02-2014 | Sommer | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community — für die Community

Java aktuell

Java blüht auf

Praxis
Performance richtig bewerten

Technologie
HTML5 und Java

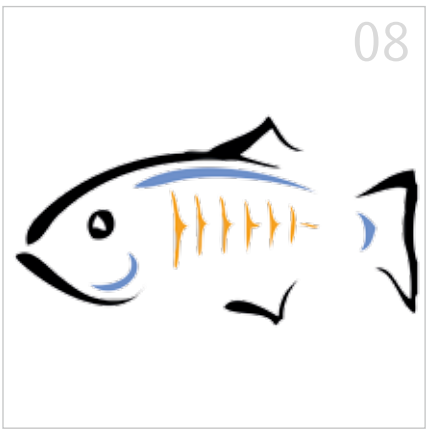
Java Server Faces
Umstieg auf 2.x

Oracle
Anwendungsentwicklung mit WebLogic

Sonderdruck

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

iJUG Verbund



Die neue Oracle-Strategie zum GlassFish Server, Seite 8

HTML5 und Java-Technologien, Seite 18

Java und funktionale Programmierung - ein Widerspruch?, Seite 23

Eingebettete DSLs mit Clojure, Seite 26

Contexts und Dependency Injection – die grundlegenden Konzepte

Dirk Mahler, buschmais GbR

Die Voraussetzung zur Nutzung von Contexts und Dependency Injection (DI/CDI) ist natürlich die Verwendung eines entsprechenden Containers. Dies kann ein vollständig Java-EE-6-kompatibler Application-Server sein (etwa JBoss AS 7.x), für kleinere Ansprüche stehen JBoss Weld (Referenz-Implementierung) und Apache OpenWebBeans als Stand-Alone-Bibliotheken zur Verfügung. Als Einstieg in den Artikel empfiehlt sich der Beitrag „Contexts und Dependency Injection – der lange Weg zum Standard" aus der letzten Ausgabe.

Ein Container durchsucht während der Deployment- beziehungsweise Initialisierungs-Phase den Klassenpfad der Anwendung nach sogenannten „Bean Archives". Das sind JAR- beziehungsweise WAR-Archive, die durch die Existenz eines XML-Deskriptors namens „beans.xml" im Verzeichnis „META-INF/" beziehungsweise „WEB-INF/" für WAR-Archive gekennzeichnet sind (siehe Listing 1). Damit wird signalisiert, dass der Container die Instanzen der darin enthaltenen Klassen als Beans betrachten und verwalten soll. Das betrifft sowohl die Erzeugung als auch die Injizierung benötigter Abhängigkeiten. Der Deskriptor selbst kann leer sein, er dient in erster Linie als Marker.

Bean und Injection-Point

Jede in einem Bean-Archive enthaltene Klasse ist potenziell eine Bean, die durch den Container verwaltet werden kann, sofern sie vorgegebenen Konventionen entspricht. Dies ist bereits der Fall, wenn sie über einen nicht privaten Default-Konstruktor verfügt. Sogenannte „Plain Old Java Objects" (POJO) erfüllen im Normalfall diese Bedingung – im einfachsten Fall sind keine weiteren Annotationen oder Deskriptoren notwendig. Benötigt eine Bean weitere Abhängigkeiten (wie Dienste), können diese mit der Annotation „@Inject" als sogenannte „Injection Points" definiert werden. Dazu gibt es verschiedene Möglichkeiten, am gebräuchlichsten ist die Verwendung von Instanz-Variablen („Field-Injection", siehe Listing 2).

Fordert „ClientImpl" eine Instanz am Container an, wird sie dort erzeugt und die darin mit „@Inject" annotierten Variablen werden mit verfügbaren Instanzen der jeweiligen Typen initialisiert. Dazu muss in der Gesamtheit aller Bean-Archive des Klassenpfads exakt eine zu diesem Typ zuweisungskompatible und durch den Container verwaltete Bean existieren. Im einfachsten Fall gibt es schlicht eine Klasse, die das Interface „IService" implementiert.

