Studienarbeit

- Thomas Kraft -
MatNr. 011270

Betreut von Prof. Dr.-Ing. Holger Vogelsang

JAVA WEB START

"Sun arbeitet hart an der Java-Technologie für den Desktop und verbessert weiterhin die Performance und vereinfacht die Nutzung. Java Web Start ist ein großer Schritt vorwärts in Richtung der Möglichkeit, webfähige grafische Applikationen in der heute allgegenwärtigen vernetzten Welt zu verteilen"

Rich Green, Vice President Sun Microsystems.

Inhalt:

1. Vorwort

„With a single click“ – mit diesem Schlagwort beschreibt Sun Ihre plattformunabhängige, webzentrierte Deployment-Technologie Java Web Start (JWS).
An diesem Schlagwort kann man schön erkennen, welche Ziele Sun mit dieser neuen Technologie verfolgt. JWS soll die Installation, das Update und den Start einer beliebigen Java-Applikation auf dem Client durch einen einfachen Klick auf einen Link innerhalb einer Webseite ermöglichen.

In dieser Studienarbeit über Java Web Start möchte ich die Architektur von JWS beschreiben und die Vorteile dieser neuen Technologie aufzeigen. Ich werde versuchen, die zeitliche Entwicklung von Java auf dem Client bis heute aufzuzeigen und einen kurzen Ausblick in die Zukunft zu geben, welche Möglichkeiten sich mit JWS auf dem Client bieten. Dabei werde ich Anwendungsbeispiele aufzeigen und versuchen, die Vorteile von JWS gegenüber bisherigen Lösungen herauszuarbeiten.

Als Abschluss werde ich Schritt für Schritt eine kleine Beispielapplikation entwickeln, um die neuen Möglichkeiten von JWS aufzuzeigen. Dabei sollen insbesondere die Vorteile gegenüber Applets hervorgehoben werden, wie Zugriffe auf das Lokale Dateisystem des Clients und das Caching der Programmdateien.

Genau genommen ist Java Web Start die Referenzimplementierung von Sun eines JNLP-Clients. JNLP steht für „Java Network Launching Protocol“ und ist die Grundlage von Java Web Start.

2. Die Evolution von Java

2.1 Der Anfang

Wie ist Java enstanden? Begonnen hat alles damit, dass Sun eine Steuerungssprache für Haushaltsgeräte entwickelte. Der Name dafür war recht schnell gefunden. Der Chefentwickler schaute in seinem Büro zum Fenster hinaus und direkt davor stand eine große Eiche, also nannte man die neue Sprache „Oak“. Oak sollte über eine gemeinsame Steuerungskonsole alle Geräte eines Haushalts mit einer gemeinsamen Sprache (fern-) bedienen, vom Toaster über den Fernseher bis hin zum Rechner. Demzufolge war die Sprache auch auf geringen Speicherbedarf, hohe (Absturz-) Sicherheit und einfache Programmerstellung ausgelegt.

Allerdings war die Zeit damals noch nicht ganz reif für eine solche Vision. Glücklicherweise lies sich das große Potential von Oak auch auf eine ganz andere Weise nutzen: Wie wäre es, wenn man einen WWW-Browser als unabhängiges, von anderen Geräten abgekapseltes System betrachtet? Man könnte dieses System durch das Oak-System erweitern und zur Steuerung von Aufgaben benutzen, die es von sich aus nicht bewältigen kann. So entstand dann schließlich Java.

1994 wurde dann von Sun ein eigener Browser mit Java-Unterstüzung namens HotJava vorgestellt, kurz danach folgte das erste Java Development Kit, mit dem endlich jedermann problemlos eigene Java-Anwendungen schreiben konnte.

2.2 Java und der Client

Damit begann dann auch endlich der Siegeszug von Java auf dem Client. Durch die Integration einer Java VM (Virtual Machine) in die großen, marktführenden Browser war eine breite Unterstützung der neuen Technologie gewährleistet. Dadurch sahen sich viele Entwickler angespornt, kleine Programme zu schreiben, die innerhalb eines Browsers ablauffähig waren – die sogenannten Java-Applets. Vor gar nicht allzu langer Zeit schmückte sich fast jede Website, die Up to Date sein wollte mit diesen Miniprogrämmchen, sei es als kleines Gimmick wie Spiele oder als grundlegender Teil der Seitennavigation wie Menüs oder ähnlichem.

Diese Miniprogrämmchen haben den großen Vorteil, dass sie ihren Programmcode vollständig über das Netz herunterladen können, also sind keinerlei Installationsarbeiten auf dem Client von Nöten. Es reicht völlig aus, einen aktuellen Browser zu haben und der Rest passiert vom Anwender unbemerkt im Hintergrund. Sie stellen also eine sehr gute Möglichkeit dar, kleinere Applikationen einem großen Userkreis schnell und einfach zur Verfügung zu stellen.

Wenn man die Möglichkeiten, die einem ein Applet bietet mit den Möglichkeiten vergleicht, die normales HTML bietet, versteht man auch warum die Applets sich so schnell verbreitet

haben. Sie können auf einen reichen Fundus von Funktionen z.B. für User Interfaces zurückgreifen.

Applets sind in HTML-Seiten eingebunden und laufen aus Sicherheitsgründen innerhalb einer speziellen Umgebung ab, der sogenannten Sandbox, meist innerhalb eines Web-Browsers. Diese Sandbox kapselt das Applet komplett vom dahinterliegenden System ab, so dass z.B. keinerlei Zugriffe auf das lokale Dateisystem oder lokal angeschlossene Peripherie möglich sind.

Im Laufe der Jahre wurde das am Anfang noch strikt eingehaltene „Write Once, Run Everywhere“ – Paradigma mehr und mehr durch diverse Betriebssystem- und Browserhersteller untergraben, so dass Entwickler sich heute nicht mehr sicher sein können, dass ihr Applet auch wirklich auf allen Maschinen läuft und genau so aussieht, wie sie es sich vorgestellt haben.

Ein weiterer gravierender Nachteil von Applets ist der, dass diese Programme clientseitig nicht gecached werden, so dass sie bei jedem Aufruf der Seite wieder komplett vom Server an den Client übertragen werden müssen, was unter Umständen sehr lange dauern kann und einen hohen Netzwerkverkehr verursacht.

In den letzten Jahren konnten wir deshalb beobachten, wie die Java-Applets mehr und mehr von den Webseiten verschwanden, jedoch gewann Java in gleichem Maße auf dem Server mehr und mehr an Bedeutung. Web Services ist nur ein Stichwort, welches die wichtige Rolle von Java auf dem Server beschreibt.

2.3 Java und der Server

In letzter Zeit wurde Java als serverseitige Programmiersprache immer beliebter. Die clientseitigen Inkompatibilitäten können durch die Logik-Verlagerung auf den Server sehr schön umgangen werden. Durch JSPs, Servlets, Beans bzw. EJBs in Verbindung mit einem Application Server bieten sich dem Web-Developer mächtige Werkzeuge, um seine Anforderungen im Bereich Business Logik umzusetzen. Auch können so recht einfach Strukturen geschaffen werden, die es komfortabel ermöglichen, z.B. den Inhalt einer Seite zu aktualisieren und komplexe Strukturen einzupflegen.

Der Nachteil dieser serverseitigen Verlagerung der Logik ist die nicht zufriedenstellende Komplexität und Auswahl an möglichen Userinterfaces, da diese an die jeweilige Markup-Language (meist HTML) gebunden sind und damit weit weniger komplexe Interaktionsmöglichkeiten mit dem Anwender gestatten, als dies bei einer Applikation der Fall wäre. Weiterhin ist durch die komplette Verlagerung der Anwendungslogik auf den Server bei jeder Interaktion eine Kommunikation mit diesem nötig, was dazu führt, dass diese Anwendungen nicht offline ausgeführt werden können und je nach Art der Anwendung auch während des Betriebes eine recht hohe Netzlast erzeugt wird, was unter Umständen je nach verfügbarer Infrastruktur ein Problem darstellt.

Wie bei Applets ist es bei diesen serverseitigen Anwendungen ebenfalls nicht möglich auf lokal angeschlossene Peripherie zuzugreifen bzw. Daten auf dem lokalen Dateisystem zu speichern. Java Web Start bietet hier neue Möglichkeiten der Applikationsentwicklung, welche im Folgenden aufgezeigt werden.

3. Java Web Start – Die Revolution auf dem Client?

3.1 Die Vorteile des Anwenders

Für den Anwender eröffnen sich durch Java Web Start ganz neue, ungeahnt einfache Möglichkeiten, eine Applikation über das Netz zu starten. Im Gegensatz zu normalen Applikationen muss der Anwender sich nicht selber um diverse Details wie den Download, die Installation oder den Speicherort kümmern. All diese Aktionen übernimmt Java Web Start für ihn. Der Anwender muss nur einen Hyperlink anklicken und Java Web Start kümmert sich selbständig darum, dass alle zum Programm gehörenden Klassen geladen werden und die heruntergeladenen Dateien im lokalen Dateisystem gecached werden, so dass sie bei einem erneuten Start der gleichen Applikation nicht noch einmal übers Netz geladen werden müssen.

Nachdem die Applikation heruntergeladen wurde, gibt es für den Anwender auch die Möglichkeit, eine Verknüpfung auf dem Desktop oder in seinem Startmenü anlegen zu lassen, so dass er für den nächsten Start der Applikation nicht einmal mehr seinen Browser starten muss.

Trotz dieser recht einfachen Mechanismen zum Start einer solchen Anwendung muss der User nicht auf die von Java gewohnten Sicherheitsmechanismen verzichten. Standardmäßig laufen auch alle JWS-Applikationen zuerst in einer Sandbox ab, die sie vom System abkapseln. Falls die Anwendung darüber hinaus noch zusätzliche Rechte will, z.B. um Dateien ins lokale Dateisystem abzulegen oder die Zwischenablage zu benutzen, bekommt der User automatische eine Warnung und kann es der Anwendung dann entweder erlauben oder verbieten, die jeweiligen Aktionen durchzuführen. Eine Ausnahme machen da signierte Anwendungen, welche nach einmaliger Zustimmung des Anwenders beim Start der Applikation, vollen Zugriff auf das Client-System haben und somit beispielsweise auch auf die Festplatte zugreifen können, ohne das für jede Aktion eine extra Sicherheitsmeldung angezeigt wird.

JWS enthält auch den „Java Web Start Application Manager“, der eine zentrale Verwaltungseinheit für alle auf dem System installierten JWS-Applikationen darstellt. Der Application Manager hat ein plattformunabhängiges Look & Feel und ist zentraler Bestandteil von JWS. Über diesen Application Manager hat der User die Möglichkeit, alle bereits lokal installierten Anwendungen und Informationen über sie zu sehen, und diese auch wieder zu löschen. Auch kann er hier grundlegende Einstellungen für Java Web Start machen, wie Root-Zertifikate oder Verknüpfungsoptionen.

