Server-Laufzeitumgebung in Eclipse eingerichtet haben. Dies lässt sich in den Einstellungen unter *Window* → *Preferences* → *Server* → *Runtime Environments* bewerkstelligen.

Ist das erledigt, kann unser Projekt auf zweierlei Arten auf dem Server installiert werden. Wir können einfach per Drag & Drop das Projektverzeichnis aus dem *Package Explorer* oder dem *Project Explorer* auf den Server-Eintrag in der *Servers View* ziehen. Alternativ können wir mit der rechten Maustaste auf dem Projekt das Kontextmenü aufrufen und dort unter der Option *Run As* → *Run on Server* den Zielserver auswählen.

Bei beiden Varianten wird anschließend der Server gestartet und die Anwendung darauf installiert. Wenn der Server hochgefahren ist, kann die *Hello Vaadin* Anwendung unter der folgenden URL besucht werden (Eclipse wird dann automatisch ein internes Browserfenster öffnen):

```
http://localhost:8080/HelloVaadin/
```

Sie müssen gegebenenfalls nur den Port 8080 an den von Ihrem Servlet Container verwendeten Port angleichen.

## Schreiben unseres Anwendungscodes

Lassen Sie uns jetzt den vom Vaadin Eclipse Plugin generierten Beispielcode mit unserem eigenen Code ersetzen. Wir wollen das folgende Beispielprogramm als unsere erste Vaadin-Anwendung schreiben.

Die *Hello Vaadin* Beispielanwendung

```
1  package org.vaadin.hellovaadin;
2
3  import com.vaadin.annotations.Theme;
4  import com.vaadin.server.VaadinRequest;
5  import com.vaadin.ui.Button;
6  import com.vaadin.ui.Button.ClickEvent;
7  import com.vaadin.ui.ComponentContainer;
8  import com.vaadin.ui.Label;
9  import com.vaadin.ui.Notification;
10 import com.vaadin.ui.TextField;
11 import com.vaadin.ui.UI;
12 import com.vaadin.ui.VerticalLayout;
13
14 @Theme("hellovaadin")
15 public class HelloVaadinUI extends UI {
16
17   @Override
18   protected void init(VaadinRequest request) {
19     // {1} //
20     final VerticalLayout layout = new VerticalLayout();
21     layout.setMargin(true);
22     layout.setSpacing(true);
23
24     buildHomeScreen(layout); // {2} //
25     setContent(layout);      // {3} //
26   }
27
28   private void buildHomeScreen(final ComponentContainer layout) {
29     // {4} //
30     final TextField nameTextField = new TextField("Wie lautet Ihr Name?");
31     nameTextField.setRequired(true);
32     final Button sayHelloButton = new Button("Sag mal Hallo...");
33
34     // {5} //
35     sayHelloButton.addClickListener(new Button.ClickListener() {
36       @Override
37       public void buttonClick(ClickEvent event) {
38         if (nameTextField.isValid()) {                        // {6} //
39           layout.removeAllComponents();                       // {7} //
40           buildHelloScreen(layout, nameTextField.getValue()); // {8} //
41         } else {
42           Notification.show("Geben Sie bitte Ihren Namen ein."); // {9} //
43         }
44       }
45     });
```

```
46
47      // {10} //
48      layout.addComponent(nameTextField);
49      layout.addComponent(sayHelloButton);
50    }
51
52    private void buildHelloScreen(final ComponentContainer layout, String name)\
53   {
54      // {11} //
55      final Label helloLabel = new Label(String.format("Hallo %s!", name));
56      final Button backButton = new Button("<< Zurück");
57
58      backButton.addClickListener(new Button.ClickListener() {
59        @Override
60        public void buttonClick(ClickEvent event) {
61          layout.removeAllComponents(); // {12} //
62          buildHomeScreen(layout);      // {13} //
63        }
64      });
65
66      // {14} //
67      layout.addComponent(helloLabel);
68      layout.addComponent(backButton);
69    }
70  }
```

Zugegeben, dieser Code geht über ein simples *Hello World*-Programm doch ein wenig hinaus. Aber wenn wir ehrlich sind, würde es uns, nachdem wir den vom Vaadin Eclipse Plugin generierten Code gesehen haben, nicht mehr sonderlich aus den Socken hauen, wenn wir uns anschauen würden, wie man ein simples Label auf die Benutzeroberfläche legt.

Stattdessen sehen wir an diesem einfachen Beispiel die wichtigsten Grundbestandteile, aus der jede nicht-triviale Vaadin-Anwendung besteht, komprimiert an einer Stelle: Ereignisbehandlung, Navigation und Eingabevalidierung.

Schauen wir uns das Beispiel einmal Schritt für Schritt an. Die `init()`-Methode {1} übernimmt typische Konstruktoraufgaben — mit dem Unterschied, dass sie keine Klasse initialisiert, sondern unsere Vaadin-Anwendung. Hier wird das Hauptlayout eingerichtet {2} und als Inhalt (*Content*) der UI-Klasse gesetzt {3}. Mit der Methode `UI#setContent(com.vaadin.ui.Component)` wird der Inhalt des Browserfensters festgelegt. Das ist typischerweise eine Layout-Komponente, die selbst wiederum rekursiv sämtliche geschachtelten Layouts und UI-Komponenten der Benutzeroberfläche enthält.

Das Zusammenstellen der UI-Komponenten, die auf dem Hauptlayout zu sehen sein sollen, wurde in eine eigene Methode `buildHomeScreen()` ausgelagert. Diese Methode werden wir später noch einmal benötigen.

Werfen wir einen Blick auf `buildHomeScreen()`. Hier werden zwei UI-Komponenten erzeugt {4} und auf das Layout gelegt {10}. Wir erstellen uns ein Textfeld, in das die Anwender ihren Namen eingeben können, und eine Schaltfläche, die uns auf eine zweite Seite leiten wird.

Auf dieser werden die Anwender mit ihrem Namen begrüßt. Die Eingabe in das Textfeld ist verbindlich. Das Textfeld wird daher mit `setRequired(true)` als Pflichtfeld konfiguriert.

Auf der Schaltfläche `sayHelloButton` registrieren wir einen Event Listener für Button Clicks. Den `com.vaadin.ui.Button.ClickListener` implementieren wir als anonyme Klasse. In dieser Ereignisbehandlungsroutine überprüfen wir zuerst, ob die Eingabe in das Textfeld gültig ist {6}. Ist das nicht der Fall, wird ein Hinweis angezeigt {9}.

Hat man einen Namen angegeben, findet als nächstes eine Navigation auf eine zweite "Seite" statt. Dazu entfernen wir sämtliche vorhandenen UI-Komponenten von dem Hauptlayout {7} und ersetzen diese mit den Komponenten, die in der Methode `buildHelloScreen()` erzeugt werden {8}. Dieser Methode übergeben wir den Inhalt des Textfeldes.

Die Methode `buildHelloScreen()` erzeugt die zweite Seite unserer Beispielanwendung. Hier wird ein Label mit dem Gruß an die Anwenderin oder den Anwender und ein Zurück-Knopf {11} auf das Hauptlayout gelegt {14}. Auch hier registrieren wir wieder eine anonyme Klasse als `ClickListener` für den Zurück-Knopf. Der Code für diesen Listener ist sehr einfach: wir entfernen wieder sämtliche Komponenten vom Layout {12} und rufen die Methode `buildHomeScreen()` {13} auf, die unsere Benutzeroberfläche in ihren Ursprungszustand zurückversetzt.