Sie kann natürlich wiederum über Injection Points verfügen. Der so entstehende – oft recht komplexe – Abhängigkeitsgrad wird durch den Container aufgelöst. Er ist übrigens dazu verpflichtet, bereits

```
serviceApi.jar:
      /com/buschmais/service/api/IService.java
      /META-INF/beans.xml
serviceImpl.jar:
      /com/buschmais/service/impl/ServiceImpl.java
      /META-INF/beans.xml
client.war:
      /WEB-INF/beans.xml
      /WEB-INF/web.xml
      /WEB-INF/classes/com/buschmais/client/ClientImpl.java
```

Listing 1

```
ClientImpl.java:
public class ClientImpl implements IClient {
    @Inject
    private IService service;
    …
}
```

Listing 2

während der Initialisierung der Anwendung das Vorliegen möglicher Inkonsistenzen zu prüfen. Der Sinn besteht darin, Probleme so früh wie möglich – also beim Deployment und nicht erst zur Laufzeit – zu erkennen.

Das wäre im beschriebenen Beispiel der Fall, wenn keine oder mehr als eine Implementierung von „IService" als Bean zur Verfügung stehen. Der Container würde die Initialisierung mit einer Fehlermeldung abbrechen, die auf eine unbefriedigte Abhängigkeit („unsatisfied dependency") beziehungsweise eine mehrdeutige Abhängigkeit („ambiguous dependency") hinweist.

Qualifier

In größeren Anwendungen lassen sich die letztgenannten, Typ-bezogenen Mehrdeutigkeiten nicht vermeiden. Sie müssen durch „Qualifizierung" aufgelöst werden. Dazu definiert man im Anwendungscode entsprechende Annotationen, die an den zu erzeugenden Beans und Injection Points verwendet werden. Die Annotationen sind mit der Meta-Annotation „@Qualifier" zu versehen (siehe Listing 3).

Die Qualifier-Annotationen ergänzen also Typ-Definitionen, sind aber von ihnen unabhängig. Es kommt daher häufig vor, dass ein und dieselbe Qualifier-Annotation auf verschiedene Vererbungs-Hierarchien angewendet wird. Dies ist etwa der Fall, wenn mehrere technische Dienste (wie EntityManager und Datasource) jeweils für unterschiedliche fachliche Belange zur Verfügung stehen sollen. Die fachliche Trennung erfolgt dann anhand entsprechender Qualifikation (wie „@UserData" und „@ShopData"). Darüber hinaus ist es möglich, Attribute in Qualifier-Annotationen zu deklarieren, deren Werte ebenfalls zur Auflösung herangezogen werden (siehe Listing 4).

Producer

Oft ist es notwendig, Instanzen benötigter Abhängigkeiten im Anwendungscode selbst erzeugen zu können. Ein Beispiel dafür könnten die erwähnten Instanzen eines JPA-EntityManager oder einer JDBC-Datasource sein, für die keine Beans als eigenständige Klassen in einem Bean-Archiv zur Verfügung stehen. In diesem Fall kommen sogenannte „Producer-Felder" beziehungsweise „-Methoden" zum Einsatz. Der Container erkennt sie an der Annotation „@Produces".

```
InMemory.java:
      @Qualifier
      public @interface InMemory {
      }
Persistent.java:
      @Qualifier
      public @interface Persistent {
      }
InMemoryServiceImpl.java:
      @InMemory
      public class ServiceImpl implements IService {
         ….
      }
PersistentServiceImpl.java:
      @Persistent
      public class ServiceImpl implements IService {
         ….
      }
ClientImpl.java:
      public class ClientImpl implements IClient {
          @Inject
          @InMemory
          private IService service;
          …
}
```

