Mit Jenkins iOS-Projekte testen und bauen

Moritz Haarmann, *Software for mobile devices*

Wer kennt das nicht: Kurz vor Feierabend, man will noch schnell eine neue Version der App bauen und zieht sich dazu die Änderungen des Tages. Command-R, aber Xcode meldet mehrere Build-Fehler. Soviel zum Thema pünktlich Feierabend.

Solche Situationen lassen sich mit einer Continuous Integration-Lösung zwar nicht vermeiden, jedoch ist man besser informiert und hat so die Möglichkeit zeitnah eine Lösung für das Problem zu finden.

Continuous Integration beschreibt die Praxis, bei der nicht nur an einzelnen, definierten Punkten in der Entwicklung, z.B. am Ende eines Entwicklungssprints, Versionen einer Software gebaut und getestet werden, sondern während des gesamten Entwicklungszeitraums – permanent und fortlaufend. Unter Integration versteht man in diesem Kontext die Zusammenführung aller für den Betrieb einer Software benötigten Komponenten. Bei einer App ist dies in der Regel nur der Quellcode, der im Integrationsschritt kompiliert wird.

Als Ergebnis dieses Continuous Integration-Prozesses sind Versionen nach jeder Änderung am Quellcode verfügbar, die Archiviert werden und bei Bedarf auch z.B. an Tester verteilt werden können. Auch aktuelle Testergebnisse werden vorgehalten und geben Informationen über die aktuelle Stabilität des Produkts preis.

Continuous Integration kann dazu beitragen, eine höhere Transparenz zwischen Entwicklungsteams und internen oder externen Kunden zu fördern: Wenn Versionen nicht nur alle paar Wochen oder Monate, sondern im Extremfall mehrfach täglich verteilt werden, hat jeder Stakeholder frühzeitig die Möglichkeit die "Hand zu heben", falls es zu Problemen, Missverständnissen oder abweichenden Vorstellungen kommt.

In diesem Guide installieren wir Jenkins, konfigurieren einen Job der ein iOS-Projekt testet und baut und schieben – im Erfolgsfall – das gebaute IPA-File direkt zu HockeyApp, um es von dort aus an Tester verteilen zu können. Dieser Prozess wird hier in vielen Projekten genau so benutzt und hat sich in der Praxis mehr als bewährt.

1 Jenkins

Jenkins ist ein Continuous Integration Server. Jenkins hat seinen Ursprung in der Java-Welt und basiert selbst auch auf Java, daher ist Jenkins problemlos auch auf OS X lauffähig – eine notwendige Bedingung, da sich iOS-Projekte generell nur unter Mac OS bauen lassen.

1.1 Installation

Die Installation von Jenkins unter OS X ist recht simpel: auf `http://jenkins-ci.org/` findet sich ein Installer für OS X.

> Achtung: Nicht die JAR-Datei laden, sondern den Installer für OS X.

Die Installation legt einen eigenen Benutzer für Jenkins an der, wenig überraschend, Jenkins heisst. Dieser Nutzer hat kein Login-Passwort, man kann über passwd allerdings problemlos eines vergeben. Der neue Nutzer hat sein Homeverzeichnis in der aktuellen Version unter /Users/Shared/Jenkins/Home.

Damit Jenkins beim Starten bzw. Hochfahren des Rechners automatisch gestartet wird, legt der Installer unter /Library/LaunchDaemons/ die Datei org.jenkins−ci.plist an und registriert bei launchctl diesen Service. Möchte man also manuell Jenkins stoppen oder starten kann man dies mit

```
sudo launchctl start org.jenkins−ci
```

bzw.

```
sudo launchctl stop org.jenkins-ci
```

> Sudo ist notwendig, da der Prozess in dem Jenkins läuft, nicht dem Benutzer gehört, mit dem man sich normalerweise anmeldet.

Möchte man Jenkins nicht mehr automatisch nach dem Hochfahren gestartet sehen, kann man über

```
sudo launchctl unload /Library/LaunchDaemons/org.jenkins-ci.plist
```

den Service komplett entfernen. Ersetzt man unload durch load passiert das Gegenteil, Jenkins wird der Serviceliste wieder hinzugefügt.

2 Die Weboberfläche

Nach der Installation öffnet sich automatisch der Standardbrowser mit der URL http://localhost:8080/. Auf diesem Port läuft Jenkins in der Standardeinstellung.

Im noch leeren Bereich befindet sich eine Liste aller aktiven Jobs bzw. in der deutschen Übersetzung Elemente. Im Moment ist dort noch nichts zu sehen, da wir ja noch keinen Job angelegt haben.

Die komplette Interaktion findet über diese Oberflächte statt – Allerdings gibt es auch eine REST-Api, über die 3rd-Party-Software mit Jenkins kommunizieren kann. Das ist Sinnvoll, um von aussen Builds anzustoßen, den Status einzelner oder aller Jobs zu visualisieren oder z.B. über eine App den Buildserver zu konfigurieren.

2.1 Ersteinrichtung

Bevor wir uns an die Einrichtung unseres Jobs machen können ist noch die Installation einiger notwendiger Plugins erforderlich. Jenkins hat, wie schon erwähnt, eine lebendige Community, daher gibt es für die meisten Belange passende Plugins - so auch für das Bauen von iOS-Projekten.