## 2.2 Grundeigenschaften einer Vaadin-Anwendung

Wie am Anfang versprochen, können wir an diesem Beispiel sehr schön die Eigenschaften einer typischen Vaadin-Anwendung in ihren Grundzügen erkennen: Ereignisbehandlung, Navigation und Validierung.

### Ereignisbehandlung

Das Verhalten einer Vaadin-Applikation ist ereignisgetrieben. Das heißt, nachdem wir die UI-Komponenten für die Benutzeroberfläche zusammengestellt und im Browser dargestellt haben (mit `com.vaadin.ui.UI#setContent()`), besteht der gesamte Rest der Anwendung nur noch aus der Reaktion auf Benutzeraktionen. Aktionen, die der Benutzer oder die Benutzerin im Browser durchführt, werden als Event zum Server gesendet und dort von *Event Listenern* [3] verarbeitet. Aus diesen Ereignisbehandlungsroutinen heraus wird dann unsere Business-Logik aufgerufen. Vaadin unterscheidet sich also in dieser Hinsicht nicht von anderen UI-Frameworks, wie Swing, JavaFX oder dem SWT (das *Standard Widget Toolkit* von Eclipse).

### Validierung

Die Überprüfung von Benutzereingaben ist ein wichtiger Bestandteil jeder Software. Auch das Vaadin-Framework bringt einige Schnittstellen und Funktionen mit, mit denen uns die Eingabevalidierung wesentlich erleichtert wird und die uns einen Großteil von immer wiederkehrenden Aufgaben abnimmt.

Im Beispiel haben wir die Eingabe in das Textfeld für den Benutzernamen erforderlich (*required*) gemacht. Mit der Methode `isValid()`, die wir auf der Eingabekomponente, dem Textfeld, aufgerufen haben, konnten wir das Framework die Gültigkeit der Benutzereingabe

[3] Ein anderer gängiger Begriff für diese Ereignisbehandlungsroutinen ist *Event Handler*. Während die entsprechenden Interfaces beim GWT das Suffix *Handler* tragen, heißen diese Routinen bei Vaadin bis auf wenige Ausnahmen *Listener*. Bspw. werden beim GWT Mausklicks über die Schnittstelle *ClickHandler* verarbeitet und bei Vaadin über einen *ClickListener*.

überprüfen lassen und entsprechend darauf reagieren. In diesem einfachen Fall prüfen wir, ob überhaupt eine Eingabe gemacht wurde.

Mit der Validierung und den dazugehörigen Validatoren werden wir uns an späterer Stelle noch intensiv auseinandersetzen.

## Single-Page Web Applications

Im Beispielprogramm haben wir gesehen, wie wir die Benutzer von einer Seite der Anwendung auf eine andere Seite weiterleiten können. Nachdem die Anwender ihren Namen eingegeben haben, werden sie auf eine zweite Seite geleitet, auf der sie mit Namen begrüßt werden.

Obwohl das Beispiel extrem simpel ist, demonstriert es doch das Kernprinzip der Seitennavigation mit Vaadin: Anstatt wie es in anderen Web-Frameworks üblich ist, eine komplett neue HTML-Seite zu rendern, haben wir einfach sämtliche UI-Komponenten von der Benutzeroberfläche entfernt (`removeAllComponents()`) und das Basislayout mit den Komponenten gefüllt, die die nächsten Seite darstellen. Technisch befindet sich der Anwender oder die Anwenderin immer auf der gleichen HTML-Seite. Das ist diejenige, die beim ersten Annavigieren der Anwendung geladen wurde. Es werden anschließend nur Teile der Seite oder der gesamte Seiteninhalt nach Bedarf mit Hilfe von JavaScript-Code ausgetauscht. Es wird somit den Benutzern ein vollständiger Seitenwechsel nur vorgegaukelt.

Dieses Prinzip, bei dem eine Anwendung nur auf einer einzigen HTML-Seite betrieben wird, deren Inhalt über JavaScript-Code dynamisch manipuliert wird, ist das definierende Element für so genannte *Single-Page Web Applications*.

Single-Page Web Applications (oder auf Deutsch *Einzelseiten-Webanwendungen*) zeichnen sich dadurch aus, dass sie, wie der Name schon andeutet, auf einer einzigen HTML-Seite betrieben werden. Sie unterscheiden sich damit in dieser Hinsicht wesentlich von klassischen Webanwendungen, bei denen jede Aktion der Benutzer zum Laden einer komplett neuen HTML-Seite führt.

Der erste HTTP GET-Request, den die Benutzerin oder der Benutzer beim Besuch einer solchen Anwendung an den Server schickt, lädt die HTML-Seite, auf der die Single-Page Web Application betrieben wird. Der Seiteninhalt wird anschließend dynamisch, z. B. mit JavaScript, aufgebaut. Alle weiteren Interaktionen mit der Anwendung finden über asynchrone AJAX-Requests statt. Als Ergebnis dieser Server-Anfragen werden nur bestimmte Bereiche der HTML-Seite ausgetauscht oder aktualisiert.

Diese Eigenschaft ist typisch für *Rich Internet Applications*, einer Klasse von Anwendungen, die sich dadurch auszeichnen, dass sie sehr reichhaltige Interaktionsmöglichkeiten bieten und für den Datenaustausch über das Internet mit einem Server kommunizieren können.

Für die Umsetzung von Rich Internet Applications im Browser gibt es viele Ansätze. Dazu gehören plugin-basierte Lösungen, die die Installation einer bestimmten Erweiterung im Browser voraussetzen. Beispiele hierfür sind Microsoft Silverlight oder Adobe Flash. Daneben gibt es Ansätze, die rein auf HTML5 und JavaScript aufsetzen und die damit ohne die Installation von Plugins auskommen. Zwei wichtige Vertreter hierfür sind das Google Web Toolkit und natürlich das darauf aufsetzende Vaadin Framework.

# 3. Ressourcen

In einer typischen Webanwendung werden wir neben rein textuellen Informationen auch bestimmte Ressourcenarten, wie zum Beispiel Grafiken, Icons oder Datei-Downloads, einbinden wollen. Diese Daten können entweder aus unserer Anwendung selbst oder aus einer externen Quelle stammen. Vaadin unterstützt die Verwendung solcher Ressourcen über eine eigene API.

Ressourcen werden ganz allgemein über das Interface `com.vaadin.server.Resource` abgebildet. Dieses Interface steht an der Spitze der Vererbungshierarchie der verfügbaren Ressourcenarten von Vaadin. Eine `Resource`-Instanz repräsentiert ein Ressourcenobjekt, das auf verschiedene Weise in einer Vaadin-Anwendung eingebunden werden kann: als Link, als Bild, als Icon, als Download oder als eingebettetes Medienobjekt. Damit wird uns zum Beispiel die Darstellung eines Icons auf einer UI-Komponente, die Anzeige einer Bilddatei, das Einbetten eines YouTube-Videos oder einer Webseite in einem iFrame oder der Download einer dynamisch erzeugten Datei ermöglicht.