Listing 3

```
InMemory.java:
      @Qualifier
      public @interface InMemory {
          boolean transactional();
      }
TransactionalInMemoryServiceImpl.java:
      @InMemory(transactional=true)
      public class ServiceImpl implements IService {
          ….
      }
NonTransactionalInMemoryServiceImpl.java:
      @InMemory(transactional=false)
      public class ServiceImpl implements IService {
          ….
      }
ClientImpl.java:
      public class ClientImpl implements IClient {
          @Inject
          @InMemory(transactional=true)
          private IService service;
          …
      }
```

Listing 4

```
EntityManagerProducer.java:
public EntityManagerProducer {
    @Produces
    public EntityManager getEntityManager(EntityManagerFactory factory) {
        return entityManagerFactory.getEntityManager();
    }
    …
    public void close(@Disposes EntityManager entityManager) {
        entityManager.close();
    }
}
DataSourceProducer.java:
public DataSourceProducer {
    @Produces
    @UserData //application defined qualifier annotation
    @Resource(name="jdbc/UserDataSource")
    private DataSource dataSource;
}
```

Listing 5

Listing 5 macht den Einsatz deutlich: Die Klasse „EntityManagerProducer" stellt eine entsprechend annotierte Methode bereit, die durch den Container aufgerufen wird, sobald eine Instanz eines „EntityManager" benötigt wird. Die Methode selbst kann – wie gezeigt – parametrisiert sein, für die benötigten Instanzen (im konkreten Fall vom Typ „EntityManagerFactory") muss eine entsprechende Bean beziehungsweise ein Producer verfügbar sein – die beschriebene Producer-Methode wird damit selbst zum Injection-Point („Parameter-Injection").

Die Klasse „EntityManagerProducer" enthält darüber hinaus die Methode „close()", deren Parameter mit „@Disposes" annotiert ist und die im vorliegenden Fall dazu dient, die erzeugte „EntityManager"-Instanz zu schließen, wenn sie nicht mehr benötigt wird. Wann genau dies der Fall ist, wird später erläutert. Interessant ist hier zunächst der Umstand, dass man auf diesem Weg einfach und elegant den Lebenszyklus einer Ressource kontrollieren kann. Dispose-Methoden sind ebenfalls Injection-Points, sie können also analog zu Producer-Methoden parametrisiert werden.

Die Klasse „DataSourceProducer" demonstriert die Verwendung von Producer-Feldern, wie sie typischerweise im Umfeld von EJBs vorkommen. Mittels „@Resource" und der Angabe eines JNDI-Namens wird eine DataSource injiziert, die durch die Annotation „@Produces" wiederum für CDI-Injection-Points zur Verfügung steht. Dieses Vorgehen erscheint auf den ersten Blick etwas widersinnig. Der Vorteil besteht allerdings darin, die Streuung des JNDI-Namens über verschiedene Klassen, die diese DataSource verwenden, zu vermeiden. Es existiert also nur noch eine zentrale Stelle in der Anwendung, an der er deklariert sein muss. Eine Umbenennung oder der Ersatz durch eine andere DataSource (mit oder ohne JNDI) kann nun relativ einfach durch eine Anpassung oder den Austausch des Producer erfolgen.

Das Producer-Feld ist beispielhaft mit einem Qualifier („@UserData") versehen. Dies ist natürlich auch für Producer- und Dispose-Methoden möglich. Auf diesem

```
InMemoryServiceImpl.java:
      @InMemory
      @ApplicationScoped
      public class ServiceImpl implements IService {
           @Inject
          private ConnectionFactory connectionFactory

          private Connection connection;

          @PostConstruct
    public void init() {
        this.connection = connectionFactory.createConnection();
    }
          @PostConstruct
          public void destroy() {
             this.connection.close();
          }
      }
```

Listing 6

Wege können verschiedene DataSource-Instanzen voneinander unterschieden werden. Der Einsatz der Qualifier-Annotation bietet gegenüber einem JNDI-Namen den Vorteil einer höheren „Refactoring-Sicherheit", da eine Umbenennung im Zweifelsfall zu Fehlern während des Kompilierens führt und nicht erst zur Laufzeit.

