HANSER

Leseprobe

zu

„Das Swift-Handbuch“

von Thomas Sillmann

Print-ISBN: 978-3-446-45505-4
E-Book-ISBN: 978-3-446-45730-0

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45505-4
sowie im Buchhandel

Inhalt

Vorwort

Liebe Leserin, lieber Leser,

als Apple auf der World Wide Developers Conference im Juni 2014 für alle Welt überraschend eine komplett neue Programmiersprache vorstellte, war das etwas ganz Besonderes für mich und meine Entwicklerkollegen. Bis zu diesem Zeitpunkt war Objective-C die Sprache der Wahl, wenn es um die Programmierung für Apple-Plattformen ging. Das war weiß Gott nichts Schlechtes, aber man merkt Objective-C nun einmal sowohl ein gewisses Alter wie auch diverse Eigenheiten an.

Swift ist da ganz anders. Schon beim Starten in die Programmierung mit Swift merkt man schnell, wie einfach und verständlich doch vieles von der Hand geht. Dazu kommen clevere Konzepte wie die Optionals und das Error Handling (über die Sie noch ausführlich in diesem Buch lesen werden), die Swift zu einer sehr sicheren Sprache machen, mit der man gerne programmiert. So zumindest geht es den meisten, die entweder den Sprung gewagt und sich von Objective-C kommend Swift zugewandt haben oder die frisch in die Entwicklung für Apple-Plattformen mit Swift eingestiegen sind.

Für Apple ist Swift heute ein immens wichtiger Baustein der Anwendungsentwicklung. Swift ist inzwischen Open Source und bereits bei Version 5 angekommen. Auf Apples Entwicklerkonferenzen sieht man ausschließlich Code-Beispiele, die in Swift geschrieben sind. Und Apple hat sogar manche der eigenen Apps in Swift komplett neu entwickelt. Der Weg scheint klar: Swift gehört die Zukunft.

Mit diesem Buch möchte ich Ihnen, liebe Leserin, lieber Leser, das notwendige Wissen vermitteln, um selbst eigene Apps mit Swift für iOS, macOS und Co. entwickeln und über den App Store vertreiben zu können. Hierbei ist es mir wichtig, Sie nicht mit einer Vielzahl von fertigen Beispielprojekten zu erschlagen und Ihnen zu erklären, wie Sie diese nachprogrammieren. Stattdessen möchte ich Ihnen einen möglichst umfangreichen und vielseitigen Überblick darüber geben, was Sie alles für spannende Funktionen in Apps für iPhone, iPad, Mac, Apple Watch und Apple TV umsetzen können und Ihnen ausführlich erklären, wie Sie dabei vorzugehen haben und worauf Sie achten müssen. Das Buch soll Ihnen sowohl als Neuling beim Lernen helfen, wie auch alten Hasen noch das ein oder andere Detail vor Augen führen, das einem bisher möglicherweise entgangen war.

Anhand dieser Prämisse ist auch der grundlegende Aufbau des Buches entstanden, der sich in insgesamt sechs verschiedene Teile untergliedert. Ganz zu Beginn steht die titelgebende Programmiersprache Swift. Im *ersten Teil* erfahren Sie in insgesamt fünfzehn Kapiteln alles,

um eigenen Swift-Code schreiben und die vielen spannenden Facetten dieser Sprache wie Generics und die bereits erwähnten Optionals optimal einsetzen zu können. In diesen Kapiteln geht es nur um Swift und nichts anderes.

Im *zweiten Teil* geht es weiter mit Xcode, der Entwicklungsumgebung von Apple. Sie erfahren, wie Sie die IDE installieren, wie sie aufgebaut ist und wie Sie durch Projekte navigieren und sich darin zurechtfinden.

In *Teil III* geht es schließlich um die Betrachtung der verschiedenen Plattformen von Apple und darum, wie Sie Apps für diese entwickeln. Jedes Betriebssystem – macOS, iOS, watchOS und tvOS – verfügt in Teil 3 über wenigstens ein Kapitel, in dem Sie alles über die grundlegende Architektur der jeweiligen Plattform sowie über die Besonderheiten bei der App-Entwicklung erfahren. Auch gebe ich Ihnen eine Einschätzung, für welche Arten von Apps die verschiedenen Betriebssysteme gedacht sind – und für welche nicht. In einem abschließenden Cross-Platform-Kapitel stelle ich Ihnen außerdem einige spannende Funktionen vor, die für alle Plattformen von Apple gleichermaßen relevant sind.