Plugins können über 'Jenkins Verwalten' und den dortigen Menüpunkt Plugins einfach installiert werden. Konkret werden wir folgende Plugins verwenden:

- Git Client Plugin
- Keychains and Provisioning Profiles Plugin
- Xcode Plugin
- HockeyApp Plugin

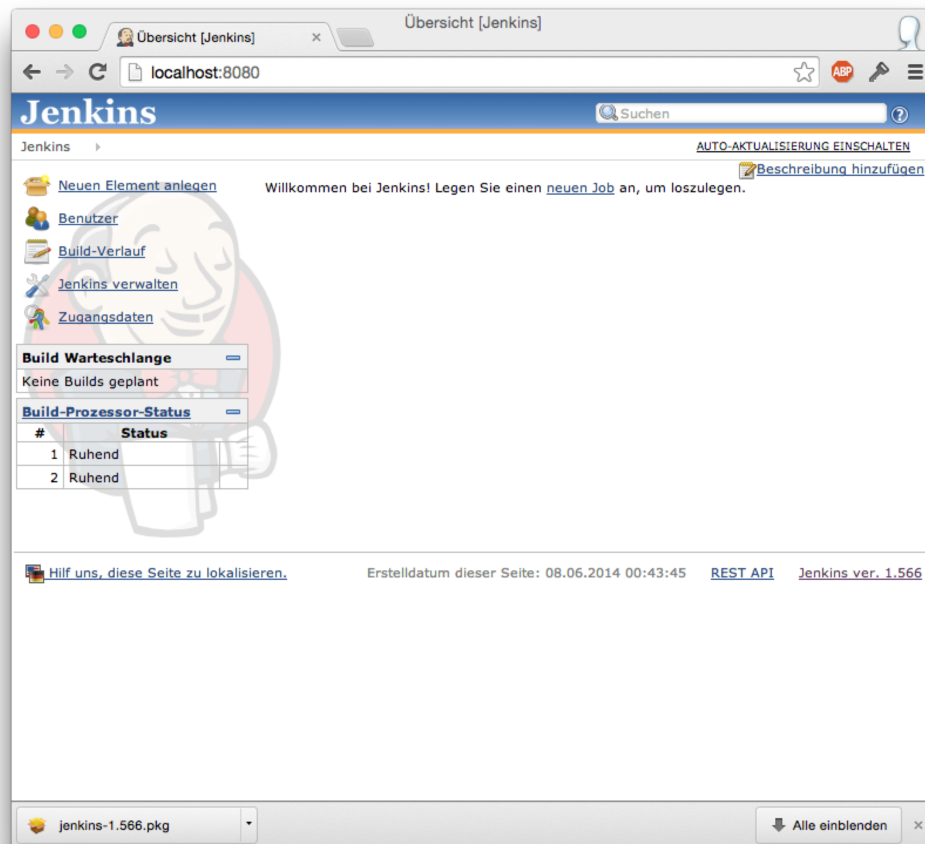

Abbildung 1: *Der frische Jenkins*

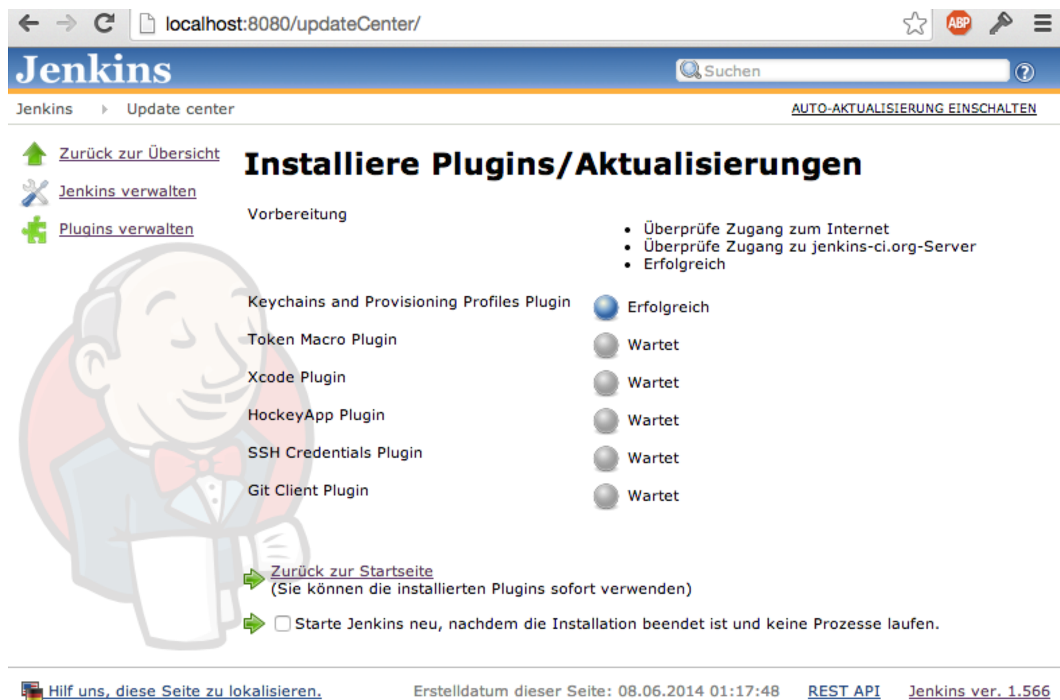

Abbildung 2: *Plugin Installation*

Die Plugins einfach auswählen und auf 'Installieren ohne Neustart' klicken.