Scope und Lebenszyklus

Am Beispiel der Producer-Methoden hat sich bereits gezeigt, dass eine existierende Dispose-Methode zum geeigneten Zeitpunkt vom Container aufgerufen wird. Doch wann ist dieser Zeitpunkt?

Jede durch den Container erzeugte Bean-Instanz ist eindeutig einem sogenannten „Scope" zugeordnet. Dieser verfügt über einen Lebenszyklus (also Erzeugung und Zerstörung), der durch Ereignisse innerhalb des Containers bestimmt ist, also etwa Start und Stopp einer Applikation. Beim Zugriff auf eine injizierte Bean an einem Injection-Point („@Inject") wird geprüft, welchem Scope sie zugeordnet werden soll und ob in diesem bereits eine Instanz existiert. Ist dies der Fall, wird diese verwendet, anderenfalls eine neue erzeugt und im entsprechenden Scope abgelegt. Erreicht dieser das Ende seines Lebenszyklus (etwa durch Stopp der Applikation), werden die darin enthaltenen Bean-Instanzen ebenfalls zerstört. Im Falle der Erzeugung der Instanzen über eine Producer-Methode wird also zu diesem Zeitpunkt eine gegebenenfalls existierende Dispose-Methode aufgerufen.

Wird eine Bean durch den Container direkt verwaltet (also ohne den Umweg über Producer), können in ihr sogenannte „Lifecycle-Callbacks" (also mit „@PostConstruct" beziehungsweise „@PreDestroy" annotierte Methoden ohne Parameter beziehungsweise Rückgabewert) implementiert werden, die bei Erzeugung beziehungsweise Zerstörung der Instanz durch den Container aufgerufen werden. Sie eignen sich hervorragend, um benötigte Ressourcen wie JMS-Connections zu allokieren beziehungsweise wieder freizugeben. Listing 6 zeigt ihre Verwendung.

```
UIController.java:
      public class UIController {

          @Inject
          private Conversation conversation;

          public void begin() {
        conversation.setTimeout(TimeUnit.MINUTE.asMillis(5));
        conversation.begin();
        … // load conversation data
    }
    public void end() {
              … // store conversation data
              conversation.end();
         }
      }
```

Listing 7

Der zu verwendende Scope für eine Bean wird durch eine entsprechende Annotation auf Klassenebene oder am jeweiligen Producer (Methode oder Feld) festgelegt, im Beispiel wurde bereits „@ApplicationScoped" verwendet. Folgende Scopes sind durch CDI vordefiniert:

- *@RequestScoped*
  Dauer einer Anfrage (wie HTTP-Request, Konsumieren einer JMS-Nachricht)
- *@SessionScoped*
  Lebensdauer einer Sitzung (wie HTTP)
- *@ConversationScoped*
  Durch die Anwendung bestimmte Dauer einer Konversation/Interaktion im Rahmen einer Sitzung, die sich über mehrere Anfragen (Requests) erstrecken kann (wie Wizard oder modaler Dialog in einer Web-Anwendung)
- *@ApplicationScoped*
  Start und Stopp einer Applikation im Container (wie Deployment/Undeployment einer WAR/EAR-Datei)
- *@Singleton*
  Definiert ein klassisches Java-Singleton,

```
UIController.java:
      @Named // can be referenced by EL expressions using #{userSettings}
      @SessionScoped
      public class UserSettings {
        ...
      }
      public class SystemSettings {
        @Produces
        @Singleton
        @Named(„systemLocale") // can be referenced by EL expressions using #{systemLocale}
        public Locale getDefaultLocale() {
          ...
        }
      }
```

Listing 8

es ist also im Java-EE-Container an den Lebenszyklus des Applikations-Classloader gekoppelt. Damit ergibt sich eine große Ähnlichkeit zu „@ApplicationScoped", es existiert allerdings ein Unterschied bei der Injizierung in andere Beans, der noch erläutert wird.