Vaadin bietet uns eine Reihe von Implementierungsklassen für das `Resource`-Interface. Diese erlauben es uns, Datenobjekte aus verschiedenen Quellen in unserer Anwendung einzubinden. Wir werden in diesem Kapitel die folgenden Ressourcenarten kennenlernen:

- URLs: Mit einer `com.vaadin.server.ExternalResource` legen wir das Ziel eines Links oder die Adresse eines eingebundenen Medienobjekts fest.
- Dateien innerhalb eines Vaadin Themes: Dateien, die innerhalb eines Themes abgelegt sind, können mithilfe einer `com.vaadin.server.ThemeResource` angesprochen werden. Themes werden wir in einem späteren Kapitel kennenlernen.
- Dateien aus dem Dateisystem oder dem Klassenpfad werden mit den beiden Klassen `com.vaadin.server.FileResource` und `com.vaadin.server.ClassResource` eingebunden.
- Dynamisch generierte Daten: Ressourcen können auch *on-the-fly* durch die Vaadin-Anwendung oder von einem beliebigen externen Prozess erzeugt werden. Dies geschieht über die Klasse `com.vaadin.server.StreamResource`.
- *Font Icons*: Dies sind spezielle Schriftarten, deren Zeichen keine Buchstaben, sondern einzelne Piktogramme darstellen. Damit bieten Font Icons eine Alternative zur Verwendung von Bilddateien. Das Interface `com.vaadin.server.FontIcon` bildet die Grundlage für diese Art von Grafiken.

In diesem Kapitel wollen wir uns mit dem Einsatz solcher Ressourcen in einer Vaadin-Anwendung beschäftigen. Wir werden dazu die verschiedenen Ressourcenarten und die zu ihnen gehörenden Implementierungsklassen kennenlernen. Außerdem werden wir natürlich auch die Szenarien ansprechen, für die die einzelnen Ressourcenklassen eingesetzt werden können, und wir werden herausfinden, in welcher Situation eine bestimmte Ressourcenart am besten geeignet ist.

# 3.1 Verwendung von Ressourcen

Ressourcenobjekte können in einer Vaadin-Anwendung an verschiedenen Stellen eingesetzt werden. Sie werden dabei immer ganz allgemein über das `Resource`-Interface referenziert. So gibt man mit einer `Resource` das Ziel einer `Link`-Komponente an, oder man definiert mit ihr die Quelle eines `com.vaadin.ui.Image`-Objekts, oder man setzt die Adresse für ein `BrowserFrame`, einer Komponente, mit der sich ein iFrame in eine Anwendung einbetten lässt.

**Einige Methodensignaturen, die das Resource-Interface als Parameter verwenden**

```
com.vaadin.ui.Link#setResource(Resource resource)
com.vaadin.ui.AbstractEmbedded#setSource(Resource source)
com.vaadin.ui.AbstractComponent#setIcon(Resource icon)
com.vaadin.ui.Notification#setIcon(Resource icon)
com.vaadin.ui.Video#Video(String caption, Resource source)
com.vaadin.ui.Video#setPoster(Resource poster)
com.vaadin.ui.BrowserFrame#BrowserFrame(String caption, Resource source)
```

Die Art und Weise, wie diese Ressource dann behandelt wird – das heißt ob ihre Daten auf der Programmoberfläche angezeigt werden oder ob sie zum Download angeboten wird – hängt von der Komponente ab, der man das `Resource`-Objekt übergibt. So stellt `com.vaadin.ui.Link` eine Ressource einfach in einem `<a>`-Tag als Hyperlink auf die Ressource dar, während die `com.vaadin.server.FileDownloader`-Extension die Ressourcendaten als Download bereitstellt [1].

Eine Ausnahme bei den `Resource`-Implementierungen bilden Font Icons. Diese können nur an einer einzigen Stelle, und zwar ausschließlich als Komponenten-Icons verwendet werden. Das heißt, sie dürfen lediglich als Parameter von Methoden Verwendung finden, mit denen sich ein Icon für eine Oberflächenkomponente setzen lässt. Ein Font Icon darf also insbesondere nicht als Quelle für ein `Image`-Objekt oder als Linkziel benutzt werden. Eine detailliertere Beschreibung von Font Icons finden Sie am Ende dieses Kapitels.

## Ein Beispiel: Setzen eines Links

Bevor wir uns in die Details der verschiedenen Ressourcenarten vertiefen, wollen wir zunächst mit einem kleinen Beispiel die wohl am häufigsten verwendete Ressourcenklasse ausprobieren: Wir setzen einen einfachen Hyperlink, der auf die Vaadin-Homepage verweist.

**Listing 1: Setzen eines Hyperlinks mit Vaadin**

```
ExternalResource address = new ExternalResource("http://www.vaadin.com");
Link vaadinHomepage = new Link("Zur Vaadin-Homepage", address);
layout.addComponent(vaadinHomepage);
```

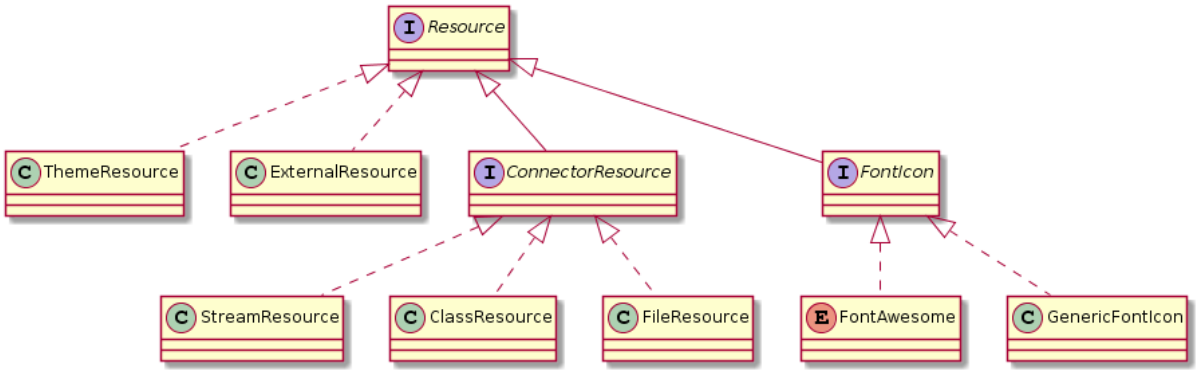
Das war schon alles. Wir erzeugen ein Objekt vom Typ `ExternalResource` und initialisieren dieses mit der Adresse *http://www.vaadin.com*. Dieses Ressourcenobjekt übergeben wir als nächstes dem Konstruktor der Link-Komponente, die wir anschließend auf einem Layout platzieren. Im HTML-Code der Anwendung sehen wir dann das folgende Ergebnis:

[1] Der *Extension*-Mechanismus von Vaadin erlaubt es, eine beliebige Vaadin-Komponente mit einer bestimmten Funktionalität zu erweitern. Dazu gibt es eigene Extension-Komponenten, die sich mit einer normalen UI-Komponente kombinieren lassen. Die Klasse `FileDownloader` ist so eine Erweiterung. Wir werden uns in diesem Grundlagenbuch leider nicht näher mit diesem Konzept befassen können. Die Verwendung von Extensions und die Entwicklung eigener Erweiterungen wird Thema in einem weiteren Buch zu fortgeschrittenen Vaadin-Themen sein.

```
<div class="v-link v-widget">
  <a href="http://www.vaadin.com">
    <span>Zur Vaadin-Homepage</span>
  </a>
</div>
```

## 3.2 Das `Resource`-Interface

Die Hauptschnittstelle für Vaadin Ressourcen ist `com.vaadin.server.Resource`. Dieses Interface wird von jeder Ressourcenart implementiert. Eine Variable vom Typ `Resource` kann damit jede beliebige Ressourcenart repräsentieren. Durch ihren sehr allgemeinen Charakter definiert `Resource` kaum eigene Funktionalität. Es wird nur eine einzige Methode deklariert: `getMIMEType()` zum Festlegen des *MIME Types* einer Ressource. Die Bestimmung des konkreten *MIME Types* wird von Vaadin automatisch übernommen. Beim Zugriff auf eine Ressource wird diese Information immer mitgeschickt und vom Browser entsprechend ausgewertet.