3.2 Die Vorteile des Entwicklers

Der Entwickler hat bei JWS die Möglichkeit, den vollen Funktionsumfang der Java API zu benutzen. Er muss sich dabei nicht darum kümmern ob bei dem Endanwender alle benötigten Bibliotheken oder auch das benötigte Java Runtime Environment installiert sind. Java Web Start prüft beim Start einer Applikation nach, ob alle benötigten Komponenten auf dem Client-System verfügbar sind, und installiert bei Bedarf alle benötigten Ressourcen „On The Fly“ nach.

Auch bei eventuellen Updates seiner Applikation, seien es komplette Updates oder auch nur inkrementelle, hat der Programmierer einen vergleichsweise sehr niedrigen Arbeitsaufwand. Er muss einfach nur die JAR-Dateien auf dem Server ersetzen, und Java Web Start wird bei dem nächsten Start der Applikation mit Hilfe des HTTP-Timestamps überprüfen, welche Ressourcen in aktualisierter Form auf dem Server zur Verfügung stehen und diese dann automatisch auf dem Client aktualisieren, so dass der Endanwender immer die aktuellste Version der Applikation hat, ohne dass er sich darum kümmern muss. Zum Verbreiten von Updates oder Patches ist das wohl die denkbar einfachste Lösung.

Serverseitig muss sich der Entwickler auch nicht mit irgendwelchen Speziallösungen herumschlagen, sondern kann praktisch jeden 08/15-Server benutzen. Das einzige was der Entwickler beachten muss ist, dass auf dem Server die MIME-Types aktualisiert werden, so dass er die `.jnlp`-Dateien kennt. Möglich macht dies das neue Protokoll, welches Sun für JWS entwickelt hat, nämlich das Java Netwok Launching Protokoll (JNLP), welches später noch detaillierter betrachtet wird.

Auch eventuelle Zugangsbeschränkungen für einen ausgesuchten Benutzerkreis lassen sich so recht einfach mit herkömmlichen Web-Techniken erreichen. Java Web Start unterstützt standardmäßig die HTTP - Autorisation und wird den Benutzer um eine Anmeldung bitten, falls der Server dies erfordert. Auch etwas komplexere Schutzmechanismen wie dynamisch erzeugte `.jnlp`-Dateien mit einem benutzerspezifischen Argument sind denkbar, wenn auch ein wenig komplizierter zu implementieren. In diesem Falle müsste der Anwender dann die Applikation mittels eines Lizenzstrings oder ähnlichem erst freischalten, bevor er sie in vollem Umfang nutzen kann.

4. JNLP – Das Herz von Java Web Start

4.1 Was ist JNLP?

JNLP, die Abkürzung von „Java Network Launching Protokoll“ ist das Herzstück von Java Web Start und beschreibt alle wichtigen Eigenschaften einer JWS Applikation.
Genau genommen ist Suns Java Web Start nur eine mögliche Implementierung eines JNLP Clients. Denkbar, und auch in Ansätzen schon verfügbar, sind auch andere, nicht von Sun stammende, JNLP Clients. Beispiele hierfür sind die Projekte Open JNLP und NetX, welche beide bei SourceForge gehostet werden.

Ein JNLP-Client ist genau genommen nur eine Software, welche die Spezifikation des Java Network Launching Protokolls implementiert. Dieser Client muss eigentlich nur die benötigten Ressourcen vom Netz laden, die Laufzeitumgebung von Java hochfahren und den Einsprungpunkt der Applikation an diese übergeben.

Die Spezifikation von JNLP umfasst unter anderem:

- Ein webzentriertes Applikationsmodell ohne offensichtliche Installationsphase, welches ebenso transparente und inkrementelle Updates der Applikation ermöglicht.
- Ein Protokoll, durch welches eine Applikation mittels einer `.jnlp`-Datei auf einem Web-Server an einen beliebigen JNLP-Client (z.B. Java Web Start) übertragen und ausgeführt werden kann.
- Eine Laufzeit-Umgebung für die Applikation, in der sowohl eine sichere Umgebung für nicht-signierte Anwendungen (Sandbox) als auch eine unbeschränkte Umgebung enthalten ist, anhand derer der uneingeschränkte Festplatten- und Peripheriezugriff möglich ist.

4.2 Die .jnlp-Datei

Die `.jnlp`-Datei ist das Herzstück von Java Web Start. Bei dieser Datei handelt es sich eigentlich um ein XML-Dokument, welches die dazugehörige Applikation (`application descriptor`) oder die entsprechende Erweiterung (`extension descriptor`) beschreibt. In dieser Datei stehen alle für die Programmausführung relevanten Daten, die im Weiteren kurz erläutert werden.

Aus Abbildung 4.1 können sie den grundsätzlichen Aufbau einer solchen `.jnlp`-Datei erkennen.

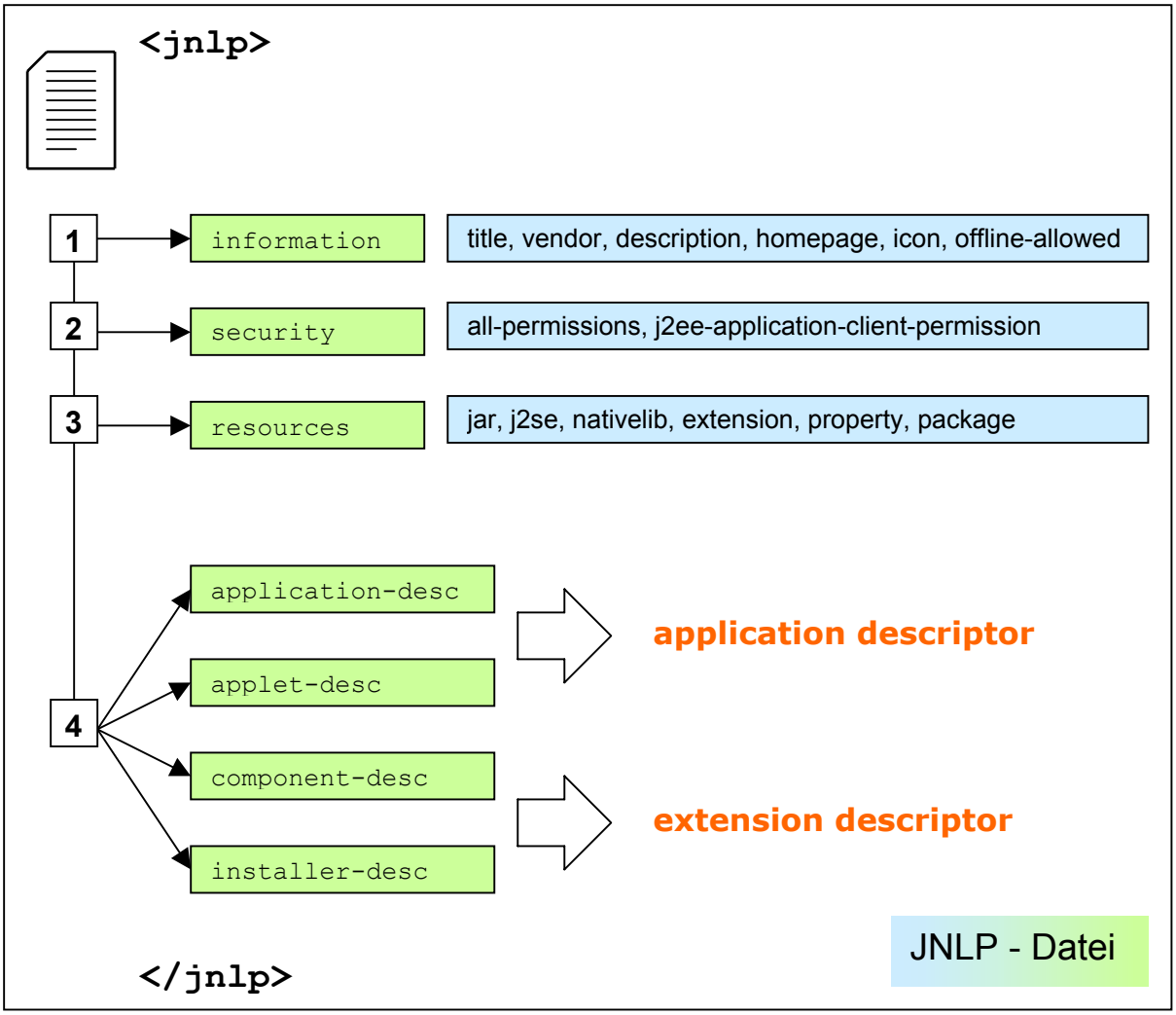


Abb. 4.1

Eine `.jnlp`-Datei enthält folgende grundlegende Informationen:

- JNLP Version und die URL der `.jnlp`-Datei selbst
- Allgemeine Information über die Applikation
- Benötigte JRE Versionen und Erweiterungen
- Sicherheitsanforderungen
- Speicherort der Ressourcen der Applikation
- Applikationskonfiguration

Auf den folgenden Seiten werden der genaue Aufbau der `.jnlp`-Datei im Detail erläutert, die einzelnen XML-Elemente kurz vorgestellt, und ihre Syntax und Funktion aufzeigt.

4.2.1 Das JNLP-Root Element <jnlp>

Das Root-Element `<jnlp>` hat 6 mögliche Subelemente: `information`, `security`, `jre`, `resources`, `application-desc` und `extension-desc`. Die letzten zwei Elemente schliessen sich gegenseitig aus und beschreiben ob die `.jnlp`-Datei ein Applikationsdeskriptor (`application-desc`) oder ein Erweiterungsdeskriptor (`extension-desc`) ist.