- *@Dependent*
  Übernahme des Scope vom jeweiligen Injection-Point einer erzeugten Instanz. Dies ist bei Injizierung in ein Feld einer Bean-Instanz beziehungsweise im Falle einer Producer-Methode der jeweils für diese Bean deklarierte Scope. Dieser kann selbst wiederum „@Dependent" sein, auf diese Art können ganze Abhängigkeitsketten entstehen. Verfügt eine Bean oder ein Producer über keine der aufgezählten Annotationen, werden die entsprechenden Instanzen implizit als „@Dependent" betrachtet.

Eine Besonderheit für „@Dependent" und „@Singleton" besteht darin, dass die erzeugten Instanzen im Gegensatz zu allen anderen sogenannten „normalen Scopes" direkt, also ohne Verwendung dynamischer Proxies, in den Konsumenten injiziert werden und so beispielsweise eine Verwendung von Interzeptoren nicht möglich ist.

Es bleibt noch die Frage offen, wie der Lebenszyklus einer Konversation zur Arbeit mit „@ConversationScoped"-Beans gesteuert werden kann. Eine Konversation ist an eine umgebende Sitzung gekoppelt und überlebt mehrere Anfragen. Sie muss explizit begonnen und beendet werden (siehe Listing 7). Pro Sitzung existiert nur jeweils eine aktive Konversation. Wird eine Konversation begonnen, aber nicht beendet, wird sie nach Ablauf eines konfigurierbaren Timeout automatisch verworfen.

Enterprise Java Beans

Die CDI-Spezifikation sieht die Integration mit anderen Technologien aus dem Umfeld von Java EE 6 vor. Das betrifft vor allem die Arbeit mit Enterprise Java Beans. Sie können über ihr Business-Interface via „@Inject" oder „@EJB" in CDI-Beans injiziert werden. Dies gilt ebenso für technische Ressourcen („@Resource", „@PersistenceContext").

Umgekehrt kann „@Inject" in EJBs verwendet werden, im Falle von Stateful Session Beans (SSB) ist darüber hinaus die Deklaration von Scopes möglich. Letzteres ersetzt die Verwendung einer mit „@Remove" annotierten Methode, die SSB-Instanz wird automatisch bei der Zerstörung des jeweiligen Scope verworfen.

Diese beidseitige Integration sorgt am Anfang oftmals für Verwirrung: Wann sollte man nun EJBs und wann CDI-Beans verwenden? Ein kleiner Perspektiv-Wechsel birgt die Antwort in sich: EJBs können als CDI-Beans aufgefasst werden, die mit speziellen Eigenschaften ausgestattet sind – sie können also Transaktionen öffnen beziehungsweise schließen, Autorisierung erzwingen etc. Dieser Mehrwehrt ist in der Regel nur an wenigen definierten Stellen innerhalb einer Anwendung notwendig.

In der klassischen Drei-Tier-Architektur ist dies die Schnittstelle der Anwendungslogik zu anderen Schichten beziehungs-

weise Modulen – das Stichwort lautet „EJB-Facade". Im bildlichen Sinne kommen darüber (etwa JSF-Controller- und Model-Beans) und darunter (etwa DAOs, Berechnungslogik) wiederum nur CDI-Beans zur Anwendung, die in ihrer Natur wesentlich leichtgewichtiger als EJBs sind.