**Vererbungshierarchie der Resource-Schnittstelle**

Das `Resource`-Interface wird von den beiden Klassen `ExternalResource` und `ThemeResource` direkt implementiert. Wir werden diese beiden Ressourcenarten weiter unten noch genauer kennenlernen. Daneben gibt es noch die sogenannten *Connector-Ressourcen*.

## 3.3 Connector-Ressourcen

Direkt von `Resource` abgeleitet ist die Schnittstelle `com.vaadin.server.ConnectorResource`. Eine Besonderheit von Connector-Ressourcen ist, dass ihre Daten über einen sogenannten *Connector* bereitgestellt werden. Um das zu verstehen, müssen wir erst einmal wissen, was ein Connector ist. Ein Connector ist derjenige Teil einer UI-Komponente, der für ihre Kommunikation zwischen Client und Server zuständig ist. Jede Vaadin-Komponente hat ihren eigenen Connector. Damit steht eine Connector-Ressource immer in einer engen Beziehung zu einer bestimmten UI-Komponenteninstanz.

Mit Connector-Ressourcen können Daten bereitgestellt werden, die aus der Anwendung selbst stammen und dabei keinen eigenen, vordefinierten URI mitbringen. Dies können zum Beispiel Daten sein, die erst bei Abruf durch den Browser erzeugt werden.

Die Schnittstelle `ConnectorResource` deklariert daher auch zwei für diesen Zweck passende Methoden: `getStream()` und `getFilename()`. Während `getFilename()` den Dateinamen festlegt,

unter dem der Browser die Ressourcendaten herunterladen soll, liefert `getStream()` die Daten selbst. Diese Methode gibt ein Objekt vom Typ `com.vaadin.server.DownloadStream` zurück. Dieses enthält alle zu einer Connector-Ressource gehörenden Informationen, darunter unter anderem ein `InputStream`, der die Ressourcendaten selbst enthält.

Implementiert wird das Interface `ConnectorResource` von den Klassen `ClassResource`, `FileResource` und `StreamResource`. Während bei `ClassResource` und `FileResource` die Ressourcendaten aus dem Klassenpfad bzw. aus dem Dateisystem stammen, haben wir mit einer `StreamResource` die Möglichkeit, den `InputStream`, der über das `DownloadStream`-Objekt zurückgegeben wird, selbst zu definieren. Damit können wir also dynamisch generierte Daten, wie zum Beispiel Reports oder Datenexporte, bereitstellen.

Wie weiter oben schon erwähnt, sind dies alles Ressourcen, deren Daten aus der Anwendung stammen und die keinen eigenen URI besitzen. Weiter wissen wir, dass man ein beliebiges `Resource`-Objekt zum Beispiel als Zielressource in eine `Link`-Komponente stecken kann. Hierbei haben wir aber scheinbar einen Konflikt, denn wie kann ich eine Ressource ohne eigenen URI als Ziel eines Links angeben, wenn dieses Ziel doch ein URI sein muss? Wie kann aber eine dynamisch generierte oder aus dem Klassenpfad gelesene Ressource mit einem eindeutigen URI adressiert werden? Die Daten liegen ja nicht als durch den Webserver zugreifbare Dateien in einem Dokumentenverzeichnis. Dieser Frage wollen wir im folgenden Abschnitt nachgehen.

## Adressierung von Connector-Ressourcen

Connector-Ressourcen besitzen von Haus aus keinen eigenen URI, unter dem sie abgerufen werden könnten. Sie liegen nicht wie normale Dateien im *Document Root* eines Webservers, sondern werden vom Vaadin-Servlet direkt bereitgestellt und übertragen. Trotzdem kann man Connector-Ressourcen überall dort verwenden, wo ein URI erwartet wird. Zum Beispiel kann man eine Connector-Ressource, die dynamisch erzeugte Inhalte bereitstellt, in ein Link-Objekt setzen. Wie passt das zusammen?

Hier kommt der Connector ins Spiel, der namensgebend für diese Ressourcenart ist. Jede UI-Komponenteninstanz, die auf einer Benutzeroberfläche platziert ist, wird durch ein eigenes Connector-Objekt repräsentiert. Jedes dieser Objekte hat eine eindeutige Id, die von Vaadin intern verwaltet wird. Eine UI-Komponenteninstanz, die auf einer Benutzeroberfläche liegt, befindet sich immer im Inneren einer Komponentenhierarchie unterhalb eines `UI`-Objekts. Wie wir wissen, können sämtliche `UI`-Instanzen einer HTTP-Session über deren UI Id identifiziert werden. Zusammen mit dieser UI Id und einer Connector-Id kann also eine ganz bestimmte UI-Komponenteninstanz innerhalb einer Benutzersession eindeutig adressiert werden.

Mithilfe dieser Adressierung kann Vaadin nun für Connector-Ressourcen einen künstlichen URI erzeugen. Der generierte URI beinhaltet neben dem Dateinamen der Ressource (wird durch `getFilename()` geliefert) die Connector-Id der UI-Komponenteninstanz und die Id des `UI`-Objekts, in dem sich die Instanz befindet. Diese Connector-URIs werden durch das Vaadin-Servlet interpretiert und behandelt. Damit das Vaadin-Servlet weiß, welche Ressourcendaten durch eine bestimmte Komponente bereitgestellt werden sollen, muss es das dazugehörige `Resource`-Objekt kennen. Anhand des Connector-URIs kann sich das Vaadin-Servlet dieses Objekt gezielt aus der HTTP-Session fischen. Dadurch wird es möglich, auch dynamisch generierte Ressourcen eindeutig zu adressieren.

Schauen wir uns ein einfaches Beispiel dazu an. Wir wollen eine Datei `Readme.txt`, die im lokalen Dateisystem liegt, mithilfe eines Links zum Download anbieten. Dazu verwenden wir

eine `FileResource` und die `Link`-Komponente.

**Verlinkung einer lokalen Dateiressource**

```
1 FileResource fileResource = new FileResource(new File("C:/readme.txt"));
2 Link link = new Link("Download readme.txt", fileResource);
3 mainLayout.addComponent(link);
```

Wenn wir jetzt den HTML-Code betrachten, der durch dieses Beispiel erzeugt wird, sehen wir folgendes Ergebnis:

**Von Vaadin erzeugter Connector-URI für die Datei readme.txt**

```
<div class="v-link v-widget">
  <a href="http://localhost:8080/APP/connector/1/19/href/readme.txt">
    <span>Download readme.txt</span>
  </a>
</div>
```

Hier bekommen wir den künstlichen URI

```
http://localhost:8080/APP/connector/1/19/href/readme.txt
```

Dieser enthält die Id 1 des aktuellen `UI`-Objekts und die Id 19 des Connectors für das `Link`-Objekt. Anhand dieser Information kann das Vaadin-Servlet das `Link`-Objekt aus der HTTP-Session ermitteln und dessen `FileResource`-Objekt auslesen. Mit der Information aus dieser Dateiressource kann Vaadin nun den Inhalt von `readme.txt` als Download ausliefern.

Auch wenn der Connector für uns eindeutige URIs generiert, so muss man hierbei beachten, dass diese temporär und damit nicht stabil sind. Sie haben außerhalb des aktuellen Session-Kontextes keinerlei Bedeutung. Man darf diese URIs also nicht auslesen und an anderer Stelle als Referenz auf die Ressource verwenden. Genauso wenig eignen sich diese künstlichen URIs für die Weitergabe an andere Nutzer, zum Beispiel über einen per Email geschickten Link. Das liegt darin begründet, dass in den generierten URIs die Ids von Connector und `UI`-Objekt einkodiert sind, und die sind vom Zustand einer ganz bestimmten HTTP-Session abhängig. Jeder Anwender sieht also in seinem Browser einen anderen URI für das gleiche Link-Objekt!