Eventuell ist ein Neustart notwendig. Sollte dass der Fall sein, wie im Screenshot zu sehen, die Checkbox unten (Starte Jenkins neu..) auswählen, Jenkins wird dann sofort neu gestartet. Das dauert ein paar Momente, und ein manueller Reload ist eventuell schneller als auf den Timeout der Oberfläche zu warten.

3 Unser iOS-Projekt

Um ein iOS-Projekt zu bauen, brauchen wir erstmal ein iOS-Projekt. Grundsätzlich ist jedes Projekt in Ordnung, solang es in Xcode ordentlich baut. Um alle Schritte hier nachvollziehen zu können, solltest du ausserdem Zugriff auf ein gültiges Entwicklerzertifikat sowie ein für die App passendes Provisioning Profile haben.

Als Standard enthält jedes iOS-Projekt ein Test-Target, dass dazu dient, Tests auszuführen. Dieses Target verwenden wir genau dafür. Es empfiehlt sich im Vorfeld, vor Einrichtung des Jobs darauf zu achten, dass die Tests wirklich durchlaufen. Über Command-U können die Tests gefahren werden, um das zu validieren.

Das Projekt muss mittels git oder subversion gehostet sein. Wie das genau geht liegt ausserhalb des Umfangs dieser Anleitung, dazu gibt es allerdings im Internet zahlreiche Resourcen. Wir empfehlen bitbucket oder github als zuverlässige Hostingprovider für git-Repositories.

Ist das Projekt eingecheckt, kann es weitergehen.

4 Der Job

Wir fangen an, indem wir auf der Startseite einen neuen Job bzw. ein neues Element anlegen. Als Vorlage wählen wir das Free Style-Template. Mit dieser Vorlage erstellt Jenkins ein komplett leeren Job, den wir nach unseren Wünschen anpassen können.

Nach dem Anlegen werden wir auf die Einrichtungsseite weitergeleitet.

Ein Job, so wie Jenkins ihn kennt, ist in mehrere Phasen unterteilt, die auch in dieser Reihenfolge konfiguriert werden sollten. Oben befinden sich die Stammdaten. Wichtig ist neben dem Namen hier die Option, ein Projekt deaktivieren zu können. Das bedeutet, das ein Projekt nicht mehr gebaut wird, auch wenn z.B. Änderungen in git eingecheckt werden oder andere Trigger einen Build veranlassen würden. Das ist nützlich wenn ein Projekt abgeschlossen ist oder andere Umstände die Entwicklung pausieren. Ein deaktiviertes Projekt kann jederzeit reaktiviert werden.

Die nächste bedeutende Phase ist die Wahl des Source-Code-Management Systems. Damit beantworten wir Jenkins die Frage, wo der Code für das Projekt her-