Mir war es wichtig, darauf zu achten, alle Kapitel des dritten Teils möglichst unabhängig voneinander zu gestalten. Sie haben bereits grundlegende Erfahrung mit der iOS-Entwicklung? Dann steigen Sie direkt in das zugehörige Kapitel zur App-Entwicklung ein und überspringen Sie die Basics! Dafür ist Ihnen die Programmierung für macOS noch vollkommen fremd? Kein Problem, das Grundlagenkapitel zur macOS-Entwicklung nimmt Sie an die Hand und zeigt Ihnen, wie die Plattform aus Sicht eines App-Entwicklers funktioniert. Auch die Themen, die in den einzelnen Kapiteln behandelt werden, sind so geschrieben, dass Sie direkt mit einer bestimmten Funktion (beispielsweise dem Erstellen von Tabellen unter iOS) einsteigen können, ohne vorher die anderen Kapitel zwingend lesen zu müssen.

Nach der Betrachtung der verschiedenen Plattformen von Apple geht es im *vierten Teil* des Buches um einige spezielle Frameworks und Technologien, mit denen Sie die Funktionalität Ihrer Apps erweitern können. Dazu gehören unter anderem Themen wie iCloud, Siri oder die Implementierung von Touch ID beziehungsweise Face ID.

In *Teil V* stelle ich Ihnen die Source Control-Möglichkeiten von Xcode vor und zeige Ihnen, wie Sie Unit-, UI- und Performance-Tests für Ihre Apps schreiben und ausführen. Im sechsten und letzten Teil erfahren Sie schließlich alles zur Veröffentlichung Ihrer Anwendungen im App Store, welche Geschäftsmodelle sich umsetzen lassen und wie Sie Apps für Beta-Tests mittels TestFlight verteilen.

Bekanntermaßen lebt die digitale Welt vom ständigen Wandel und Fortschritt. Das betrifft auch die Arbeit von App-Entwicklern. Jedes Jahr werden auf Apples Entwicklerkonferenz WWDC Neuerungen und Änderungen in Bezug auf iOS, macOS und Co. vorgestellt. Auch die Programmiersprache Swift entwickelt sich stetig weiter. Um diesem Wandel Rechnung zu tragen, erhalten Sie für zwei Jahre nach Erscheinen des Swift-Handbuchs (also bis Mai 2021) kostenlose Buch-Updates in PDF-Form. Sie werden persönlich von uns benachrichtigt, wenn neue Updates zum Download zur Verfügung stehen. Registrieren Sie sich dazu einfach unter *www.hanser-fachbuch.de/swift-update* mit dem Passwort von Seite II.

Die kommenden Updates berücksichtigen Änderungen an bestehenden Funktionen, neue Features für die verschiedenen Betriebssysteme von Apple und für die Entwicklungsumgebung Xcode sowie Aktualisierungen von Swift. Darüber hinaus finden Sie aktuelle Artikel zur Programmierung sowie ergänzende Lehrvideos auf meiner Entwickler-Website unter *letscode.thomassillmann.de*.

Für mich persönlich ist die Programmierung mit Swift für die unterschiedlichen Apple-Systeme eine sehr erfüllende Aufgabe und mir war es wichtig, all meine Begeisterung für diese Thematik in dieses Buch einfließen zu lassen. Ich hoffe von Herzen, dass Sie Ihre Freude mit dem Werk haben werden, ganz gleich, ob Sie entweder frisch in die spannende Welt der Swift-Programmierung einsteigen oder als alter Hase noch das ein oder andere Neue lernen, das Ihnen bei zukünftigen Projekten nützlich ist.

Herzlichst,

Ihr Thomas Sillmann

Aschaffenburg, Januar 2019

1 Die Programmiersprache Swift

Die Programmiersprache Swift hat sich seit ihrer erstmaligen Vorstellung im Juni 2014 immens weiterentwickelt und dabei mehrere spannende Meilensteine durchlaufen. Nicht nur, dass wir zwischenzeitlich bereits bei Version 5 von Swift angelangt sind, nein, inzwischen ist die Programmiersprache auch Open Source und besitzt eine eigene Online-Plattform unter *https://swift.org* (siehe Bild 1.1). Dort finden sich ein Blog mit Informationen zur Weiterentwicklung der Sprache, vorgefertigte Downloadpakete, eine Dokumentation, Verweise auf weitere Swift-Projekte und vieles mehr (mehr zu *Swift.org* erfahren Sie in Abschnitt 1.7, „Swift-Ressourcen und weiterführende Informationen“).