Sie können das einmal nachvollziehen, indem Sie eine Anwendung, in der ein `Link`-Objekt mit einer Connector-Ressource verwendet wird, in zwei Browser-Tabs öffnen. Wenn Sie sich dann das Linkziel des `Link`-Objekts im HTML-Quelltext anschauen, sehen Sie, dass Sie hier zwei unterschiedliche Zieladressen bekommen.

## 3.4 Implementierungsklassen von `Resource`

Es ist nun an der Zeit, sich die verschiedenen Ressourcenimplementierungen ein wenig genauer anzuschauen.

### ExternalResource

Die einfachste Ressourcenart haben wir am Anfang schon in einem Beispiel gesehen. Wir verwenden `ExternalResource`, um eine externe Ressource über deren URI zu definieren. Die Klasse erwartet eine gültige URL als Konstruktorargument. Diese kann entweder als String oder als Objekt vom Typ `java.net.URL` übergeben werden.

Neben der Definition der Zieladresse für einen Link ist ein weiterer typischer Verwendungszweck von `ExternalResource` die Angabe der Quelle eines `com.vaadin.ui.Image`-Objekts. Mit der `Image`-Komponente lässt sich ein Bild auf einer Seite anzeigen.

### ThemeResource

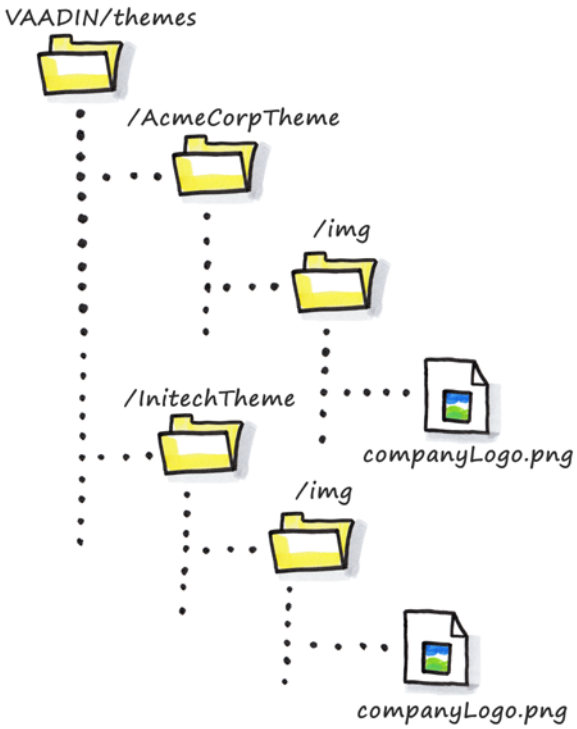
Die Ressourcenart `ThemeResource` ermöglicht das Einbinden von Dateien, die unterhalb eines Vaadin Themes abgelegt sind. Als Theme wird die Anpassung der optischen Gestaltung einer Vaadin-Anwendung mithilfe von *Cascading Style Sheets* und Bilddateien bezeichnet. Wie wir in dem Kapitel über Themes noch sehen werden, besteht ein Theme aus einem speziellen Wurzelverzeichnis, dem weitere Dateien und Verzeichnisse untergeordnet sind. Unter anderem befinden sich darin CSS-Dateien, die das Erscheinungsbild einer Vaadin-Anwendung anpassen. Man kann daneben aber auch weitere Ressourcen ablegen, wie zum Beispiel Bilder, Icons oder andere Mediendateien. Mithilfe einer `ThemeResource` kann eine solche Datei immer in Bezug auf das aktuell verwendete Theme referenziert werden.

Das Besondere an Theme-Ressourcen ist, dass erst zur Laufzeit abhängig vom aktuell verwendeten Theme bestimmt wird, welche konkrete Datei von der Ressource tatsächlich referenziert wird. Wechselt man das Theme, werden dadurch von den Theme-Ressourcen auch andere Dateien referenziert. Theme-Ressourcen werden immer über relative Pfadangaben definiert, die sich auf das aktive Theme beziehen.

Theme-Ressourcen werden in erster Linie für Bilddateien verwendet, die vom aktiven Theme abhängig sein sollen, also zum Beispiel Icons, Logos oder grafische Gestaltungselemente. Beispielsweise kann man damit theme-abhängige Icon-Sets definieren. Wenn das aktuelle Theme gewechselt wird, sollen dadurch natürlich auch die von der Anwendung dargestellten Icons ausgetauscht werden, so dass ein zur Optik des Themes passendes Icon-Set verwendet wird.

Mit dieser Fähigkeit lassen sich sehr leicht mandantenfähige Anwendungen erstellen, die sich in ihrem Erscheinungsbild an die jeweiligen Mandanten anpassen können, ohne dass für neue Mandanten der Quelltext der Anwendung angefasst werden müsste. Eine solche Anwendung kann dann für verschiedene Kunden oder Anwenderkreise unterschiedlich aussehen.

Stellen Sie sich vor, Sie hätten zwei für Ihre Kunden zugeschnittene Themes *AcmeCorpTheme* und *InitechTheme*. Beide Themes enthalten in einem Unterverzeichnis `img` jeweils eine Grafik `companyLogo.png`, welche das Firmenlogo des jeweiligen Kunden darstellt. Das folgende Schaubild zeigt die Verzeichnisstruktur, die sich für die beiden Themes in Ihrem Projekt ergibt.




**Verzeichnisstruktur für zwei Themes innerhalb des `VAADIN/themes`-Verzeichnisses**

Sie können dieses Logo wie folgt auf Ihrer Programmoberfläche einbinden:

**Verwendung einer ThemeResource zum Einbinden eines Logos**

```
ThemeResource logoResource = new ThemeResource("img/companyLogo.png");
Image image = new Image("", logoResource);
layout.addComponent(image);
```

Wie Sie sehen, müssen wir hier keine explizite Referenz auf ein bestimmtes Theme angeben. Wir spezifizieren lediglich einen relativen Pfad, dessen Wurzel sich immer auf das Basisverzeichnis des aktiven Themes bezieht. Ist z. B. das Theme für den Kunden Initech aktiv, wird auch das Firmenlogo aus diesem Theme angezeigt. Damit stellen wir sicher, dass für jeden Kunden immer das passende Logo angezeigt wird.

Der Pfad zu einer Theme-Ressource darf nicht mit einem `/` beginnen, das heißt die Pfadangabe für eine Theme-Ressource muss immer relativ sein. Sie werden sonst eine `IllegalArgumentException` erhalten.

### `FileResource`

Mit einer `FileResource` lässt sich eine beliebige Datei aus dem lokalen Dateisystem als Ressource einbinden. Dabei spielt es keine Rolle, ob die Datei im Kontext (im *Document Root*) der Webanwendung liegt. Die einzige Voraussetzung ist, dass die Datei durch den System-User lesbar ist, mit dem der Web Server betrieben wird, und dass der Java Security Manager den Zugriff auf das lokale Dateisystem zulässt.

Initialisiert wird eine `FileResource` mit einem `java.io.File`-Objekt. Das folgende Beispiel fügt ein PDF-Dokument aus einem lokalen Verzeichnis in eine Anwendung ein.