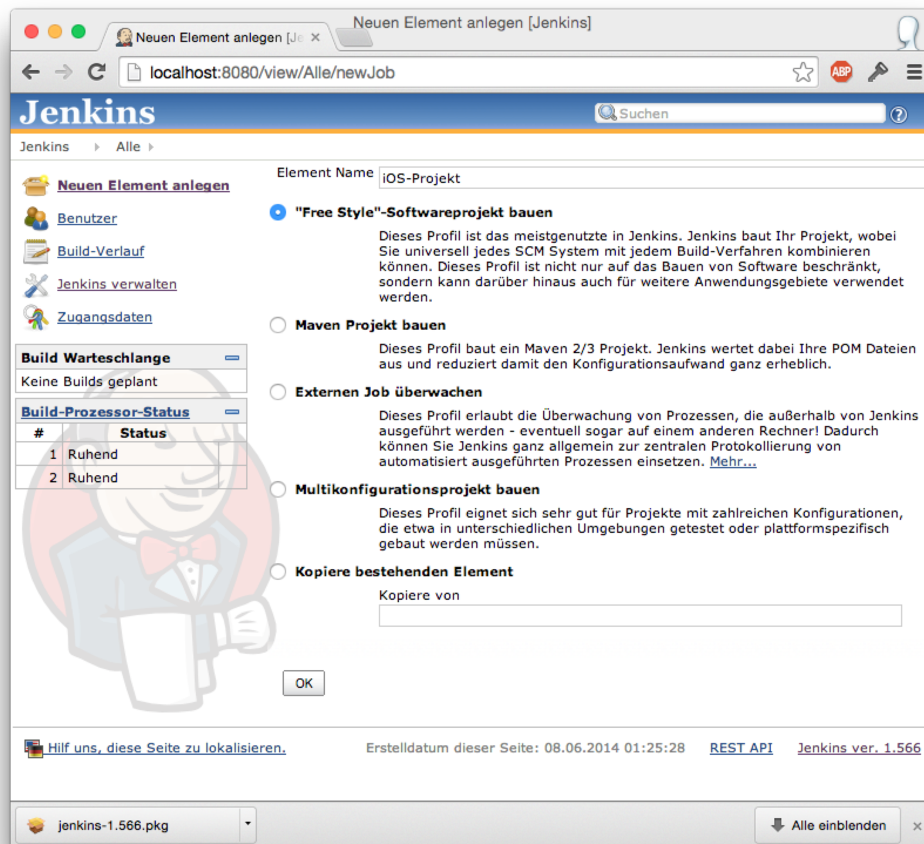

Abbildung 3: *Neues Projekt*

kommt. Hier trägt man unter git die URL des Repositories ein, in dem sich das Projekt befindet. Für unser Beispielprojekt wäre das git@bitbucket.org:derwildemomo/mtddemo.git.

4.0.1 Schlüsselpaar für die Anmeldung bei Github oder Bitbucket erzeugen

Da die wenigsten Repositories öffentlich sind muss man sich für das Abrufen des Codes bei seinem Provider anmelden. Neben der üblichen Nutzername/Passwort-Variante, die grundsätzlich unsicher ist, steht bei allen großen Providern die Möglichkeit bereit, sich über einen SSH-Key zu authentifizieren. Dazu muss man ein Schlüsselpaar anlegen, von dem der öffentliche Teil bei dem Provider hinterlegt ist und der private Teil auf der lokalen Maschine. Um das für Jenkins zu tun, müssen wir uns im Terminal als dieser Benutzer anmelden und ein neues Schlüsselpaar anlegen:

```
$ sudo su jenkins
$ ssh-keygen
    Generating public/private rsa key pair.
    Enter file in which to save the key (/Users/Shared/Jenkins/.ssh/id_
    Created directory '/Users/Shared/Jenkins/.ssh'.
```

Jetzt liegen in /Users/Shared/Jenkins/.ssh/ zwei Dateien: id_rsa enthält den privaten Schlüssel und sollte den Rechner nie verlassen, sowie id_rsa.pub, die den Teil des Schlüssels enthält den man bei Bitbucket und Github in seinen Benutzerdaten hinterlegen kann. So spart man sich bei der Anmeldung das Passwort und Jenkins kann die Quelldateien laden.

Um diesen Schlüssel nun auch für die Anmeldung zu nutzen, müssen wir nach dem Eintragen der Repository-URL noch sog. Credentials anlegen. Dazu die Option 'Add' auswählen und in dem erscheinenden Dropdown '

5 Build-Auslöser

Nachdem Jenkins jetzt weiss, woher der Code für den Job kommt, stellt sich die Frage wann Jenkins aktiv werden soll – genannt Build-Auslöser.

Es gibt mehrere Strategien, jede passend für unterschiedliche Anforderungen. *Nightly Builds* werden jeden Tag um die selbe Uhrzeit, meist Nachts, gebaut und dienen hauptsächlich dazu, langwierige Tests auszuführen, die Tagsüber zu viele Ressourcen in Anspruch nehmen würden.

Für den develop-Branch, der die aktiven Entwicklungsergebnisse enthält empfiehlt sich eine Strategie, bei der nach jeder Änderung in git ein Build angestoßen wird.