Java Server Faces

Ein maßgeblicher Treiber für die Entwicklung von CDI war die Schaffung eines einfacheren Programmiermodells für JSF. Dies lässt sich gut an den vordefinierten Scopes („@RequestScoped", „@SessionScoped", „@ConversationScoped") erkennen. Leider besteht auch hier wieder genügend Raum für Verwirrung, da die JSF-Spezifikation eigene Annotationen für sogenannte „Managed Beans" und „Managed Properties" vorsieht, die sich aber nicht mit CDI integrieren und bei denen darüber hinaus sogar Namensdopplungen mit CDI-Scopes bestehen. Das bedeutet, dass für die Konzeption einer JSF-basierten Anwendung im Java-EE-6-Kontext eine Entscheidung für den einen oder den anderen Weg getroffen werden muss. Aufgrund des deutlich einfacheren Programmiermodells fällt die Empfehlung klar für CDI aus, obwohl hierfür leider kein „@ViewScoped" vordefiniert ist. Dieser durchaus nützliche Scope kann aber durch eine entsprechende CDI-Erweiterung leicht nachgerüstet werden.

Um CDI-Beans oder EJBs für die Referenzierung in Expression-Language (EL)-Ausdrücken in Views zugänglich zu machen, kommt die Annotation „@Named" zum Einsatz. Diese kann an Bean-Deklarationen oder Producer-Feldern beziehungsweise -Methoden eingesetzt werden (siehe Listing 8).

Fazit

CDI definiert ein relativ abstraktes, aber sehr konsequentes Modell zur Arbeit mit Dependency Injection und verlangt dafür von seinem Nutzer Verständnis für die zu lösenden Probleme und angebotenen Konzepte. Wer sich darauf einlässt und das notwendige Vertrauen in die Container-Infrastruktur entwickelt, gewinnt ein hohes Maß an Flexibilität und kann seinen Code weitestgehend frei von komplizierten technischen Konstrukten zur Entkopplung von Diensten und deren Konsumenten halten.

Darüber hinaus erhält er weitere, noch nicht näher benannte Boni: Angerissen wurden bereits die Möglichkeiten zur Arbeit mit Interceptoren und Delegates sowie die Umsetzung des Observer-Patterns zum Feuern und Konsumieren von Ereignissen (Events). Darüber hinaus definiert CDI eine umfangreiche Erweiterungsschnittstelle, die unter anderem die Definition eigener Scopes ermöglicht. Auf diese Extras wird in den folgenden Ausgaben im Detail eingegangen.

Dirk Mahler
dirk.mahler@buschmais.com

Dirk Mahler ist als Senior Consultant auf dem Gebiet der Java-Enterprise-Technologien tätig. In seiner täglichen Arbeit setzt er sich mit Themen rund um Software-Architektur auseinander, kann dabei eine Vorliebe für den Aspekt der Persistenz nicht verleugnen.

Distributed Java Caches

Dr. Fabian Stäber, ConSol* Consulting & Solutions Software GmbH

Mit Ehcache, Infinispan und Hazelcast gibt es gleich drei Java-Produkte, die Cluster-fähige Caches zur Verfügung stellen. Alle drei Implementierungen bieten eine ähnliche Schnittstelle, unterscheiden sich jedoch erheblich in den zugrunde liegenden Architekturen. Dieser Artikel stellt diese vor und erklärt, welche Architektur sich für welche Anforderungen am besten eignet.

Horizontale Skalierbarkeit und Cluster-Fähigkeit gehören inzwischen zu den Standardanforderungen bei der Entwicklung neuer Java-Server-Anwendungen. Es wird erwartet, dass bei steigenden Zugriffszahlen die höhere Last durch das Starten neuer Server-Instanzen abgefangen werden kann. Diese Entwicklung bringt auch neue Anforderungen an Caches mit sich: Es genügt nicht mehr, Caches als einfache, lokale Speicher zu implementieren. Im Cluster-Betrieb müssen „gecachte" Daten synchronisiert werden, sodass Cluster-weite Daten-Konsistenz über alle Instanzen hinweg gewährleistet werden kann. Um diese Anforderungen zu erfüllen, haben sich moderne Java-Caches zu mächtigen In-Memory-Data-Management-Systemen entwickelt, die verteilte atomare Operationen sowie verteilte Transaktionen unterstützen. Nachfolgend sind drei ver-

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN
0700 11 36 24 39

ODER BESTELLEN SIE ONLINE
go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr
☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:
Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.