**Verwendung einer FileResource als Linkziel**

```
1 File file = new File("/home/rkrueger/documents/agb.pdf");
2 FileResource resource = new FileResource(file);
3 Link link = new Link("AGB herunterladen", resource);
4 layout.addComponent(link);
```

Vorsicht ist hier bei der Angabe von relativen Pfaden für ein `File`-Objekt geboten! Das Basisverzeichnis für ein `File`-Objekt, das mit einem relativen Pfad initialisiert wurde, bezieht sich immer auf das aktuelle Arbeitsverzeichnis des Java-Prozesses, in dem die Anwendung läuft. Im Falle einer Vaadin-Anwendung ist dies der Prozess des Web Servers. Die Verwendung von relativen Pfaden bei Dateiressourcen sollte daher möglichst vermieden werden, um sich nicht an die Ausführungsumgebung des Servlet Containers zu binden.

Bei der Verwendung von Dateiressourcen sollte man auch besonders vorsichtig sein, um nicht aus Versehen die Möglichkeit für eine *Directory Traversal* Attacke zu schaffen. Bei diesem Angriffsvektor versucht ein Angreifer, durch manipulierte Pfadangaben auf Dateien zuzugreifen, die außerhalb eines öffentlich freigegebenen Verzeichnisses liegen. Dies kann immer dann geschehen, wenn Benutzereingaben ungeprüft von der Anwendungslogik verwendet werden.

Gibt man zum Beispiel den Anwendern die Möglichkeit, eine bestimmte Datei aus einem öffentlich zugänglichen Verzeichnis durch Eingabe des Dateinamens auszuwählen, sollte man sicherstellen, dass die eingegebenen Dateinamen keine relativen Verzeichnisangaben enthalten. Sonst ist es einem Angreifer möglich, zum Beispiel die folgende Datei abzurufen: `../../../etc/passwd`.

Directory Traversal Attacken gehören zu den Angriffsvektoren, die durch das OWASP-Projekt[2] [3] dokumentiert sind. Sie werden dort als *Path Traversal Attacks*[4] aufgeführt.

Die Einsatzmöglichkeiten von Dateiressourcen sind relativ beschränkt. Aus Sicherheitsgründen rate ich nach Möglichkeit von der Verwendung dieser Ressourcenart ab. Insbesondere sollten Dateiressourcen nicht zur Auslieferung von Standardressourcen einer Webanwendung verwendet werden. Dazu gehören eingebettete Bilder und Icons. Einer der Gründe hierfür liegt darin, dass diese Ressourcen durch die künstlichen und veränderlichen URIs von Connector-Ressourcen nur schlecht vom Browser gecacht werden können. Je nach Anwendung wird der Browser dieselbe Ressource unter Umständen mehrmals unter verschiedenen URIs cachen. Verwenden Sie für solche Ressourcen möglichst einen dedizierten Web Server, der die Daten unter festen URIs ausliefert. Die oben kennengelernte Klasse `ExternalResource` ist dann der richtige Kandidat.

[2] https://www.owasp.org
[3] Open Web Application Security Project
[4] https://www.owasp.org/index.php/Path_Traversal

### ClassResource

Ressourcen vom Typ `ClassResource` beziehen ihre Daten aus dem Klassenpfad der Anwendung. Bei einer Webanwendung setzt sich der Klassenpfad aus verschiedenen Orten zusammen. Dazu gehört das Verzeichnis *WEB-INF/classes* und *WEB-INF/lib* innerhalb des Deployment-Verzeichnisses der Webanwendung selbst. Dateien, die an diesen Orten abliegen, können mit einer `ClassResource` referenziert werden.

Um die Daten einer Ressource aus dem Klassenpfad auszulesen, holen sich `ClassResource`-Objekte über die Methode `java.lang.Class#getResourceAsStream()` eines bestimmten Klassenobjekts einen `InputStream` auf die referenzierte Datei. Damit wird auf die Ressource über denjenigen Classloader zugegriffen, der das angegebene Klassenobjekt geladen hat. Zum Laden einer `ClassResource` ist also immer die Angabe eines Klassenobjekts notwendig. Die Klasse bietet daher die folgenden beiden Konstruktoren an:

- `ClassResource(Class<?> associatedClass, String resourceName)`
- `ClassResource(String resourceName)`

Mit dem ersten Konstruktor wird das Klassenobjekt explizit vorgegeben, über deren Classloader die Ressource geladen werden soll. Der zweite Konstruktor verwendet standardmäßig die `UI`-Klasse der Anwendung als Referenzklasse.

Wurde die Referenzklasse aus einer bestimmten Jar-Datei geladen, so können mit einer `ClassResource` Dateien eingebunden werden, die aus dieser Jar-Datei stammen.

Wenn Sie Maven als Buildtool verwenden, können Sie Ihre Klassenpfadressourcen im Verzeichnis `src/main/resources/` ablegen. Dateien aus diesem Verzeichnis werden in den Klassenpfad der Anwendung (nach `WEB-INF/classes`) kopiert. Beispielsweise können Sie die Bilddatei `src/main/resources/logo.gif` mit der folgenden `ClassResource` referenzieren:

```
Resource logo = new ClassResource("/logo.gif");
```

Achten Sie hier auf die Angabe eines absoluten Pfades. Ein relativer Pfad bezieht sich immer auf das Package der von der `ClassResource` referenzierten Klasse. Hätten Sie in dem Beispiel also statt "/logo.gif" den Wert "logo.gif" verwendet, dann würde diese Datei im Package Ihrer `UI`-Klasse gesucht werden, also z. B. unter `/com/example/myapp/logo.gif`, wenn Ihre `UI`-Klasse im Package `com.example.myapp` liegt.

Wenn Sie also einmal eine `ClassResource` in Ihrer Anwendung einbinden, und die Daten werden scheinbar nicht geladen, dann überprüfen Sie zuerst, ob der angegebenen Pfad der Ressource korrekt ist.

Die Verwendung von Klassenpfadressourcen bietet sich vor allem dann an, wenn die anderen Ressourcenarten nicht oder nur eingeschränkt zur Verfügung stehen. Hat man zum Beispiel keinen Zugriff auf das Dateisystem oder es steht kein externer Webserver zur Verfügung, der die statischen Ressourcen einer Anwendung ausliefern kann, bleiben einem nur noch `ClassResource` oder `ThemeResource` zum Einbinden von Ressourcen.

Das kann beispielsweise dann der Fall sein, wenn man seine Anwendung in der *Google App Engine*[5] betreibt. Dort hat eine Anwendung nur eingeschränkten Zugriff auf ihre Umgebung. So

[5] https://cloud.google.com

ist zum Beispiel der Zugriff auf ein lokales Dateisystem nicht möglich. Alle Ressourcen, auf die eine Anwendung zugreifen möchte, müssen deshalb innerhalb der Anwendung selbst liegen und mit ihr ausgeliefert werden. Hier bietet es sich besonders an, die Ressourcen in den Klassenpfad der Anwendung zu legen und mit `ClassResource`-Objekten auf diese zuzugreifen.

### `StreamResource`

Mit Ressourcen vom Typ `StreamResource` können wir dynamisch generierte Daten bereitstellen. Mit ihnen haben wir die Möglichkeit, die Ressourcendaten selbst und bei Bedarf zu erzeugen. Um dies zu erreichen, müssen wir uns selbst um die Erzeugung der Daten kümmern, die von dem `InputStream` des Ressourcenobjekts geliefert werden.

Die Klasse `StreamResource` erlaubt uns damit, dynamisch generierte Ressourcen, wie zum Beispiel Datenexporte oder ad-hoc erzeugte Grafiken, zu verwenden.