Als kleines Beispiel möchte ich einen Auszug eines Applikationsdeskriptors aufzeigen:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0" codebase="http://..." href="MyApp.jnlp" version 1.2">
  <information> ... </information>
  <security> ... </security>
  <jre> ... </jre>
  <resources> ... </resources>
  <application-desc> ... </application-desc>
</jnlp>
```

4.2.1.1 Die Attribute des JNLP-Root Elements <jnlp>

`spec`: *(Attribut)*Beschreibt die Version der JNLP Spezifikation die benutzt wird. Für die Version 1.0 der Spezifikation muss der Wert des Attributes „`1.0`“ sein. Der Wert dieses Attributes sollte von den JNLP-Clients auf 1.0 interpretiert werden, falls er nicht explizit angegeben ist.

`codebase`: *(Attribut)* Definiert die Codebase für die Applikation. Diese Codebase wird als Basis URL für alle `href` Elemente innerhalb einer JNLP Datei verwendet. So ist es möglich z.B. nur den Namen eines Elements anzugeben, wenn die Codebase bereits auf das entsprechende Verzeichnis verweist. Auch kann diese Codebase von der Applikation über die JNLP API ausgelesen werden.

`href`: *(Attribut)* Beschreibt den Speicherort der `.jnlp`-Datei selbst als HTTP-URL.

`version`: *(Attribut)* Beschreibt die Version der `.jnlp`-Datei bzw. der zugehörigen Applikation.

4.2.2 Das Informations Element <information>

Das Informationselement enthält Informationen, die speziell darauf ausgelegt sind dem JNLP-Client Informationen zu geben, damit dieser die Applikation besser in den Desktop integrieren kann, dem User ein Feedback zu geben oder einfach nur zu informieren.

Ein kleines Beispiel:

```
<information>
  <title>Cool App 1.0</title>
  <vendor>My Corporation</vendor>
  <description>Helps you to keep cool</description>
  <description kind="tooltip">CoolApp</description>
  <homepage href="doc/index.html"/>
  <icon href="icon.gif"/>
  <offline/>
</information>
<information locale="da_DK">
  <description>Lidt for koldt?</description>
  <description kind="tooltip">Køligt</description>
</information>
```

4.2.2.1 Die Attribute und Elemente des <information> Elements

`locale`: *(Attribut)* Mit diesem Attribut wird die lokale Einstellung des JNLP Clients beeinflusst. Dieser sucht beim Start einer Applikation nach den lokalen Einstellungen der Applikation beruhend auf den lokalen Einstellungen des ausführenden Systems. Das `locale`-Attribut besitzt unter anderem den `language identifier` (z.B. en für Englisch) einen `country identifier` (z.B. US für USA) und eine mögliche Angabe einer Variante.
Die Syntax dafür lautet:

```
locale ::= language [ "_" country [ "_" variant ] ]
```

Die Attribute des `<information>` Tags werden auf Übereinstimmung mit den `locale`-Einstellungen des Clients durchsucht um ein möglichst perfektes Ergebnis zu bekommen. Die Suchreihenfolge ist wie folgt:

- Versuche einen passenden Eintrag für `language`, `country` und `variant` zu finden, falls diese alle für dieses Attribut angegeben sind
- Versuche einen passenden Eintrag für `language` und `country` zu finden, falls diese beide angegeben sind
- Versuche einen passenden Eintrag für `language` zu finden
- Keine `locale` - Attribute sind angegeben worden oder keine Übereinstimmung gefunden

Wenn mehrere `locale` - Attribute mit den gleichen Werten angegeben werden ist das Ergebnis undefiniert.
Im obigen Beispiel wurde die Beschreibung für Dänisch eingefügt. D.h. falls auf einem ausführenden Rechner die lokalen Einstellungen für Dänisch gesetzt werden, bekommt der User den Dänischen Text angezeigt, in allen anderen Fällen den Englischen.

`title`: *(Element)* Der Name der Applikation.

`vendor`: *(Element)* Der Hersteller / Verkäufer der Applikation.

`homepage`: *(Element)* Enthält ein einziges Attribut, `href`, welches eine URL ist die auf eine Homepage für die Applikation zeigt. Diese URL wird vom JNLP-Client benutzt um dem User eine Homepage aufzuzeigen, wo er nähere Informationen über die Applikation findet.

`description`: *(Element)* Eine kurze Beschreibung der Applikation. Description-Elemente sind optional. Das „`kind`“ - Attribut beschreibt, in welcher Art die Beschreibung benutzt werden soll, es kann eines der folgenden Werte haben:

- **`one-line`:** Wenn ein Hinweis für die Applikation in einer Zeile einer Liste oder einer Tabelle angezeigt werden soll, wird diese Beschreibung benutzt.
- **`short`:** Wenn ein Hinweis für die Applikation angezeigt werden soll und Platz für einen Absatz ist, wird diese Beschreibung benutzt.
- **`tooltip`:** Eine Beschreibung der Applikation die als Tool-Tip benutzt wird.

Nur jeweils ein `description` - Element jedes Typs darf benutzt werden. Ein `description` - Element ohne Angabe des `kind` - Attributes wird als Standardbeschreibung benutzt. So z.B. wenn eine JNLP-Client eine Beschreibung vom Typ „`short`“ anzeigen will aber keine vorhanden ist, so wird der JNLP-Client die Beschreibung anzeigen, bei der keine Angabe des „`kind`“ - Attributes vorhanden ist.
Alle `description` - Elemente enthalten Plaintext, d.h. keinerlei Formatierungsangaben wie z.B. HTML sind erlaubt.

`icon`: *(Element)* Enthält eine HTTP - URL zu einer Grafik, entweder im GIF oder im JPG Format. Diese Grafik kann vom JNLP-Client dazu benutzt werden die Applikation für den User identifizierbar zu machen, also das Icon zu verwenden. Das `icon` - Element hat 5 optionale Attribute:

- **`width`:** Wird benutzt um eine Größenangabe für das Icon zu machen (hier: Iconbreite) Angabe in Pixel.
- **`height`:** Wird benutzt um eine Größenangabe für das Icon zu machen (hier: Iconhöhe) Angabe in Pixel.
- **`depth`:** Wird benutzt um die Farbtiefe des Icons anzugeben.
- **`kind`:** Wird benutzt um die Verwendung des Icons anzugeben. Mögliche Werte dieses Attributes: `default`, `selected`, `disabled` und `rollover`.

Der JNLP-Client sollte ein Icon mit der Auflösung von 32 x 32 Pixel mit 256 Farben erwarten.

`offline`: *(Element)* Das optionale `offline` - Element wird benutzt, um zu beschreiben ob die Applikation auch benutzt werden kann, wenn der Client keine Verbindung zum Internet hat. Standardmäßig läuft eine Web-Start Applikation nur mit Verbindung zum Internet.
Die online/offline Information kann vom JNLP-Client entweder dazu benutzt werden um den User zu informieren, um zu verhindern dass eine Applikation gestartet wird die ohne Internet nicht lauffähig ist, oder er kann dieses Attribut einfach ignorieren. Eine Applikation kann niemals sicher sein, nicht gestartet zu werden, ohne dass eine Internet - Verbindung besteht, auch wenn dieses Attribut nicht angegeben ist.

4.2.3 Das Sicherheits Element <security>

Jede Applikation wird standardmäßig in einer gesicherten Umgebung ausgeführt, welche ähnlich der von Applets bekannten Sandbox ist. Das `security` - Element kann dazu benutzt werden weitergehende Rechte vom Client anzufordern.

Ein kleines Beispiel:

```
<security>
  <all-permissions/>
</security>
```

4.2.3.1 Das Element des <security> Elements

Es gibt nur ein erlaubtes Subelement des `security`-Elements:

`all-permissions`: *(Element)* Wenn dieses Element angegeben wird, bekommt die Applikation vollen Zugriff auf den ausführenden Rechner und das lokale Netzwerk. Ein JNLP-Client kann der Applikation jedoch nur dann vollen Systemzugriff erlauben, wenn bestimmte Rahmenbedingungen erfüllt sind:

- Die Applikation ist signiert (dazu später mehr)
- Der User und/oder der JNLP-Client vertraut den Zertifikaten, die benutzt wurden um die Applikation zu signieren.

In welcher Weise ein JNLP-Client einem Zertifikat vertraut, hängt ganz von der jeweiligen Implementierung des JNLP-Clients ab. Normalerweise wird ein JNLP-Client den User warnen und um Erlaubnis fragen bevor er eine Applikation startet, die vollen Systemzugriff verlangt. Diese Userentscheidung kann auch gecached werden um beim nächsten Applikationsstart nicht erneut beim User nachgefragt werden zu müssen.

4.2.4 Das JRE Element <jre>

Das `jre` - Element beschreibt, für welche Version der JRE die Applikation designed ist, sowie die Standard-Parameter für die Virtual Machine. Es können mehrere JREs angegeben werden, was dann eine Liste der unterstützten JREs darstellt, prioritätsmässig geordnet, die idealste Version am Anfang.

Ein kleines Beispiel:

```
<jre version="1.2" os="Windows" arch="x86">
  <resources> ... </resources>
</jre>
<jre version="1.3" initial-heap-size="64m"/>
```

4.2.4.1 Die Attribute und Elemente des <jre> Elements

`version`: *(Attribut)* Beschreibt die unterstützten Versionen der JRE

Die JRE kann in zwei verschiedenen Variationen angegeben werden. Sie kann entweder in einem herstellerunabhängigen Kontext angegeben werden durch Angabe einer bestimmten Plattform-Version der Java2 Plattform, oder durch Angabe der Versionsnummer einer bestimmten Implementierung der JRE durch einen anderen Hersteller.

Definition: Plattform Version

> Das ist die Version einer bestimmten Revision der Java2 Plattform. Eine Plattform Version beschreibt einen bestimmten Satz an APIs (Klassen und Interfaces), Semantiken und die Syntax der Java2 Plattform.
>
> Die Versions-ID ist in der Form „x.y“ Manchmal werden auch sogenannte „dot-dot“ Versionen freigegeben in der Form „x.y.z“. Typischerweise als Sicherheits-Update. Die aktuelle Version der JRE ist 1.4 (zum Zeitpunkt der Erstellung dieses Dokuments).
>
> Die Plattform Version einer JRE kann durch die System-Variable `java.specification.version` ausgelesen werden.

Definition: Produkt Version

> Dies ist die Version einer bestimmten Implementierung der JRE. Diese Produkt Version ist herstellerabhängig. Ein Produkt implementiert eine bestimmte Plattform Version. Die Versionen des Produktes und der dazugehörigen Plattform sind nicht zwingendermassen die gleichen.
>
> Die Produktversion kann durch die Systemvariable `java.version` ausgelesen werden.

Wenn kein `href` Attribut angegeben wird, wird die Versionsangabe als Plattform Version der Java2 Plattform interpretiert. Beispiel:

```
<jre version="1.4">
```

Der JNLP-Client kann nun irgendeine installierte JRE aussuchen, welche diese bestimmte Revision implementiert. (mit Hilfe des `java.specification.version`)

Wenn das `href` Attribut vorhanden ist, wird eine herstellerspezifische JRE angefordert. Eine spezielle JRE Implementierung wird eindeutig bezeichnet durch eine URL und eine Produktversion. Beispiel:

```
<jre href="http://java.sun.com/products/j2se" version="1.2.2-w"/>
```

Die Produktversion kann der Systemvariable `java.version` entnommen werden. Jeder JRE Hersteller ist dafür verantwortlich, eine eindeutige URL bereitzustellen, welche ihre spezielle Implementierung beschreibt.

`os`: *(Attribut)* Spezifiziert ein Betriebssystem, für welches das `jre` - Element interpretiert werden soll. Falls der Wert dieses Attributes ein Präfix der `os.name` System Variable ist, kann das `jre` - Element benutzt werden. Wenn das Attribut nicht angegeben ist, trifft es auf alle Betriebssysteme zu.

`arch`: *(Attribut)* Spezifiziert eine Rechnerarchitektur, für welche das `jre` - Element interpretiert werden soll. Falls der Wert dieses Attributes ein Präfix der `os.arch` System Variable ist, kann das `jre` - Element benutzt werden. Wenn das Attribut nicht angegeben ist, trifft es auf alle Rechnerarchitekturen zu.

`initial-heap-size`: *(Attribut)* Mit diesem Attribut kann die Anfangsgröße des Java Heap Speichers angegeben werden. Die Zusätze `m` und `k` können für Megabyte bzw. Kilobyte benutzt werden. Zum Beispiel „`128m`“ ist das gleiche wie „`134217728`“ (128*1024*1024). Die beiden Zusätze sind nicht Case - Sensitiv.

`max-heap-size`: *(Attribut)* Mit diesem Attribut kann die maximale Größe des Java Heap Speichers angegeben werden. Die Zusätze `m` und `k` können für Megabyte bzw. Kilobyte benutzt werden. Zum Beispiel „`128m`“ ist das gleiche wie „`134217728`“ (128*1024*1024). Die beiden Zusätze sind nicht Case - Sensitiv.

`resources`: *(Element)* Ein `jre` - Element kann zusätzlich noch ein eingebettetes `resources` - Element besitzen. Wenn der JNLP-Client die JRE auswählt, welche im umgebenden `jre` - Element definiert ist, werden alle Ressourcen im eingebundenen `resources` - Element geladen und zum Start der Applikation verwendet. Alle anderen `resources` - Elemente innerhalb der `.jnlp` - Datei werden ignoriert.

4.2.5 Das Ressourcen Element <resources>

Das Ressourcen Element beschreibt einen Satz von Ressourcen, welche für den Ablauf der Applikation benötigt werden, wie JAR-Dateien oder Erweiterungen.

Ein kleines Beispiel:

```
<resources>
  <jar href="lib/myjar.jar" version="1.2"/>
  <extension name="coolaudio" version="1.0"
             href="http://www.mysite.com/ext/coolaudio">
    <part name="mp3"/>
  </extension>
  <property name="key1" value="value1"/>
  <property name="key2" value="value2"/>
</resources>
```

4.2.5.1 Die Elemente des <resources> Elements

`jar`: *(Element)* Das Attribut `href` dieses Elements ist die HTTP - URL einer JAR Datei welche die Applikation zum Ablauf benötigt. Das optionale `version` Attribut beschreibt die benötigte Version der JAR Datei.

Das optionale `size` Attribut kann dazu benutzt werden, um die Downloadgröße der JAR Datei in Bytes anzugeben.

Das optionale `kind` Attribut beschreibt wie dieses JAR Element heruntergeladen werden soll. Es gibt drei verschiedene Möglichkeiten:

- **`eager`** Die JAR Datei muss vor dem Applikationsstart heruntergeladen werden.
- **`lazy`** Die JAR Datei muss nicht vor dem Applikationsstart heruntergeladen werden. (Kann sie aber trotzdem) Typischerweise wird diese JAR Datei vom JNLP-Client dann heruntergeladen wenn die JVM eine Klasse anfordert die in den bereits heruntergeladenen JAR Dateien nicht vorhanden ist.
- **`main`** Dieses `jar` - Element enthält die `main` - Klasse der Applikation. Diese JAR-Datei wird auf jeden Fall vor dem Applikationsstart heruntergeladen.

Es muss in einer `.jnlp`-Datei mindestens ein `jar` - Element als „`main`“ spezifiziert werden. Falls kein `jar` - Element als solches gekennzeichnet ist, wird das erste `jar` - Element als „`main`“ angenommen, egal was in seinem „`kind`“ Attribut steht.
Alle `jar` - Elemente ohne das „`kind`“-Attribut werden standardmäßig als „`eager`“ behandelt. Also werden alle `jar` - Elemente ohne das optionale „`kind`“-Attribut auf jeden Fall vor dem Applikationsstart heruntergeladen.

`extension`: *(Element)* Beschreibt eine zusätzliche Bibliothek, die zum Ausführen einer bestimmten Applikation benötigt werden. Eine `extension` kann plattformabhängigen nativen Code enthalten oder JAR Dateien die plattformunabhängigen und/oder plattformabhängigen Code enthalten. Dieses Element besitzt drei benötigte Attribute: `name`, `version` und `href`. Die Erweiterungen werden in einem separaten Abschnitt noch einmal genauer betrachtet.

`part`: *(Element)* Eine Erweiterung kann in mehrere Teile aufgesplittet werden, welche unabhängig voneinander heruntergeladen werden können. Das `part` - Element wird benutzt, um Teile der Erweiterung zu markieren, die vor dem Start der Applikation heruntergeladen werden müssen. Das `all` Element kann benutzt werden, was dazu führt, dass alle Teile der Erweiterung vor dem Start der Applikation heruntergeladen werden.

`property`: *(Element)* Wird benutzt um für eine Applikation spezielle System Variablen zu beschreiben. Spezielle, vordefinierte System Variablen der JVM lassen sich damit jedoch nicht überschreiben. Beispiel: Ein Versuch die Variable `java.version` zu überschreiben wird ignoriert werden.

4.2.6 Das Applikationsbeschreibungs Element <application-desc>

Eine `.jnlp` - Datei ist ein sogenannter „Application Descriptor“ wenn das `application-desc` Element vorhanden ist.
Dieses Element enthält alle Informationen die nötig sind um die Applikation zu starten.

Ein kleines Beispiel:

```
<application-desc main-class="com.example.MyMain">
  <argument>Arg1</argument>
  <argument>Arg2</argument>
</application-desc>
```

4.2.6.1 Die Attribute und Elemente des <application-desc> Elements

`main-class`: *(Attribut)* In diesem Attribut wird der Name der Klasse, welche die `public static void main(String [])` Methode enthält festgelegt. Dieses Attribut kann weggelassen werden, wenn die Main Klasse über den `Main-Class` Eintrag im Manifest der Main - JAR Datei gefunden werden kann, d.h. der JAR Datei bei welcher das `kind` - Attribut als `main` gesetzt wurde. Eventuelle Manifeste für nicht - `main` JAR Dateien werden ignoriert. Falls das `main-class` Attribut gesetzt ist, wird immer der Wert dieses Attributes benutzt unabhängig von eventuellen anderslautenden Einträgen in diversen Manifesten.

`argument`: *(Element)* Enthält eine geordnete Liste von Argumenten für diese Applikation.

4.2.7 Das Erweiterungsbeschreibungs Element <extension-desc>

Eine `.jnlp` - Datei ist ein sogenannter „Extension Descriptor“ wenn das `extension-desc` Element vorhanden ist.
Dieses Element enthält Informationen über einen potentiellen Installer für plattformabhängigen Code und Einstellungen, sowie eine Reihe von JAR Dateien welche die Erweiterung enthalten.

Ein kleines Beispiel:

```
<extension-desc main-class="com.example.InstallerMain">
  <package name="com.example.*" part="default"/>
  <part-def part="default" href="example.jar"/>
</extension-desc>
```

4.2.7.1 Die Attribute und Elemente des <extension-desc> Elements

`main-class`: *(Attribut)* Beschreibt die `main` - Klasse für den Installer/Uninstaller für plattformabhängigen Code. Die JAR Dateien für den Installer sind die, welche schon in der `resources` Sektion der JNLP Datei angegeben wurden. Dies ist identisch mit den Beschreibungen für das `application-desc` Element.

`package`: *(Element)* Beschreibt den Satz an Packages, welche innerhalb der Erweiterung verfügbar sind. Dies ermöglicht es dem JNLP-Client festzustellen, ob eine bestimmte, benötigte Klasse in der Erweiterung enthalten ist, ohne dafür einen Netzwerkzugriff oder gar einen Download machen zu müssen.

`part-def`: *(Element)* Beschreibt einen Satz an JAR-Dateien, welche die API für die Erweiterung enthalten. Ein Package wird durch den Namen des parts indirekt auf eine JAR-Datei oder einen Satz von JAR-Dateien abgebildet.

Auf die Benutzung des Extension - Descriptors werde ich aufgrund der hohen Komplexität des Bereichs später noch einmal detaillierter eingehen.

4.3 Die .jnlp Services

Hier wird etwas genauer auf die JNLP-API eingegangen und vor allem die von dieser API zu Verfügung gestellten Services näher erläutert. Die JNLP-Api stellt den Java Web Start Applikationen zusätzliche Funktionen, sogenannte Services, zur Verfügung. Über diese Services kann die JWS Applikation auf Funktionen zugreifen, die außerhalb Ihrer eigentlichen Sandbox-Funktionalität liegen, wie z.B. Dateioperationen und Clipboard Services.

Das JNLP Package (`javax.jnlp`) wird den JWS Anwendungen automatisch über den JNLP-Client zur Verfügung gestellt, muss also nicht explizit als zusätzliche Ressource übertragen werden.

Um mit den Services zu arbeiten benutzt man zuerst die `lookup()`- Methode der Klasse `ServiceManager`. Diese Klasse kann zum Beispiel auch eine Liste der auf dem Client-System vorhandenen Services ausgeben (`ServiceManager.getService-Names()`), denn es ist nicht unbedingt erforderlich dass jeder JWS-Client alle acht in der JNLP-Spezifikation vorgesehenen Services dem Client zur Verfügung stellt. Es müssen lediglich drei der möglichen acht Services zwingen implementiert sein, und zwar der `BasicService`, der `DownloadService` und der `ExtensionInstallerService`.

Nun möchte ich die einzelnen in der Spezifikation vorgesehenen Services näher erläutern:

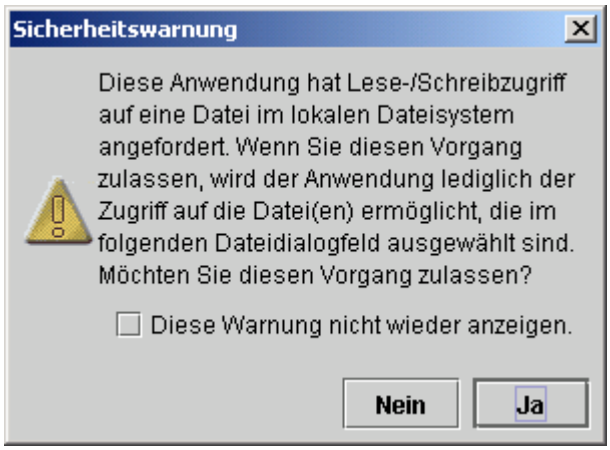
4.3.1 BasicService

Der `BasicService` (`javax.jnlp.BasicService`) wird zwingend von jedem JNLP-Client zur Verfügung gestellt und kann somit von jeder JWS-Applikation genutzt werden. Dieser grundlegende Service beinhaltet Methoden um, ähnlich dem AppletContext bei einem Java-Applet, mit der Umgebung der Applikation, also der Client-Umgebung zu interagieren. So kann eine Applikation zum Beispiel über die Methode `isOffline()`, feststellen ob der Client auf dem sie ausgeführt wird eine aktive Internetverbindung bereithält und kann diese, sofern sie existiert, über die Methode `showDocument()` dazu benutzen, im Standard – Webbrowser des Systems eine URL aufzurufen und darstellen zu lassen. Mit dieser Methode können also sehr schön irgendwelche Links auf die Homepage des Herstellers direkt aus der Applikation heraus realisiert oder auch im Hilfe-Menü auf eine weiterführende Online-Hilfe verwiesen werden.

4.3.2 DownloadService

Mit dem `DownloadService` (`javax.jnlp.DownloadService`) kann eine JNLP-Applikation den Download und das Caching der für die Ausführung benötigten Ressourcen kontrollieren und steuern. So kann die Applikation zum Beispiel kontrollieren, ob eine benötigte Ressource aktuell bereits im Cache vorliegt oder diese bei Bedarf zum Download anfordern. Ebenso kann die Applikation natürlich nicht mehr benötigte Ressourcen wieder aus dem Cache löschen, und zwar mittels der Methode `removeRessource()`. Auch können in der JNLP Beschreibung (den `.jnlp` - Dateien) sogenannte „Lazy Downloads“ definiert werden. Diesen Mechanismus benützt man zum Beispiel für selten gebrauchte Funktionalitäten. Diese „Lazy Downloads“ werden erst dann wirklich vom Server heruntergeladen, wenn der Client die darin enthaltene Funktionalität zum ersten mal benutzt. Mit den Methoden des `DownloadService` können solche noch nicht heruntergeladene Ressourcen gefunden werden und dann natürlich auch bei Bedarf nachinstalliert werden.

4.3.3 FileOpenService und FileSaveService

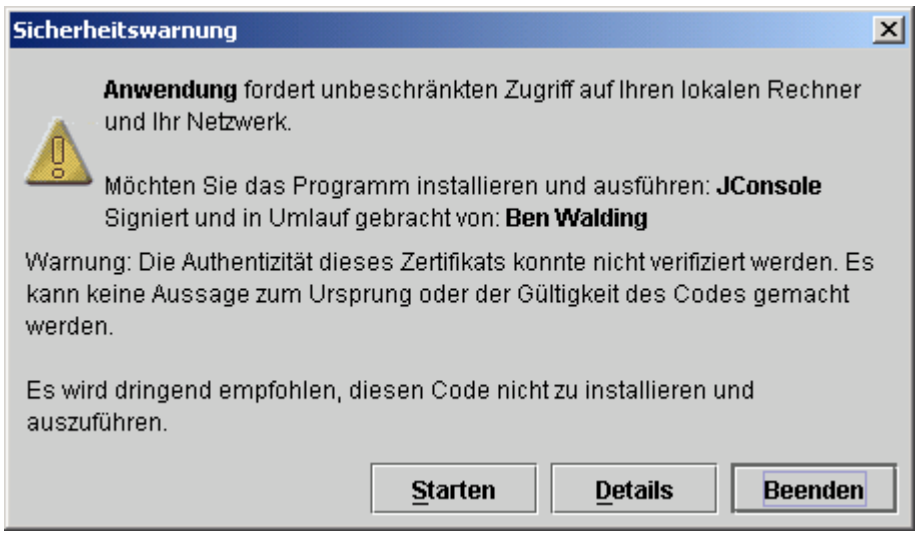


Mit Hilfe dieser zwei File-Services, nämlich `FileOpenService` und `FileSaveService` (`javax.jnlp.FileOpenService` und `javax.jnlp.FileOpenService`) können speziell die nicht-signierten JWS-Applikationen, welche ja standardmäßig innerhalb einer recht restriktiven Sandbox ausgeführt werden, trotzdem Zugriff auf das lokale Dateisystem des ausführenden Clients erhalten. Über diese Methode können so einzelne Dateisystemzugriffe erfolgen. Der User bekommt eine Sicherheitswarnung angezeigt, die es Ihm ermöglicht der Applikation die angeforderten Rechte einzuräumen oder zu verwehren. (siehe Abbildung ....)

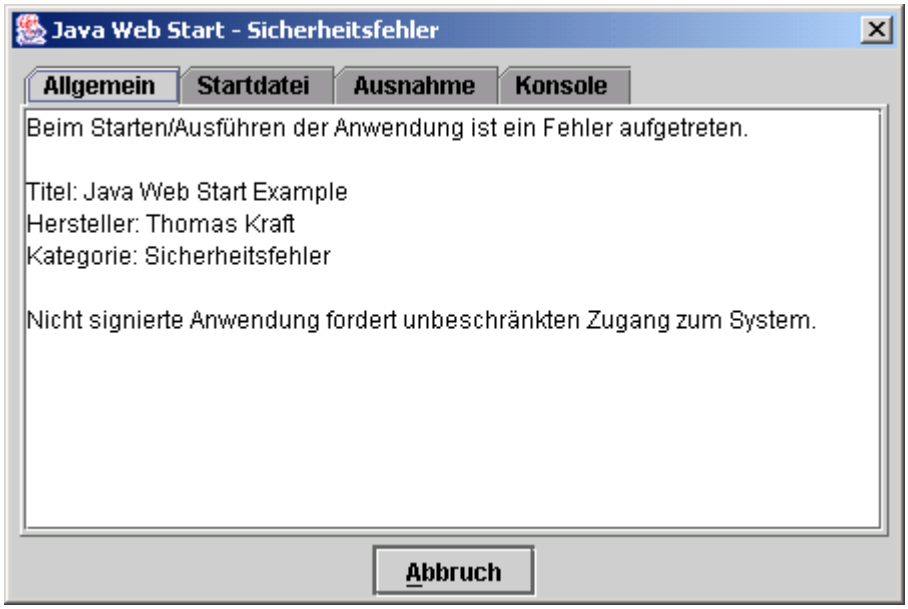
Die Methoden `openFileDialog()` und `saveFileDialog()`, welche die oben gezeigte Sicherheitswarnung bei unsignierten bzw. nicht `<all-permissions/>` benutzenden Applikationen anzeigen, geben bei Zustimmung des Anwenders, und nur dann, ein `FileContens` - Objekt zurück über welches dann z.B. ein `InputStream` respektive ein `OutputStream` auf die referenzierte Datei erzeugt werden kann.

Eine Applikation kann darüber hinaus nicht nur bei jeder einzelnen Lese- oder Schreiboperation die Erlaubnis des Anwenders einholen sondern auch gleich beim Start der Applikation vollen Zugriff anfordern.
Um den vollen Zugriff auf das Clientseitige Dateisystem bereits vom Start der Applikation an zu haben, muss in der JNLP-Datei im Abschnitt `<security>` der Tag `<all-permissions/>` eingefügt werden. Mit diesem Tag fordert die Applikation bereits beim Start alle Rechte an und der Anwender muss somit diese Rechte nur einmal gewähren und nicht bei jedem Benutzen der entsprechenden Services. Um das `<all-permissions/>` Tag benutzen zu können müssen zusätzlich die JAR Dateien der Applikation mit Hilfe des Jarsigner-Tools

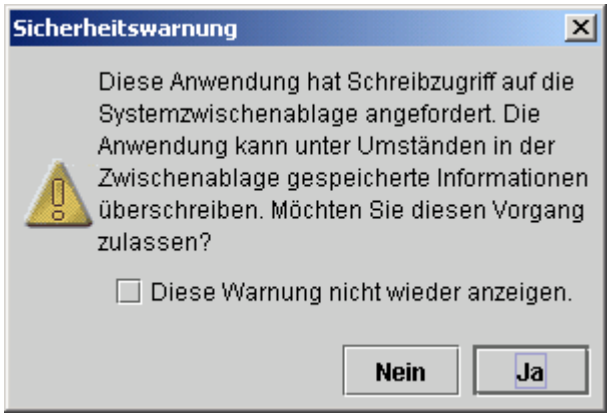
signiert werden. Bei signierten Applikationen muss der Anwender zuerst sicherstellen, dass der Autor der Software der ist, der er vorgibt zu sein, und weiterhin muss der Anwender dem Autor und dessen Code soweit vertrauen, dass er Ihm von vornherein vollen Zugriff auf alle System-Ressourcen die innerhalb der virtuellen Maschine zur Verfügung stehen gewährt. JWS gibt dazu eine Sicherheitswarnung aus (siehe Abbildung .....), wo der Anwender dem Code vertrauen muss. Allerdings rät der Dialog von JWS bei nicht von speziellen Zertifizierungsstellen wie Verisign zertifizierten Signaturen den Code nicht zu installieren, was die potentielle Vertrauensbasis zwischen Anwender und Entwickler nicht gerade fördert, und ein Zertifikat von Verisign oder ähnlichen Trust-Centern ist natürlich auch nicht billig.



Bei Benutzung des `<all-permissions/>` Tags mit unsignierten JARs kommt eine Fehlermeldung (Sicherheitsfenster) und der JWS-Client verweigert die Ausführung dieses potentiell gefährlichen Codes. (siehe Abbildung ....)

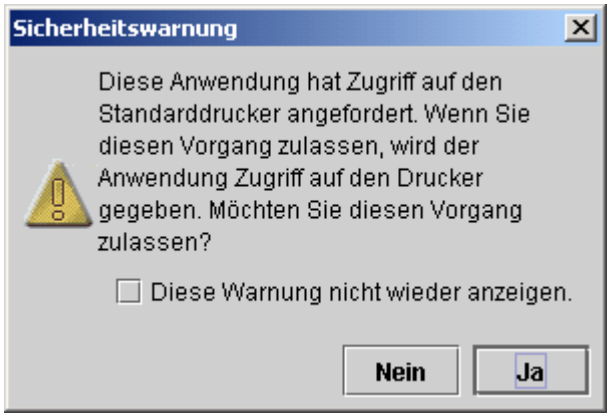


4.3.4 Clipboard Service



Über diesen Service (`javax.jnlp.ClipBoardService`) ermöglicht JWS der Applikation den Zugriff auf die Zwischenablage des Systems. Bevor jedoch die zwei dazu bereitgestellten Methoden `setContents()` und `getContens()` benutzt werden können wird wieder vom JNLP Client ein Sicherheitshinweis angezeigt, der den Anwender über mögliche Sicherheitsrisiken informiert und Ihn dazu auffordert der Anwendung explizit den Zugriff auf die Zwischenablage zu erlauben. (Siehe Abbildung .....)

4.3.5 PrintService



Über diesen Service (`javax.jnlp.PrintService`) ist es möglich, Druckaufträge aus der JWS-Applikation heraus an das ausführende System weiterzugeben. Wiederum muss eine Sicherheitswarnung vom Anwender bestätigt werden (siehe Abbildung .....) , danach wird der Druckauftrag (ein beliebiges `Pageable` - bzw. `Printable` - Objekt) in die Durckerwarteschlange des Systems eingereiht.

4.3.6 PersistenceService

Mit dem `PersistenceService` (`javax.jnlp.PersistenceService`) können kleine, persistente Datenpäckchen auf dem Clientrechner gespeichert werden, ähnlich wie man das von Browsern her kennt. Ähnlich wie bei den dort verwendeten Cookies kann auf diese Datenpäckchen nur diejenige Applikation zugreifen, die diese Daten auch abgelegt hat. Der Zugriff auf die innerhalb dieser Datenpäckchen abgelegten Werte („`values`“) wird über Schlüssel (sog. „`keys`“) geregelt. Als „`key`“ dienen dabei die URLs, die anhand der Codebase einer Applikation erzeugt werden. (z.B. `www.server.de/jwsapps/app1`).
Um anderen Applikationen diese Daten trotzdem zur Verfügung zu stellen, müssen diese vom gleichen Host zur Verfügung gestellt werden, und die Daten müssen weiterhin mittels eines Schlüssels abgelegt werden, der überhalb der Verzeichnisse beider (oder mehrerer) Applikationen liegt.

Beispielsweise könnte die Applikation 1 Ihre Daten nicht wie oben angegeben im Verzeichnis `www.server.de/jwsapps/app1/` sondern im Verzeichnis `www.server.de/jwsapps/` speichern, so hätte auch eine Applikation 2 unter `www.server.de/jwsapps/app2` zugriff auf diese von der Applikation 1 abgelegten Daten.

Für die Verwaltung und Manipulation dieser Daten („`values`“) stellt dieser Service die Methoden `get()`, `create()` sowie `delete()` zur Verfügung. Die JNLP-Spezifikation sieht für diesen Service mindestens 128KB für alle abgespeicherten Daten einer Applikation vor, das heisst jede Applikation kann, die Verfügbarkeit des `PersistenceService` vorausgesetzt, mind. 128KB an Daten innerhalb dieses Service auf dem Clientrechner persistent speichern.

4.3.7 ExtensionInstallerService

Mit dem `ExtensionInstallerService` (`javax.jnlp.ExtensionInstallerService`) ist es möglich, plattformabhängigen Code zu installieren, sowie Einstellungen für diesen anzugeben. So ist es möglich, zum Beispiel laufzeitkritische Routinen aus Java auszulagern um mehr Effizienz bei der Programmausführung zu erreichen, wobei man jedoch das „Write Once – Run Everywhere“ - Paradigma untergräbt. So ist eine JWS-Applikation mit plattformabhängigem Code logischerweise nicht mehr überall lauffähig, was aber je nach Art der Anwendung nicht unbedingt von Nachteil sein muss.

Über diesen `ExtensionInstallerService` ist es zum Beispiel möglich, eine Fortschrittsanzeige bei der Installation zu verändern, um den Anwender über den Status der Installation zu informieren. Auch können mit diesem Service dem JNLP-Client die Pfade zu der installierten Erweiterung innerhalb des lokalen Dateisystems angegeben werden, da diese Erweiterungen nicht zwingenderweise innerhalb des Verzeichnisses des JNLP Clients liegen müssen.

5. Eine kleine Beispielapplikation in Java Web Start

In diesem Kapitel wird exemplarisch die Entwicklung eines Java Web Start Programms erläutert. Zuerst wird eine kleine Stand-Alone Applikation entwickelt, und diese dann per JNLP WebStart-fähig gemacht. Sie werden sehen, dass es sehr einfach ist Applikationen über das Web mit Hilfe von JNLP zur Verfügung zu stellen. Diese kleine Applikation wird dann Schritt für Schritt um die JNLP-Spezifische Services erweitert, um zu zeigen, wie einfach auch diese speziellen Sevices verwendet werden können um am Schluss eine vollständige, WebStart – fähige Applikation zur Verfügung zu haben.

Als erstes muss dafür gesorgt werden, dass Java Web Start auf dem Rechner überhaupt verfügbar ist. Falls sie bereits das JDK in der Version 1.4 oder höher installiert haben, befindet sich Java Web Start schon auf Ihrem Rechner und sie müssen nichts mehr tun. Falls sie bereits eine Java Laufzeitumgebung (JRE) installiert haben, jedoch ohne Web Start Client genügt ein Update, welches mit gerade einmal 720KB recht überschaubar gehalten ist. Sollten sie noch gar kein Java installiert haben, dann müssen Sie das komplette JRE-Paket herunterladen, welches in der englischen Version aktuell ca. 5,5 MB Groß ist.
In beiden Fällen können Sie sich die benötigte Software kostenlos direkt von Sun herunterladen, und zwar unter `http://java.sun.com/products/javawebstart/`

5.1 Mini-Applikation WebStart-fähig machen

5.1.1 Erstellen der Applikation

Zuerst wird eine kleine Applikation erstellt, die dann mittels JNLP WebStart-fähig gemacht wird. Um die ganze Sache einfach zu halten besteht die Applikation nur aus einem sehr simplen Swing-Frame. Ohne Menüs oder Buttons, nur ein Window-Listener ist implementiert, damit man das Fenster wieder schließen kann. Auch sind in dieser „frühen“ Stufe der Applikation noch keine JNLP-spezifischen Services verwendet. Diese Beispielapplikation soll nur dazu dienen den Mechanismus der Veröffentlichung und Bereitstellung mittels JNLP dienen.

Zuerst einmal muss eine Java-Datei erstellt werden, die den Quellcode der Applikation enthält.

Hier der Quellcode der Datei `JWS_example.java` :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class JWS_Example extends Jframe {
    JLabel javaWebStartLabel;

    public JWS_Example() {
        //WindowListener einfügen, um Fenster zu schliessen
        this.addWindowListener(new MyWindowAdapter());
        //Größe und Titel angeben
        this.setSize(300,300);
        this.setTitle("Java Web Start Example");
        this.show();
    }

    public static void main(String args[]) {
        new JWS_Example();
    }

    class MyWindowAdapter extends WindowAdapter {
        public void windowClosing (WindowEvent we) {
            System.exit(0);
        }
    }
}
```