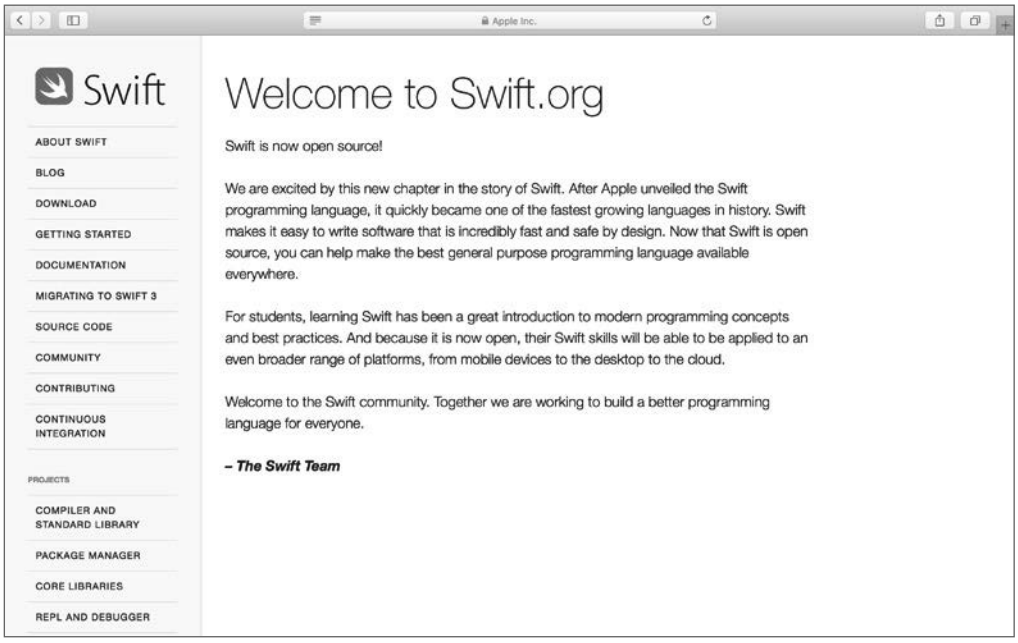


Bild 1.1 Die Plattform Swift.org ist die zentrale Anlaufstelle für die Programmiersprache Swift.

Aufgrund dieser massiven Weiterentwicklungen ist Swift inzwischen mitnichten nur eine Programmiersprache für die Plattformen von Apple. Auf Linux ist es bereits heute möglich, Swift-Code auszuführen, weitere Plattformen werden mit Sicherheit folgen. Dank IBM hat es Swift sogar schon auf die Server und in die Cloud geschafft, woraus sich ebenfalls ganz neue Einsatzgebiete und Möglichkeiten zur Nutzung von Swift für Entwickler ergeben.

Nichtsdestotrotz widmet sich dieses Buch dem Bereich, in dem Swift heute noch immer die größte und relevanteste Rolle spielt: der Entwicklung von Apps für die verschiedenen Plattformen von Apple. Ganz gleich ob macOS, iOS, watchOS oder tvOS: Für all diese Betriebssysteme lassen sich mithilfe von Swift innovative Anwendungen auf Basis der Entwicklungsumgebung Xcode programmieren, und hierfür liefert Ihnen dieses Buch alle grundlegenden und essenziellen Informationen. Sie dürfen gespannt sein. :)

1.1 Die Geschichte von Swift

Viele Details sind über die genaue Entstehungsgeschichte von Swift nicht bekannt. Was man weiß, ist, dass der Apple-Entwickler Chris Lattner wohl in gewisser Weise als „Vater“ von Swift bezeichnet werden kann. Er begann die Entwicklung von Swift im Juli 2010 aus eigenem Antrieb heraus und zunächst im Alleingang. Ab Ende 2011 kamen dann weitere Entwickler dazu, währen das Projekt im Geheimen bei Apple fortgeführt wurde. Das erste Mal zeigte Apple die neue Sprache der Weltöffentlichkeit auf der WWDC (Worldwide Developers Conference) 2014 (siehe Bild 1.2).

Bild 1.2 Auf der WWDC 2014 präsentierte Apple Swift erstmals der Weltöffentlichkeit.

Mit dieser erstmaligen Präsentation von Swift überraschte Apple sowohl Presse als auch Entwickler gleichermaßen. Dabei war die Sprache zunächst – ähnlich wie Objective-C – ausschließlich auf die Plattformen von Apple beschränkt. Ein Mac mitsamt der zugehörigen IDE Xcode von Apple waren also Pflicht, wollte man mit Swift Apps für macOS, iOS, watchOS oder tvOS entwickeln. Im Herbst 2014 folgte dann die erste finale Version von Swift, die Apple den Entwicklern zusammen mit einem Update für Xcode zugänglich machte.