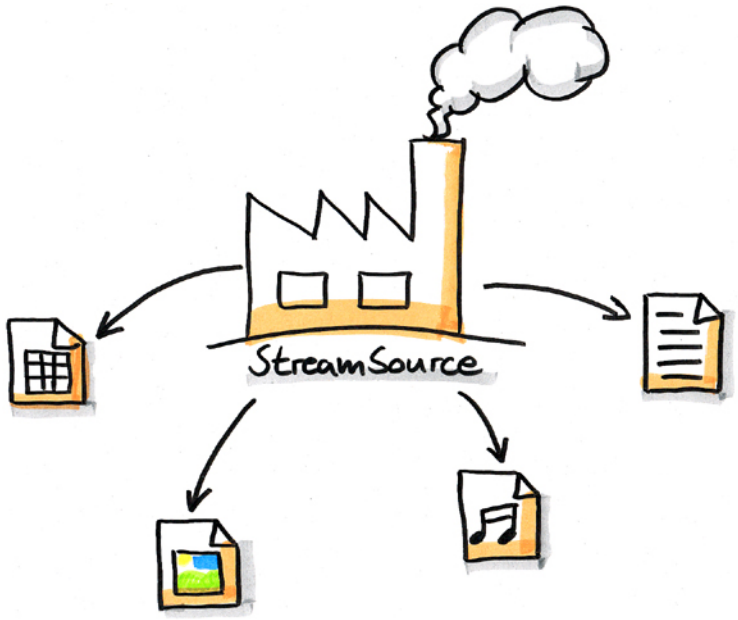
Um eine `StreamResource` zu erstellen, müssen wir uns um zwei Dinge kümmern. Zum einen müssen wir den Dateinamen angeben, unter dem die generierte Ressource angesprochen werden soll. Zum anderen müssen wir die Daten der Ressource selbst erzeugen. Die Daten werden von einer Klasse geliefert, die das Interface `StreamResource.StreamSource` implementiert.

Dieses Interface ist wie folgt definiert:

**Die StreamSource-Schnittstelle**

```
1 public interface StreamSource extends Serializable {
2    public InputStream getStream();
3 }
```

`StreamSource` ist also eine *Fabrikschnittstelle* zur Erzeugung von `InputStreams`, die über `getStream()` die Daten der dynamisch erzeugten Ressource liefert.




**`StreamSource` als Fabrikschnittstelle**

Verwendet wird eine Instanz von `StreamSource` dann als Konstruktorargument für `StreamResource`:

```
com.vaadin.server.StreamResource#StreamResource(StreamSource streamSource,
                                                 String filename)
```

Man mag sich an dieser Stelle vielleicht fragen, warum hier der Zwischenschritt über ein spezielles Interface gegangen wird. Kann man die `StreamResource` nicht gleich mit dem `InputStream` initialisieren? Zur Beantwortung dieser Fragen muss man sich in Erinnerung rufen, dass ein `InputStream` nur genau einmal ausgelesen werden kann. Er kann nicht wieder zurückgesetzt werden. Würde man `StreamResource` direkt mit dem `InputStream` initialisieren, könnte die Ressource auch nur genau ein einziges Mal gelesen werden. Das wäre natürlich bei der Verwendung einer solchen Ressource als Linkziel für den Download einer Datei fatal. Der Link würde genau einmal funktionieren und dann beim zweiten Klick einen Fehler werfen. Aus diesem Grund definieren wir mit `StreamResource.StreamSource` ein Fabrikobjekt für `InputStreams`. Diese Fabrik kann beliebig viele Stream-Instanzen erzeugen — eben für jeden Klick auf den Link eine neue.

Wollen wir mit `StreamResource` bei jeder Anforderung durch den Browser die Ressourcendaten neu generieren lassen, müssen wir beachten, dass der Browser eine einmal erzeugte `StreamResource` im Cache behält und weitere Anforderungen der Ressource daraus bedient werden. Soll eine `StreamResource` wirklich jedes Mal neu erzeugt werden, können wir mit `StreamResource#setCacheTime(0)` die Zeitdauer, die die Ressource im Browser-Cache liegen darf, auf 0 Millisekunden festlegen.

Schauen wir uns ein Beispiel für eine `StreamResource` an. Um unsere Anwender vor Spammern zu schützen, die automatisiert Email-Adressen von Webseiten fischen, wollen wir alle Email-Adressen, die irgendwo angezeigt werden, generell als Grafik darstellen. Die Adressen erscheinen dann nicht im HTML-Quelltext, und wir zwingen unsere Anwender damit, bei Bedarf die Email-Adressen abzutippen. Wir müssen dazu Strings in Bilder umwandeln können. Dies erreichen wir mit der Klasse `EmailImageResource`, die von `StreamResource` abgeleitet ist.

**Die Klasse EmailImageResource, die eine gegebene Email-Adresse in ein Bild umwandelt**

```
1 public class EmailImageResource extends StreamResource {
2   public EmailImageResource(String emailAddress, String filename) {
3     super(new EmailImageSource(emailAddress), filename);
4   }
5 }
```

Wie Sie sehen, ist diese Klasse relativ unspektakulär. Wir rufen einfach nur den Super-Konstruktor auf und übergeben eine Instanz unserer eigenen Implementierung der `StreamSource`-Schnittstelle. Diese sieht wie folgt aus:

**Implementierung von StreamSource, die den übergebenen String in einem Bild ausgibt**

```
1  public class EmailImageSource implements StreamResource.StreamSource {
2    private String emailAddress;
3
4    public EmailImageSource(String emailAddress) {
5      this.emailAddress = emailAddress;              // {1} //
6    }
7
8    @Override
9    public InputStream getStream() {
10     BufferedImage image = new BufferedImage(
11         125, 30, BufferedImage.TYPE_3BYTE_BGR);    // {2} //
12     Graphics graphics = image.getGraphics();
13     graphics.setColor(Color.white);
14     graphics.fillRect(0, 0, 125, 30);
15     graphics.setColor(Color.black);
16     graphics.drawString(emailAddress, 10, 20);     // {3} //
17     try {
18       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
19       ImageIO.write(image, "png", buffer);
20       return new ByteArrayInputStream(
21           buffer.toByteArray());                   // {4} //
22     } catch (IOException e) {
23       e.printStackTrace();                         // {5} //
24       return null;
25     }
26   }
27 }
```

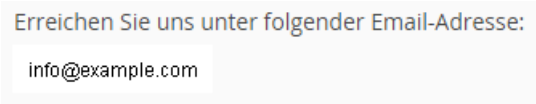
Hier merken wir uns die über den Konstruktor mitgegebene Email-Adresse, damit wir diesen Text später bei Bedarf in ein Bild einfügen können {1}. Interessant ist hier die Implementierung von `getStream()`. Diese Methode soll ja den `InputStream` liefern, der die Daten unserer Ressource enthält. Wir erzeugen uns daher mit den Klassen aus dem `java.awt`-Package ein `BufferedImage`-Objekt, mit dem wir programmatisch ein Bild erzeugen können {2}. Auf dieses Bild zeichnen wir die als Konstruktorargument übergebene Email-Adresse {3}. Wenn das Bild fertig konfiguriert ist, können wir es in einen `ByteArrayInputStream` umwandeln und diesen als Rückgabewert von `getStream()` zurückliefern {4}. Der Umgang mit Exceptions sollte natürlich in einer echten Anwendung etwas eleganter gelöst werden, als in diesem Beispiel {5}.