Nach dem erstellen der Datei `JWS_Example.java` kompilieren sie das Programm wie gewohnt mit `javac` und erhalten so die Dateien

`JWS_Example.class`

und

`JWS_Example$MyWindowAdapter.class`.

Mit diesen zwei Dateien können sie diese Applikation bereits ganz normal starten. Sie sehen ein Fenster, welches ungefähr so aussehen sollte: (Siehe Abbildung)

5.1.2 Erstellen des benötigen JAR-Archivs

Um diese (zugegeben sehr rudimentäre) Applikation jetzt WebStart-fähig zu machen ist es nötig, dass die erzeugten Klassen in eine JAR-Datei gepackt werden, da es nicht möglich ist aus einer JNLP-Datei heraus `.class` - Dateien einzeln zu verlinken, es können immer nur `.jar` - Dateien angegeben werden.

Zum erzeugen der `.jar`-Datei benutzen sie am besten das von Sun mitgelieferte JAR-Tool `jar`. Wechseln sie in das Verzeichnis Ihrer Applikation und geben sie folgenden Befehl ein um die JAR-Datei zu erstellen:

```
jar cvf JWS_Example.jar *.*
```

Jetzt gibt es innerhalb des aktuellen Arbeitsverzeichnisses eine Datei namens `JWS_Example.jar`, welche die Applikation enthält und mittels welcher sie diese Applikation auch übers Netz deployen können.

5.1.2 Erstellen der JNLP Beschreibungsdatei für die Applikation

Nachdem sie nun erfolgreich das JAR-Archiv für die Applikation gebastelt haben wird es Zeit die für Java Web Start nötige JNLP Beschreibungsdatei zu erstellen, welche grundlegende Details und Informationen über die Applikation enthält und mit deren Hilfe der JWS-Client die Applikation und die dazugehörenden Ressourcen aus dem Netz lädt und installiert.

Wenn Sie dieses Beispiel auf Ihrem eigenen WebServer ausprobieren wollen vergessen Sie bitte nicht die entsprechenden Attribute des `<jnlp>` Tags auf Ihre Bedürfnisse anzupassen.

Hier der Quellcode der Datei `JWS_example.jnlp` :

```
<jnlp spec="1.0+" version="1.0" href="JWS_Example.jnlp"
           codebase="http://www.thomaskraft.de/jws-demo/V1/">
  <information>
    <title>Java Web Start Example</title>
    <vendor>Thomas Kraft</vendor>
    <homepage href="http://www.thomaskraft.de"/>
    <description>Dies ist eine simple Beispiel-Applikation für
                 Java Web Start</description>
    <description kind="one-line">Beispiel-Applikation für Java
                                 Web Start</description>
    <description kind="tooltip">JWS-Demo</description>
    <description kind="short">Java Web Start
                              Beispiel</description>
    <offline-allowed/>
  </information>
  <security/>
  <resources>
    <j2se version="1.4 1.3 1.2"/>
    <jar href="JWS_Example.jar" main="true"/>
  </resources>
  <application-desc main-class="JWS_Example"/>
</jnlp>
```

In dieser Konfigurationsdatei sind diverse Information für den Java Web Start Client vorhanden. Innerhalb der `<information>` Tags stehen Informationen, die für den eigentlichen Betrieb der Applikation zwar nicht unbedingt von Nöten sind, aber trotzdem verwendet werden, zum Beispiel als Anzeige während des Downloads, in der Ansicht des Application Managers und wenn später eine Verknüpfung auf dem Desktop abgelegt werden soll.

Durch den Tag `<offline-allowed/>` wird es explizit erlaubt, dass diese Applikation auch ohne Internetverbindung gestartet werden kann. Der Anwender kann dies später entweder durch eine eventuell angelegte Verknüpfung auf dem Desktop oder aber direkt über den Application Manager tun.

Da in diesem trivialen Beispiel noch keinerlei JNLP Services benutzet werden, wurde hier auch das `<security/>` Tag leer gelassen, da ja keine sicherheitskritischen Funktionen aufrufen werden. Würde bei dieser Applikation jedoch innerhalb des `<security/>` Tags ein Tag `<all-permissions/>` gesetzt werden, so würde Java Web Start die Ausführung dieser Applikation verweigern, da das `<all-permissions/>` Tag nur von signierten Applikationen benutzt werden darf und diese kleine Beispiel-Applikation ja noch nicht signiert ist.

Innerhalb des `<ressources/>` Tags wurden in diesem Beispiel die JRE-Versionen 1.4, 1.3 oder 1.2 vorausgesetzt, da ja ein Swing-Frame benutzt wird, welches erst ab Version 1.2 der JRE verfügbar ist. Hier müssen auch die ganzen JAR Dateien angegeben werden, welche die Applikation benutzt. Ebenso muss angeben werden, welche der angeforderten JAR Dateien die `main` - Klasse der Applikation enthält, damit der JWS Client diese auch ordnungsgemäss starten kann.