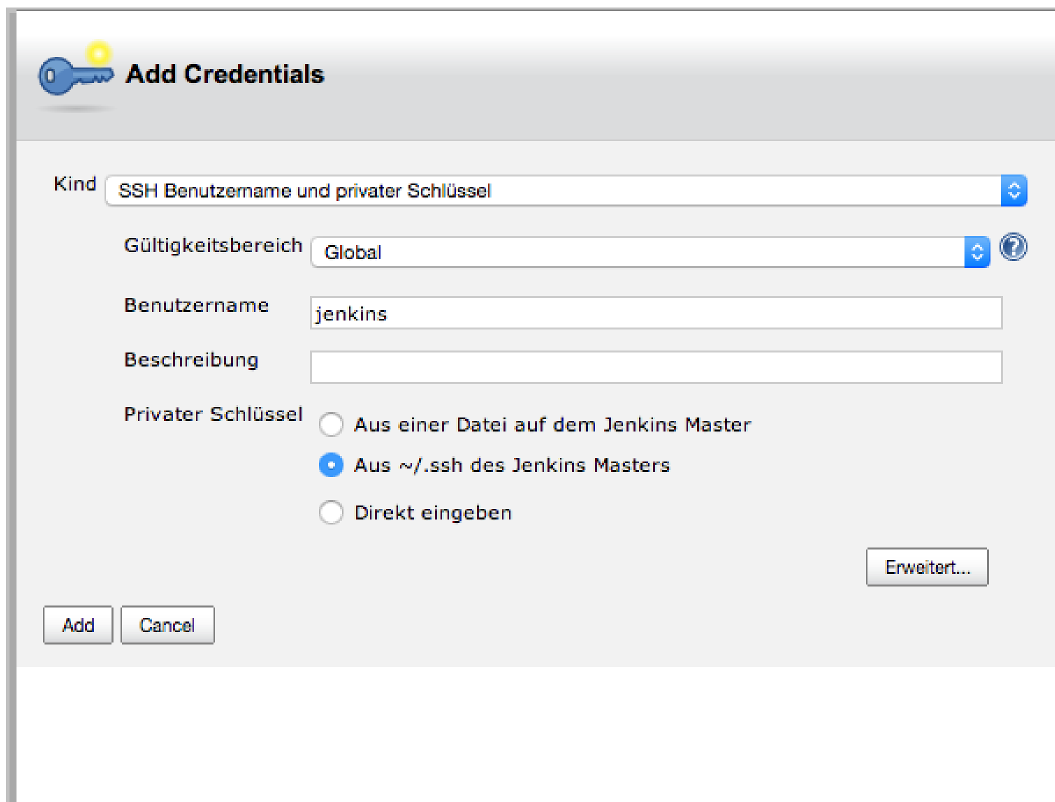

Abbildung 4: *SSH Credentials hinzufügen*

Zu guter Letzt kann man für Builds, die auf dem Master-Branch aufbauen, also fertige und Releasefähige Versionen darstellen, manuell bauen bzw. ein Build selbst anstoßen.

Wir benutzen den zweiten Ansatz und bauen nach jeder Änderung im git, allerdings lauschen wir auf dem Master-Branch. Um das zu bewerkstelligen, wählen wir "Source Code Management abfragen" und geben in das Textfeld $* * * * *$ ein. So wird jede Minute das SCM abgefragt.

> Für den produktiven Einsatz sollte hier ein größeres Abfrageintervall ausgewählt werden, z.B. stündlich.

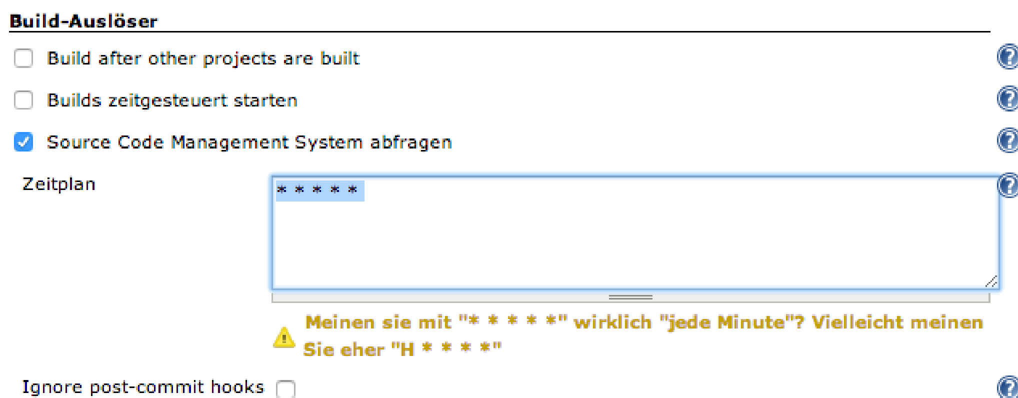

Abbildung 5: *Build-Auslöser einrichten*

6 Buildumgebung

Bevor es ans tatsächliche bauen des Projekts geht, muss noch die Buildumgebung konfiguriert werden. Viele Projekte kommen hier ohne viel Arbeit aus, für iOS-Projekte sind allerdings ein paar Schritte extra notwendig. Warum?

Um ein Projekt nicht nur kompilieren, sondern auch ein installierbares Artefakt, also ein IPA erstellen zu können benötigt xcodebuild ein Entwicklerzertifikat und ein gültiges Provisioning Profile. Beides sollte zur Verfügung stehen.

6.1 Entwicklerzertifikat in Jenkins laden

> Diese Guide geht davon aus, dass in deiner Keychain/Schlüsselbundverwaltung ein gültiges Entwicklerzertifikat liegt.

Die nächsten Schritte absolvieren wir in dem Programm "Schlüsselbundverwaltung", das dazu geöffnet sein sollte.

Als erstes legen wir über alt-command-n einen neuen Schlüsselbund an. In diesem Schlüsselbund speichern wir unser Entwicklerzertifikat. Nur dieser Schlüsselbund wird Jenkins zur Verfügung gestellt. Los geht's!

In unserem Anmelde-Schlüsselbund sollte ein gültiges Entwicklerzertifikat liegen. Wichtig ist, auf der linken Seite unten den Filter auf 'Zertifikate' einzustellen. Gibt man nun iPhone Developer in die Suchbox oben rechts ein, sollte man genau ein gültiges Zertifikat finden. Je nach Entwicklungsumfeld können auch mehrere gefunden werden, dann muss das für das Projekt richtige Profil ausgewählt werden.