Unsere eigene Ressourcenimplementierung `EmailImageResource` kann nun als Datenquelle für ein `Image`-Objekt verwendet werden:

**Verwendung der EmailImageResource als Datenquelle für ein Image-Objekt**

```
1 Image emailImage = new Image(
2         "Erreichen Sie uns unter folgender Email-Adresse:",
3         new EmailImageResource("info@example.com", "email.png"));
4 mainLayout.addComponent(emailImage);
```

Das Ergebnis sieht auf der Programmoberfläche dann wie folgt aus:




**Eine Email-Adresse dargestellt in einem Grafikobjekt**

Sie finden den vollständigen Code von diesem Beispiel im Modul *Kap10.4_StreamResource* in den Maven-Beispielprojekten zu diesem Buch.

### `FontIcon` und `FontAwesome`

Über das Interface `FontIcon` haben wir in unserer Anwendung die Möglichkeit, *Font Icons* für Icon-Ressourcen zu verwenden. Font Icons sind eine ressourcenschonende Alternative zu Grafikdateien.

Herkömmliche Icons, die über Bilddateien eingebunden werden, haben zwei wesentliche Nachteile: Icons bestehen üblicherweise aus kleinen Dateien, die einzeln vom Server geladen werden müssen. Dieser Ladevorgang kann bei einer großen Anzahl von Icons den Seitenaufbau spürbar verlangsamen. Dies ist unter anderem der Tatsache geschuldet, dass ein Browser immer nur eine bestimmte, fest vorgegebene Anzahl von Ressourcen gleichzeitig vom Webserver nachladen kann. Die maximal erlaubte Anzahl gleichzeitig geöffneter Server-Verbindungen ist von Browser zu Browser unterschiedlich, bewegt sich aber bei jedem Browser-Typ im niedrigen zweistelligen Bereich.

Ein weiterer Nachteil von Bilddateien ist, dass diese nicht skalierbar sind. Es handelt sich hierbei eben um Rastergrafiken, die man ohne Qualitätsverlust nicht beliebig vergrößern kann. Auch ist ihre Farbgebung fest vorgegeben.

Font Icons verfolgen einen anderen Ansatz. Ein Satz von Font Icons wird durch eine spezielle Schriftart definiert. Eine solche Schriftart setzt sich nicht aus Buchstaben zusammen, sondern aus Piktogrammen — dies ist vergleichbar mit der Windows-Schriftart *Webdings*. Jeder Buchstabe dieser Schriftart stellt damit ein bestimmtes Symbol dar.

Dieser Ansatz hat den Vorteil, dass zum einen der gesamte Satz aller verfügbaren Icons einer Anwendung mit einem Mal (nämlich beim Laden der Schriftart) vom Server geholt wird. Zum anderen sind Font Icons beliebig und ohne Qualitätsverlust skalierbar — Schriftarten bestehen eben aus Vektorgrafiken. Der einzige Nachteil von Font Icons: man kann sie immer nur einfarbig darstellen. Dafür können Font Icons aber mit allen Mitteln, die einem Cascading Style Sheets bieten, beliebig gestaltet werden. Somit kann man die gleichen Icons in verschiedener Darstellung verwenden: schattiert, rotiert, skaliert, durchsichtig, mit Leuchteffekt, usw. — nur eben einfarbig.

`FontIcon` ist nur eine Schnittstelle zur Definition eigener Font Icon Sets. Wenn wir Font Icons in unserer Anwendung verwenden wollen, müssen wir eine Implementierungsklasse von diesem

Interface verwenden. Vaadin bringt von Hause aus die Icons des *Font Awesome*-Projekts[6] mit, die über die Enum-Klasse `com.vaadin.server.FontAwesome` verfügbar sind. Diese Klasse definiert eine lange Liste von Enum-Konstanten, von denen jede ein bestimmtes Font Icon repräsentiert.

Wir können diese Konstanten zum Setzen von Komponenten-Icons verwenden, also immer im Zusammenhang mit der Methode `com.vaadin.ui.Component#setIcon()` (neben einigen anderen Stellen im Vaadin-Framework, wo wir ein Icon setzen können).

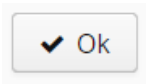
Eine Verwendung der Font Icons als Ziel für Links oder Bilder ist nicht möglich. Der Versuch, eine Font Icon-Ressource als Quelle für ein `Image`-Objekt zu verwenden wird zwar kompilieren, ergibt dann auf der Programmoberfläche aber einen Darstellungsfehler.

Wir können zum Beispiel eine Schaltfläche mit einem Font Icon aufwerten:

**Verwendung eines FontAwesome-Icons auf einer Schaltfläche**

```
1 Button okButton = new Button("Ok");
2 okButton.setIcon(FontAwesome.CHECK);
```

Im Ergebnis sieht diese Schaltfläche wie folgt aus:




**Eine Schaltfläche mit Font Icon**

Der FontAwesome Icon Font ist automatisch in Vaadins *Valo* Theme eingebunden. Möchte man FontAwesome (oder einen anderen Icon Font) mit einem anderen Theme verwenden, so muss dies explizit für das Theme aktiviert werden. Wie das funktioniert, werden wir uns in dem Kapitel über Vaadin Themes genauer anschauen.

## 3.5 Zusammenfassung

Dieses Kapitel hat uns mit dem Ressourcen-Mechanismus von Vaadin bekannt gemacht. Ressourcen sind für Vaadin Daten, die von einer Anwendung eingebunden und angezeigt werden können. Das können Bilder, Icons oder beliebige andere Datei-Downloads sein. Die Daten können entweder von der Anwendung selbst zur Verfügung gestellt werden oder sie stammen aus einer externen Quelle.

Ressourcen können von unterschiedlichen UI-Komponenten dargestellt werden. Der häufigste Fall wird die Verwendung einer `ExternalResource` zusammen mit einer `Link`-Komponente sein, um einen Hyperlink zu erhalten. Ressourcen können aber auch als Grafiken oder andere Medienobjekte eingebunden und als Piktogramm, Bild-, Video- oder Audio-Datei verwendet werden.

Vaadin stellt uns einige Ressourcenimplementierungen zur Verfügung, die ihre Daten aus unterschiedlichen Quellen beziehen. Hier haben wir `ExternalResource` zur Adressierung einer Ressource über eine URL und `ThemeResource` für Ressourcen aus einem Vaadin Theme kennengelernt.

[6] http://fontawesome.io

Weiter haben wir Bekanntschaft mit Connector-Ressourcen gemacht, eine Klasse von Ressourcen, die aus der Vaadin-Anwendung selbst stammen und die keinen eigenen URI haben. Damit diese Art von Ressourcen dennoch bspw. zusammen mit einem `Link`-Objekt verwendet werden können, kümmert sich Vaadin um die Erzeugung eines künstlichen URIs für diese Ressourcen. In diesem URI steckt die Connector-Id der UI-Komponente und die UI Id der aktuellen `UI`-Instanz.

Vaadin bietet uns drei verschiedene Connector-Ressourcenarten an. Mit `FileResource` können wir eine beliebige Datei aus dem lokalen Dateisystem als Ressource einbinden. `ClassResource` erlaubt uns den Zugriff auf Ressourcen, die im Klassenpfad einer Anwendung liegen. Und `StreamResource` ermöglicht uns die dynamische Generierung der Ressourcendaten bei Bedarf, indem wir selbst den `InputStream` der Ressource erzeugen.

Als letzte Ressourcenart haben wir die Font Icons kennengelernt. Font Icons bestehen aus einer speziellen Schriftart, deren Buchstaben einzelne Piktogramme darstellen. Damit sind Font Icons wesentlich besser skalierbar und performanter als einzelne Bilddateien, können aber immer nur mit einer Vordergrund- und einer Hintergrundfarbe dargestellt werden. Vaadin liefert über das *Valo*-Theme die Font Icons des *Font Awesome*-Projekts mit, auf die wir mit der Enum-Klasse `FontAwesome` zugreifen können.