5.1.3 Hochladen auf den Server

Nachdem sie auf Ihrem Server ein entsprechendes Verzeichnis angelegt haben (in diesem Beispiel das Verzeichnis `/jws-demo/V1/` müssen sie nur noch die zwei Dateien `JWS_Example.jar` und `JWS_Example.jnlp` in dieses Verzeichnis hochladen. Beachten sie, dass das Verzeichnis das im Codebase-Attribut der JNLP Datei angegebene sein muss. In diesem Falle werden alle beiden Dateien in das Verzeichnis `/jws-demo/V1/` hochgeladen.

Nach dem Hochladen der Dateien sollte die Applikation bereits über Web Start lauffähig sein. Geben sie dazu einfach die absolute URL der `jnlp` - Datei in die Adresszeile Ihres Browsers ein, dieser sollte dann selbstständig diese Datei an den JWS-Client übergeben und dieser wird nach dem Start die benötigte JAR Datei herunterladen und danach die Beispielapplikation starten.

Sollte wider Erwarten der JWS Client nicht gestartet werden, sondern der Browser den Inhalt der JNLP Datei anzeigen, anstatt ihn ordnungsgemäß weiterzureichen müssen auf ihrem Server noch die MIME-Typen angepasst werden, so dass der Server bei Anforderung einer JNLP Datei diesen an den Browser übermittelt.

Wenn sie Beispielsweise den WebServer Apache verwenden, so müssen sie in dessen Konfigurations-Datei `httpd.conf` folgendes eintragen:

```
AddType application/x-java-jnlp-file.jnlp
```

Falls es jetzt immer noch nicht funktioniert, müssen sie wahrscheinlich Ihren Webserver neu starten, da es durchaus möglich ist, dass dieser die neue, geänderte Konfigurationsdatei noch nicht geladen hat und deshalb den entsprechenden MIME Type immer noch nicht kennt.

Wenn nach Eingabe der URL der JWS Client ordnungsgemäss startet, es jedoch bei der Ausführung der Applikation zu Fehlern kommt, bietet ihnen der Application Manager wertvolle Hilfe, da er Details zu den aufgetretenen Fehlern anzeigt und sie so normalerweise recht schnell erkennen, wo der Fehler liegt. Meist sind es kleinere Fehler und Unsauberheiten innerhalb der JNLP Beschreibungsdatei.

Da es natürlich unschön ist, die Applikation über eine direkte URL Eingabe innerhalb des Browsers zu starten, wäre der nächste Schritt, eine HTML Datei zu erstellen, welche einen Link auf die entsprechende JNLP Datei enthält, so dass der Anwender diese wirklich „With a single click“ starten kann. Dieser Link könnte zum Beispiel so aussehen:

```
<a href="JWS_Example.jnlp">Java Web Start Beispiel</a>
```

Glückwunsch! Sie haben soeben eine Applikation mit Hilfe der neuen, zukunftsweisenden Technologie von Sun, Java Web Start deployed!

Dieses Beispiel können sie unter `http://www.thomaskraft.de/jws-demo/` testen.

5.2 Erweiterung der Applikation um JNLP Services

Im ersten Teil dieses Kapitels wurde eine kleine Applikation erstellt und diese mittels JNLP im Netz verfügbar gemacht. Allerdings hat diese Applikation noch keinerlei Funktionen der JNLP API benutzt, die Java Web Start zu dem machen was es ist. Im folgenden wird nun die Applikation Schritt für Schritt erweitert werden, um die verfügbaren Services von JNLP zu nutzen.

Dazu wird zuerst einmal ein kleines Menü in unsere Applikation eingebaut, aus dem heraus dann die einzelnen Services angesprochen werden können. Im ersten Menüpunkt kommt lediglich ein Eintrag zum Programm beenden hinein, damit die Applikation ordnungsgemäß über einen Menüpunkt beendet werden kann. Im zweiten Menüpunkt werden nach und nach die einzelnen JNLP Services eingebaut.

Als erstes wird der `BasicService` benutzt, um eine Webseite in einem Browserfenster anzeigen zu lassen, wenn eine Internetverbindung besteht.

Der Übersichtlichkeit halber werden hier nur die wichtigsten Änderungen im Quellcode angeben. Die vollständigen Quellen der jeweiligen Versionen können sie sich unter `http://www.thomaskraft.de/jws-demo/` herunterladen.

5.2.1 BasicService

Hier wird die Applikation vom JWS-Client einen `BasicService` anfordern und sofern dieser Verfügbar ist mit Hilfe der Methode `showDocument()` eine URL in einem neuen Browserfenster anzeigen lassen.

Dazu müssen zuerst noch ein paar Libraries eingebunden werden:

```
import javax.jnlp.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.net.URL;
import java.net.MalformedURLException;
```

Und noch ein paar Variablen:

```
JMenuBar myMenuBar;
JMenu dateiMenu;
JMenu JNLPServicesMenu;
JMenuItem dateiBeenden;
JMenuItem testBasicService;
```

Dann muss innerhalb des Konstruktors von `JWS_Example.java` noch ein kleines `JMenu` gebaut werden:

```
myMenuBar = new JMenuBar();

dateiMenu = new JMenu("Programm");
dateiMenu.setMnemonic('P');
dateiBeenden = new JMenuItem("Beenden");
dateiBeenden.setMnemonic('B');
dateiBeenden.addActionListener(this);
dateiMenu.add(dateiBeenden);

JNLPServicesMenu = new JMenu("JNLP-Services");
JNLPServicesMenu.setMnemonic('J');

testBasicService = new JMenuItem("BasicService");
testBasicService.setMnemonic('B');
testBasicService.addActionListener(this);
JNLPServicesMenu.add(testBasicService);

myMenuBar.add(dateiMenu);
myMenuBar.add(JNLPServicesMenu);

this.setJMenuBar(myMenuBar);
```

Weiterhin kommen noch zwei neue Methoden hinzu:

```
public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == dateiBeenden) System.exit(0);
    if (ae.getSource() == testBasicService)
                          testBasicServiceAction(ae);
}

private void testBasicServiceAction(ActionEvent e) {
    String allServices[] = ServiceManager.getServiceNames();
    for (int i = 0; i < allServices.length; i++)
                        System.out.println("Verfügbare Services: " +
                                           allServices[i]);
    try {
        BasicService bs = (BasicService)ServiceManager.
                                    lookup("javax.jnlp.BasicService");
        if (!bs.isOffline() && bs.isWebBrowserSupported()) {
            System.out.println("Internetverbindung verfügbar, " +
                               "öffne Browserfenster...");
            bs.showDocument(new URL("http://www.thomaskraft.de"));
        } else {
            System.out.println("Internetverbindung nicht verfügbar, " +
                               "oder es wird kein Browser unterstützt. " +
                               "Werde daher kein Browserfenster öffnen");
        }
    } catch (UnavailableServiceException se) {
        System.out.println("BasicService nicht verfügbar." +
                           se.toString());
    } catch (MalformedURLException ue) {
        System.out.println("Fehler: URL malformed."+ ue.toString());
    }
}
```

Die eingebundenen Bibliotheken sind nötig um die erweiterte Funktionalität abzudecken, die in diesem zweiten Beispiel benutzt werden soll.
Das eingefügte `JMenu` besitzt zwei Unterpunkte, zum einen „`Programm`“, in welchem sich nur ein Eintrag zum Programm beenden versteckt. Das andere Menü ist „`JNLP-Services`“, in welches nach und nach die ganzen Services eingebaut werden. Im Moment befindet sich nur ein Eintrag darin, nämlich „`Basic Service`“ welcher über die Methode `actionPerformed()` die Methode `testBasicServiceAction()` aufruft.

Diese Methode fordert zunächst vom Servicemanager eine Liste aller verfügbaren Services an und gibt diese auf der Standardausgabe aus. Je nach verwendetem JWS-Client ist diese Liste mehr oder weniger vollständig. Bei dem von Sun zur Verfügung gestellten Java Web Start Client (die Referenzimplementierung) sind alle Services implementiert. Um die Ausgaben dieser Liste zu sehen müssen sie die Java-Konsole im Java Web Start Client aktivieren.

Danach fordert die Applikation vom `ServiceManager` einen `Basicservice` an, mit dessen Hilfe zuerst einmal abgeprüft wird, ob eine aktive Internetverbindung vorhanden ist und ob das System einen Webbrowser unterstützt. Wenn beide Bedingungen erfüllt sind, wird mit der vom `BasicService` zur Verfügung gestellten Methode `showDocument()` in einem neuen Browserfenster eine URL angezeigt.

5.2.2 FileOpenService

In dieser dritten Version der kleinen Applikation wird der `FileOpenService` verwendet, um eine Datei vom lokalen Dateisystem des Clientrechners zu öffnen und es wird versucht, sie auf der Standardausgabe von Java auszugeben.

Zuerst müssen noch weitere API-Funktionen eingebunden werden:

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
```

Sowie einen weiteren Menüeintrag:

```
JMenuItem testFileOpenService;
```

Innerhalb des Kontruktors von `JWS_Example` wird das Menü erweitert:

```
testFileOpenService = new JMenuItem("FileOpenService");
testFileOpenService.setMnemonic('O');
testFileOpenService.addActionListener(this);
JNLPServicesMenu.add(testFileOpenService);
```

Auch der ActionEvent-Handler `actionPerformed()` bekommt einen neuen Eintrag:

```
if (ae.getSource() == fileOpenServiceTest) test.FileOpenServiceAction(ae);
```

Zuguterletzt kommt noch die Methode hinzu, mit welcher der `FileOpenService` aufgerufen wird:

```
private void testFileOpenServiceAction(ActionEvent e) {
    System.out.println("testFileOpenServiceAction:");
    try {
        FileOpenService fos = (FileOpenService)ServiceManager.
                              lookup("javax.jnlp.FileOpenService");
        String extensions[] = {"txt", "java"};
        FileContents fc = fos.openFileDialog("/", extensions);
        if (fc != null) {
            System.out.println("Inhalt der Datei " + fc.getName() + ":");
            InputStreamReader isr = new InputStreamReader(fc.
                                                    getInputStream());
            BufferedReader br = new BufferedReader(isr);
            String line = new String();
            while ( (line = br.readLine()) != null) {
                 System.out.println(line);
            }
            System.out.println("EOF");
        } else {
            System.out.println("Anwender hat FileOpenDialog abgebrochen.");
        }
    } catch (UnavailableServiceException se) {
            System.out.println("FileOpenService nicht verfügbar! " +
                               se.toString());
    } catch (IOException ie) {
        System.out.println("IOException in FileOpenDialog " +
                           ie.toString());
    }
}
```

In dieser Applikation muss auf die von JNLP bereitgestellten Services zurückgegriffen werden, um auf das lokale Dateisystem des Clients zugreifen zu können, da die Applikation noch nicht signiert ist. Hier wird nur lesend zugegriffen, und zwar mit Hilfe des Services `FileOpenService`. Über die Methode `openFileDialog()` wird ein Datei-Dialog Fenster geöffnet, dabei wird dem User zuerst eine Sicherheitsmeldung angezeigt, und er muss sich explizit damit einverstanden erklären. Die Methode erwartet zwei Parameter, zum ersten das Ausgangsverzeichnis für den `FileDialog` (Also das Verzeichnis das, wenn verfügbar, beim öffnen des Datei-Dialogs angezeigt wird), zum zweiten den Filter für Dateiendungen (Hier `.txt` und `.java`)
Bei Zustimmung des Users wird das Dialogfenster geöffnet, wo der User eine Datei auswählen kann. Wenn der User eine Datei ausgewählt hat, wird diese als Objekt vom Typ `FileContents` zurückgeliefert, über welches man dann auch einen `FileInputStream` erhalten kann, welcher dann wie gewohnt weiterverarbeitet werden kann. Die einzige Einschränkung, die `FileContents` beinhaltet ist, dass es darüber nicht möglich ist die absolute URL der ausgewählten Datei zu ermitteln.

5.2.3 FileSaveService

In der vierten Version dieser kleinen Applikation wird noch einen FileSave-Dialog hinzugefügt, welcher über den gleichnamigen JNLP-Service `FileSaveService` realisiert wird.

Ein weiterer Menüeintrag:

```
JMenuItem testFileSaveService;
```

Die Erweiterung des Menüs innerhalb des Kontruktors von `JWS_Example`:

```
testFileSaveService = new JMenuItem("FileSaveService");
testFileSaveService.setMnemonic('S');
testFileSaveService.addActionListener(this);
JNLPServicesMenu.add(testFileSaveService);
```

ActionHandler Erweiterung:

```
if (ae.getSource() == fileSaveServiceTest) fileSaveServiceTestAction(ae);
```

Und die zusätzliche Methode:

```
private void testFileSaveServiceAction(ActionEvent e) {
    System.out.println("testFileOpenServicetAction:");
    try {
        FileSaveService fss = FileSaveService)ServiceManager.
                              lookup("javax.jnlp.FileSaveService");
        URL testURL = new URL("http://www.thomaskraft.de/index.htm");
        InputStream is = testURL.openStream();
        String extensions[] = {"txt", "java"};
        FileContents fc = fss.saveFileDialog("/", extensions,
                                             is, "test.txt");
        if (fc == null) {
            System.out.println("Anwender hat Dialog abgebrochen");
        } else {
            System.out.println(fc.getName() + " gespeichert.");
        }
    } catch(UnavailableServiceException se) {
        System.out.println("FileSaveService nicht verfügbar! " +
                           se.toString());
    } catch(MalformedURLException ue) {
        System.out.println("Fehler: URL malformed. " + ue.toString());
    } catch(IOException ie) {
        System.out.println("Fehler: IOException! " + ie.toString());
    }
}
```

Der `FileSaveService` arbeitet ziemlich ähnlich dem `FileOpenService`, auch hier wird wieder ein `FileContents` Objekt erzeugt, mit dem wir die Methode `saveFileDialog()` aufrufen. Der Anwender bekommt wieder im Sicherheitsfenster angezeigt, dass die Applikation auf sein lokales Dateisystem zugreifen will.
Im Gegensatz zu `openFileDialog` besitzt `saveFileDialog` allerdings vier Parameter. Die ersten zwei Parameter sind identisch, also das Ausgangsverzeichnis für den FileDialog , zum zweiten den Filter für Dateiendungen. Zusätzlich verlangt `saveFileDialog` noch einen `InputStream`, aus welchem die Datei, die später geschrieben wird, geholt wird, und einen Vorschlag für einen Dateinamen.
In unserem Falle laden wir einfach die `index.htm` von `thomaskraft.de` und übergeben den Inhalt dieser Datei mit dem `InputStream` und speichern diesen dann lokal ab.
Sollte der Anwender den Sicherheitshinweis zurückweisen, so erhalten wir in dem Objekt `FileContents` eine leere Referenz (`null`) zurück und können entsprechend darauf reagieren (siehe Listing).

5.2.4 ClipboardService

Zuguterletzt wird die kleine Demo-Applikation noch mit Clipboard-Funktionalität ausgestattet, um auch diesen Service ein wenig zu erläutern.

Zuerst jedoch wieder einmal die Erweiterungen, die am Source-Code vorgenommen werden müssen:

Noch fehlende imports:

```
import java.awt.datatransfer.StringSelection;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.UnsupportedFlavorException;
```

Die Erweiterung für das Menü:

```
testClipboardService = new JMenuItem("ClipboardService");
testClipboardService.setMnemonic('C');
testClipboardService.addActionListener(this);
JNLPServicesMenu.add(testClipboardService);
```

Der zusätzliche Eintrag für den ActionEvent Handler:

```
if (ae.getSource() == testClipboardService) testClipboardServiceAction(ae);
```

Und last but not least die Methode die den Clipboard-Zugriff macht:

```
private void testClipboardServiceAction(ActionEvent e) {
    try {
        ClipboardService cs = (ClipboardService)ServiceManager.
                              lookup("javax.jnlp.ClipboardService");
        if (cs != null) {
            // String im Clipboard ablegen
            StringSelection ss = new StringSelection("Java Web " +
                                                     "Start rox0rt");
            cs.setContents(ss);
            // Clipboard auslesen und auf stdout ausgeben
            Transferable tr = cs.getContents();
            if (tr.isDataFlavorSupported(DataFlavor.stringFlavor)) {
                String s = (String)tr.
                           getTransferData(DataFlavor.stringFlavor);
                System.out.println("Clipboard Inhalt: " + s);
            }
        }
    } catch (UnavailableServiceException se) {
        System.out.println("ClipboardService nicht verfügbar!");
    } catch (UnsupportedFlavorException ue) {
        System.out.println("StringFlavor wird nicht unterstützt!");
    } catch (IOException ie) {
        System.out.println("Fehler: IOException! " + ie.toString());
    }
}
```

Zuerst wird beim Servicemanager der entsprechenden Service `javax.jnlp.ClipboardService` angefordert, um die Zwischenablage überhaupt nutzen zu können. Wenn der Wert des Clipboard-Objektes ungleich `null` ist und dabei keine Ausnahme aufgetreten ist, so kann die Zwischenablage des Systems benutzt werden.
Als kleines Beispiel für die Benutzung der Zwischenablage wir hier zuerst der String "Java Web Start rox0rt" in die Zwischenablage geschrieben. Dabei kommt schon ein Sicherheitsfenster, welches den User fragt ob er den schreibenden Zugriff auf die Zwischenablage erlauben will. Falls der User das gestattet wird der String in die Zwischenablage geschrieben.
Als zweiten Schritt muss natürlich der soeben geschriebenen String auch wieder aus der Zwischenablage ausgelesen werden. Auch diesen lesenden Zugriff muss der User wieder bestätigen. Nach erfolgreichem Lesen wird der Inhalt der Zwischenablage auf `stdout` geschrieben. Sie müssen die Konsole Ihres Java Web Start Clients aktiviert haben, damit sie die Ausgaben sehen können.

Die beiden zur Verfügung stehenden Methoden des `ClipboardService` lauten `setContents()` und `getContents()`. Zum schreibenden Zugriff auf die Zwischenablage wird der Methode `setContens()` ein Transferable-Objekt übergeben. Der Einfachheit halber wird hier ein `StringSelection` benutzt, da diese Klasse die einfachste Implementierung des `Transferable` - Interfaces ist. Denkbar wäre jedoch auch jedes Beliebige andere `Transferable` - Objekt. Das Lesen aus der Zwischenablage über die Methode `getContens()` gibt natürlich auch wieder ein `Transferable` Objekt zurück. Da nicht unbedingt davon ausgegangen werden kann, dass der geschriebene String auch beim Lesen der Zwischenablage noch vorhanden ist, wird mit Hilfe der Funktion `isDataFlavorSupported(DataFlavor.stringFlavor)` überprüft, ob das in der Zwischenablage vorhandende Objekt überhaupt in einen `String` verwandelt werden kann.

Wenn dem so ist, wird das `Transferable` Objekt in einen `String` gecasted und auf der Standard Ausgabe ausgegeben.

5.2.5 Signierung der JAR-Datei(en)

In allen bisherigen Versionen des Beispiels war die generierte JAR-Datei nicht signiert und damit in den Berechtigungen sehr stark eingeschränkt. Nur signierte Applikationen dürfen, die Zustimmung des Users natürlich vorausgesetzt, den vollen Zugriff auf das Client-System erlangen.
Bei unsignierten Applikationen muss jeder Zugriff auf das Client-System durch die von der JNLP-API zur Verfügung gestellten Services erfolgen und außerdem vom User einzeln bestätigt werden.

Signierte Applikationen hingegen haben die Möglichkeit, gleich beim Start vom User einen Vollzugriff auf das Client-System zu erfragen und wenn dieser die Applikation Autorisiert, so entfallen die regelmäßigen Sicherheitswarnungen von Java Web Start und die Applikation kann dann auch, anders wie im vorliegenden Beispiel, über die Standard-Java Funktionen auf das Client-System zugreifen, und muss nicht den Weg über die Services gehen.
Natürlich kann auch eine signierte Applikation die JNLP Services verwenden, auch gemischt mit Standard Java Funktionen.

Zum Abschluss noch eine kurze Beschreibung wie sie eine JAR Datei mit Hilfe der von Java mitgelieferten Bordmitteln signieren können, so dass sie gleich beim Start Ihrer Web Start Applikation mit Hilfe des `<all-permissions/>` Tags alle Rechte vom Benutzer anfordern können.

Zuerst brauchen sie einen `key`, ohne den sie überhaupt nichts signieren können. Dazu gibt es aber glücklicherweise auch ein mitgeliefertes Tool von Java, nämlich das `keytool`.

Zuerst generieren sie sich einen entsprechenden Schlüssel:

```
keytool -genkey -keystore myKeystore -alias <ihrName>
```

Mit diesem Befehl erstellen sie im aktuellen Arbeitsverzeichnis einen Schlüssel in der Datei `myKeystore`, sowie ein Test-Zertifikat.

Sie müssen natürlich noch diverse Informationen wie Name, Organisation, Land und so weiter angeben, das sind alles Informationen, die den Schlüssel als ihren eigenen Schlüssel identifizieren.

Nach Erstellen des Schlüssels können sie mit dessen Hilfe ganz einfach die JAR-Datei unserer Applikation signieren indem sie eingeben:

```
jarsigner -keystore myKeystore JWS_Example.jar <ihrName>
```

In der JNLP Beschreibungsdatei zu dieser JAR-Datei können Sie nun innerhalb des `<security/>` Abschnittes das Tag `<all-permissions/>` einfügen, und so schon beim Start der Applikation vom User alle Rechte anfordern.