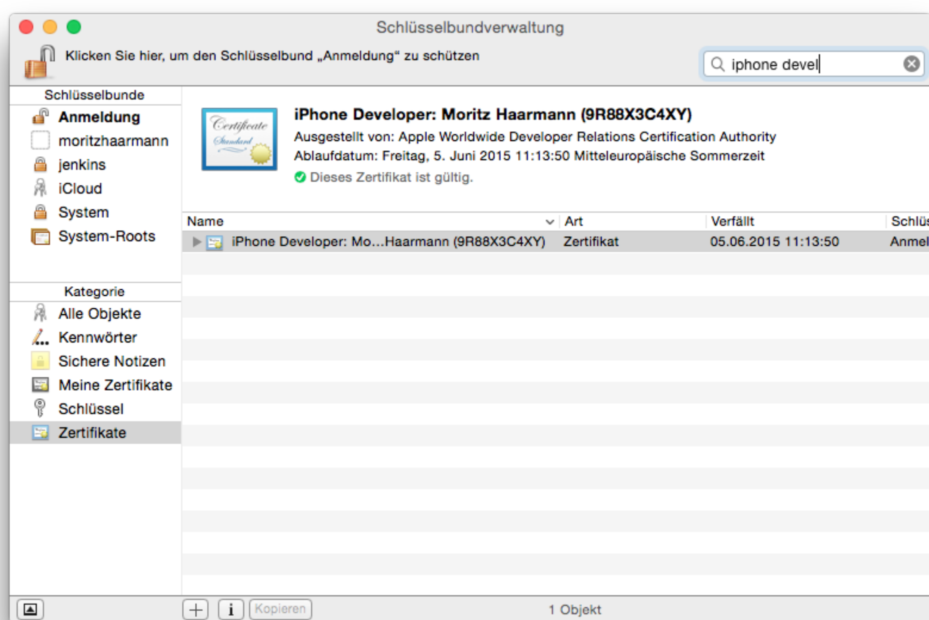

Abbildung 6: *Keychain hat das richtige Profil*

Mittels Drag and Drop ziehen wir dieses Zertifikat in den neuen Schlüsselbund. Das wars auch schon.

Jetzt ist ein guter Zeitpunkt um unsere bisherige Konfiguration in Jenkins zu speichern, da wir kurz in die Jenkins-Verwaltung müssen um die Keychain hochzuladen. Unter 'Jenkins verwalten' befindet sich der Menüpunkt 'Keychains and Provisioning Profiles Management', der jetzt relevant ist.

Prinzipiell können über das Upload-Formular dort Provisioning Profiles und

Zertifikate hochgeladen werden, die später zum bauen benutzt werden können. Als erstes laden wir unsere Keychain hoch.

Völlig unerwartet möchte das Plugin jetzt auch noch unsere *Code Signing Identity* wissen. Das ist der *Common Name* in unserem Entwicklerzertifikat. Durch einen Rechtsklick auf das Zertifikat in Jenkins und den Menüpunkt Information kann man diesen Wert extrahieren, es ist der Wert zu *Allgemeiner Name*.

Abbildung 7: *Der Identity Name*

Hat man alles richtig gemacht, sollte die Konfiguration so wie im Screenshot unten aussehen.

Als nächstes muss noch das passende Provisioning Profile hochgeladen werden, allerdings sind dazu ausser dem hochladen keine weiteren Schritte notwendig.

Bevor wir hier allerdings fertig sind, muss noch der vollständige Pfad zu den Profilen eingetragen wird. Hat man darauf verzichtet, während der Installation Amok zu laufen, ist dieser Pfad

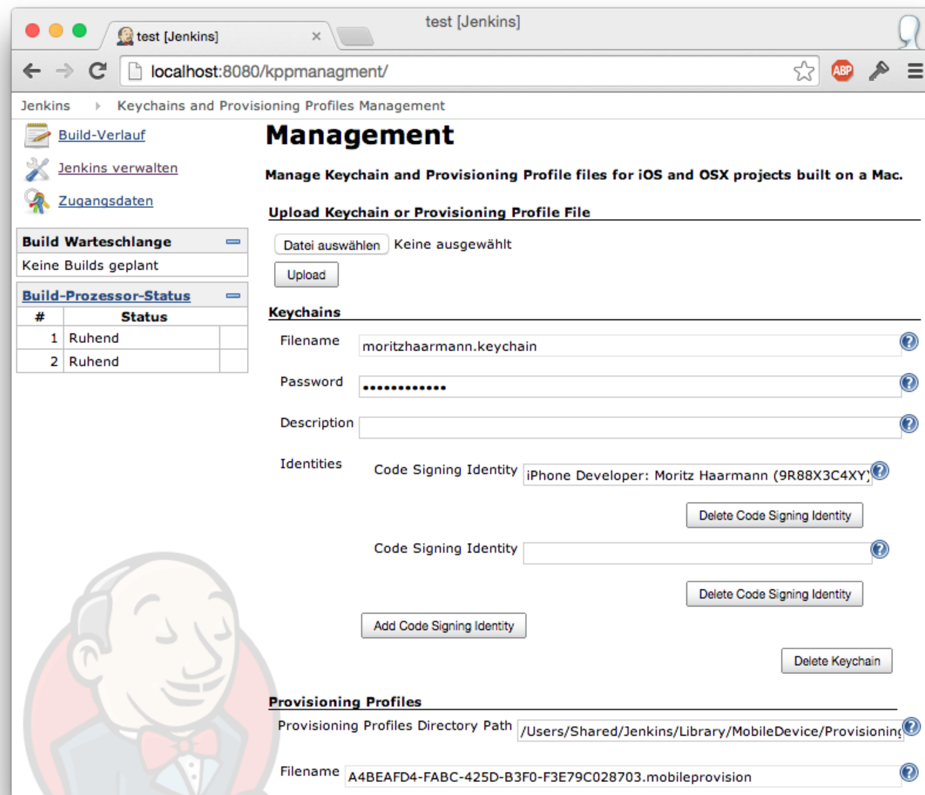

Abbildung 8: *Das konfigurierte Plugin*

```
/Users/Shared/Jenkins/Library/MobileDevice/Provisioning Profiles
```

Wir sind hier fertig, zurück zu unserem Projekt!

7 Buildumgebung

7.1 Keychains and Code Signing Identities

Wir wählen den Punkt über die Checkbox aus und selektieren unsere Keychain und die richtige Identity.

Neben dem Eintrag Variables, der im Moment leer ist und auch nicht bearbeitet werden kann befindet sich ein blaues Fragezeichen. Ein Klick darauf offenbart, dass das Plugin drei Variablen anlegt: ${KEYCHAIN_PATH}, ${KEYCHAIN_PASSWORD} und ${CODE_SIGNING_IDENTITY}. Diese Variablen werden später an Xcodebuild übergeben, so dass richtig gebaut werden kann.

7.2 Mobile Provisioning Profiles

Auch diese Option selektieren wir um das zuvor hochgeladene Profil auszuwählen.

Jetzt ist die Buildumgebung konfiguriert und wir können endlich unser Projekt bauen.

8 Buildverfahren

Wie soll Jenkins mit unserem Code verfahren? Genau das legen wir hier fest. Neben den üblichen Verdächtigen wie Shell-Skript ausführen und einigen anderen Buildverfahren steht uns dank dem Xcode-Plugin auch genau dieser Build-Schritt zur Verfügung. Wir fügen Xcode als Build-Schritt ein.

Die Standardeinstellungen dieses Buildverfahrens bauen alle verfügbaren Targets ohne am Ende ein IPA-File zu erzeugen oder den erstellten Code zu signieren. Diese Einstellungen sind also ausreichend, um zu überprüfen ob der Build lauffähig ist, jedoch nicht, um eine Version zu erstellen, die an Tester verteilt werden kann. Die Konfiguration dafür ist allerdings recht einfach.

Die meisten Konfigurationsoptionen verstecken sich hinter den Settings-Buttons. Im Abschnitt 'General build settings' werden die allgemeinen Einstellungen vorgenommen.

Grundsätzlich sollte die CI immer einen sauberen Build machen, d.h. die Option 'Clean before build' empfiehlt sich sehr. Je nach dem was man vor hat, kann als

Configuration Release oder Debug gewählt werden, das hängt vom Kontext ab. Wir benutzen Release.

Es liegt auf der Hand dass 'Pack application and build .ipa' eine sinnvolle Option sein kann, wenn man ein Ipa- File möchte. Das filename-pattern und die output directory kann leer gelassen werden, es werden dann die Standardeionstellungen verwendet.

Xcode
General build settings
Target
Leave empty for all targets
Clean before build? Yes
This will delete the build directories before invoking the build.
Allow failing build results? Yes
Checking this option will prevent this build step from failing if xcodebuild exits with a non-zero return code.
Generate Archive?
Checking this option will generate an xcarchive of the specified scheme. A workspace and scheme are are also needed for archives
Configuration Release
This is the name of the configuration as defined in the Xcode project.
Pack application and build .ipa?
.ipa filename pattern
A pattern for the ipa file name. You may use ${VERSION} and ${BUILD_DATE} (yyyy.MM.dd) in this string
Output directory
The output directory for the .ipa file, relative to the build directory.

Abbildung 9: *Xcode general build settings*

Die nächste Sektion ist etwas spannender, aber Dank unserer Konfiguration der Buildumgebung auch schnell absolviert - es geht darum, die notwendigen Werte für die Code Signing Identity sowie das Provisioning Profile zu hinterlegen.

Das Keychain and Mobile Provisioning Plugin stellt diese Werte als Variablen zur Verfügung, wie oben vermerkt. Diese Variablen benötigen wir jetzt.

> In Keychain password die Variable ${KEYCHAIN_PASSWORD} einfügen. Auch wenn das Feld eine Passworteingabe ist, werden hier Variablen akzeptiert. Das sieht komisch aus, funktioniert aber.

Bei den Advanced Build options verändern wir im Moment nocht nichts, allerdings setzen wir einen Haken in der "Provide version number and run avgtool"-Checkbox. So ist sichergestellt, dass unsere erstellten Versionen sauber durchnummeriert sind und sich so später gut auseinanderhalten lassen. In dem ersten Feld dort geben wir ${BUILD_NUMBER} ein, so dass die technische Version immer der Build-Nummer entspricht. Das andere Feld wird leer gelassen.

Code signing & OS X keychain options
Code Signing Identity ${CODE_SIGNING_IDENTITY}
Override the code signing identity specified in the project
Embedded Profile ${PROVISIONING_PROFILE}
The relative path to the mobileprovision to embed, leave blank for no embedded profile
Unlock Keychain?
Keychain none (specify one below)
The globally configured keychain to unlock for this build.
Keychain path ${KEYCHAIN_PATH}
The path of the keychain to use to sign the IPA.
Keychain password ••••••••••••••••••••
The password to use to unlock the keychain.

Abbildung 10: *Code signing & OS X keychain options*

Jetzt können wir unsere Konfiguration speichern und ausprobieren! Im Normalfall sollte der Build nach einem Klick auf "Jetzt Bauenäuf der linken Seiten des Bildschirms problemlos durchlaufen. Sollte das nicht der Fall sein hilft eventuell folgende Troubleshooting-Liste, die die wichtigsten Schritte noch einmal zusammenfasst.

- **Xcode ist auf dem Rechner installiert**, auf dem Jenkins läuft.
- Das Projekt **befindet sich vollständig in git**. Kann einfach überprüft werden indem das Projekt in einem neuen Verzeichnis direkt von github oder bitbucket geholt wird, mittels xcodebuild in einem Terminal dann ausprobieren ob das bauen klappt. Wenn nicht, liegt das Problem wahrscheinlich dort.
- Die kopierte Keychain enthält ein **gültiges Entwicklerprofil** und das hochgeladene Provisioning Profile passt zur App und Identity. Das kann über Xcode recht einfach validiert werden: Baut die App und läuft auf einem iPhone, sind hier wohl keine Fehler passiert.
- Im Xcode-Plugin wurden die richtigen Option ausgewählt, insbesondere was Code Signing angeht, siehe die Beschreibung weiter oben.

9 Post-Build

Was machen wir mit den erzeugten Ergebnissen? Zuerst archivieren wir diese sog. Artefakte. Das tut man, in dem man Jenkins ein File Pattern mitgibt, dass auf die zu archivierenden Dateien zutrifft. Wir fügen dazu den "Artefakte Archivieren"-Schritt ein und setzen das Pattern auf

```
build/Release-iphoneos/*.ipa, build/Release-iphoneos/*.zip
```

So werden die ipa-Dateien und die dazugehörigen dSYM-Dateien archiviert, die notwendig sind, um Crash Reports zu symbolicaten, d.h. Zeilen- und Dateinamen zuzuordnen. Diese Dateien werden von Jenkins dauerhaft gespeichert.

9.1 Build bei HockeyApp bereitstellen

Wenn du schon einen Account bei HockeyApp hast sind die nächsten Schritte in weniger als 2 Minuten erledigt, ansonsten ist jetzt der richtige Zeitpunkt um sich auf `hockeyapp.net` einen Account zu klicken. Trial-Accounts gibt es kostenlos.

Um von Jenkins aus eine Version hochladen zu können ist ein API-Token notwendig. Das kann bei HockeyApp in der Account-Verwaltung unter dem Punkt API-Tokens anlegen, siehe Screenshot. Die Rechte sollten für Jenkins auf Upload und Release beschränkt werden, das ist ausreichend.

> Sollte der Buildserver Ziel eines Angriffs sein, kann das Token einfach gelöscht werden. So kann sichergestellt werden, dass die Kontrolle nicht an einem Passwort hängt und die Sicherheit der Daten bei HockeyApp gewährleistet ist.

Als nächstes muss noch eine App angelegt werden, direkt in HockeyApp. Das geht über das Dashboard. Der Dialog hält die Option bereit, eine App manuell zu erstellen. Als Platform dort geben wir iOS an, sowie den Titel und den korrekten Bundle Identifier unseres iOS-Projekts, in meinem Fall de.moritzhaarmann.mtddemo. Nach dem Anlegen steht eine App ID bereit die wir für die Konfiguration des Uploads ebenfalls benötigen.

Als nächstes wird 'Upload to HockeyApp' als Post-Build Schritt eingefügt. Wir übernehmen das eben angelegte API-Token und die App-Id. Wie im Screenshot gezeigt übernehmen wir ausserdem die Dateinamen für das Ipa und die dSYM-File, beide Dateien werden von HockeyApp benötigt.

Klappt man die erweiterten Optionen auf gibt es noch die Einstellung 'Use Changelog as Release notes', was dazu führt, dass die gesammelten git-commits die in einem Build verarbeitet werden, als Release note auf HockeyApp landen.

Herzlichen Glückwunsch, dein Projekt wird automatisch gebaut und landet bei HockeyApp!

Artefakte archivieren

Dateien, die archiviert werden sollen: build/Release-iphoneos/*.ipa, build/Release-iphoneos/*.zip

Erweitert...

Löschen

Upload to HockeyApp

API Token: 5dcbc2e545b44a0f970954df2404dbb0

Public ID / App ID: 0434c73ad6601c732a14921dfd4fbcfe

App file (.ipa, .app.zip, .apk): build/Release-iphoneos/MTDiphone-${BUILD_NUMBER}.ipa

Symbols (.dSYM.zip or mapping.txt): build/Release-iphoneos/MTDiphone-${BUILD_NUMBER}-dSYM.zip

Release notes

Use Markdown for Release notes

Use changelog as release notes

Release notes filename

Allow downloads

Restrict downloads to tags

Notify team

Clean up old versions at HockeyApp

Number of old versions to keep

Use new API URL

Löschen

Abbildung 11: *Die fertige Post-Build Konfiguration*