Im darauffolgenden Jahr sorgte Apple auf der WWDC 2015 dann für die nächste große Überraschung. Sie präsentierten nicht nur die neue Version 2 von Swift, sondern gaben auch bekannt, dass Swift noch im gleichen Jahr Open Source werden würde. Dieses Versprechen wurde dann am 03. Dezember 2015 umgesetzt und Apple startete die Plattform *Swift.org*, um darüber zukünftig alle Weiterentwicklungen und Neuerungen zu Swift zusammenzutragen.

Auf der WWDC 2016 folgte sodann die Vorstellung der neuen Version 3 von Swift, die im Herbst desselben Jahres offiziell veröffentlicht wurde. Es folgten Swift 4 und Swift 5. Letztere ist die aktuelle Version von Swift und erschien Anfang 2019.

1.2 Swift-Updates

Die Sprache Swift hat in den wenigen Jahren, die sie bisher verfügbar ist, bereits einige große Versionssprünge hingelegt. Gerade am Anfang war das für Swift-Entwickler der ersten Stunde durchaus ein Problem, denn diese Versionssprünge änderten den Code und die Syntax von Swift bisweilen so stark, dass sich Projekte, die mit einer früheren Swift-Version als der aktuellen geschrieben wurden, nicht mehr kompilieren und damit ausführen ließen.

Zwar bietet Apple in seiner Entwicklungsumgebung Xcode einen Assistenten, der Swift-Code einer älteren Version nach der aktuellen migriert, aber meistens konnte auch dieser nicht alle Probleme und Fehler vollumfänglich auflösen, was bedeutete, dass Entwickler – je nach Größe des zugrunde liegenden Projekts – mal mehr, mal weniger Zeit damit verbringen mussten, ihren Code auf die neue Swift-Version zu aktualisieren und entsprechend anzupassen.

Diese Problematik soll ab Version 5 von Swift nun ein Ende haben. Natürlich wird es in Zukunft weitere Versionen der Programmiersprache geben, diese sollen nun aber nicht mehr Code, der in einer älteren Swift-Version geschrieben wurde (solange er mindestens auf Version 5 basiert), gänzlich unbrauchbar und unausführbar machen. Swift 5 stellt somit einen gewissen Meilenstein in dieser noch jungen Programmiersprache dar, ein idealer Zeitpunkt also, sich spätestens jetzt einmal damit auseinanderzusetzen.

Trotzdem sollen und wollen Sie als Swift-Entwickler natürlich auch up to date bleiben und wissen, wie sich die Sprache weiterentwickelt und welche Neuerungen sie im Laufe der Zeit mit sich bringt. In Abschnitt 1.7, „Swift-Ressourcen und weiterführende Informationen“, am Ende dieses Kapitels, stelle ich Ihnen einige wichtige und hilfreiche Ressourcen vor, die Ihnen dabei helfen, Ihr Swift-Know-how stets auf dem neuesten Stand zu halten.

1.3 Voraussetzungen für die Nutzung von Swift

Swift wird aktuell auf den folgenden Plattformen unterstützt:

- macOS
- Linux

Auf diesen kann Swift-Code ausgeführt und mithilfe passender Tools geschrieben werden. Unter macOS ist Apples Entwicklungsumgebung Xcode die erste Wahl, wenn es um die Entwicklung mit Swift geht. Unter Linux stellt Apple bisher ausschließlich die sogenannte *REPL* (Read Eval Print Loop) bereit, die es erlaubt, Swift-Code über das Terminal auszuführen. Darüber hinaus kann Swift-Code noch auf den weiteren Apple-Plattformen iOS, watchOS und tvOS ausgeführt werden.

Wer ernsthaft mit Swift entwickeln möchte, sollte zum jetzigen Zeitpunkt trotz offiziellem Linux-Support nichtsdestoweniger vorzugsweise auf einen Mac mitsamt macOS zurückgreifen. Dank der vollwerten Entwicklungsumgebung Xcode, die Apple kostenlos bereitstellt und in der Swift vollumfänglich integriert ist, ist diese IDE noch immer das Mittel der Wahl für professionelle Software-Entwicklung mit Swift. Die ebenfalls unter Linux zur Verfügung stehende REPL eignet sich ideal für Tests und zum Ausprobieren verschiedener Eigenschaften und Mechanismen der Programmiersprache.

Swift Playgrounds auf dem iPad

Neben den genannten Plattformen ist es auch möglich, Swift-Code auf Apples iPad zu schreiben und auszuführen. Seit Version 10 von iOS – dem Betriebssystem des iPad – bringt dieses nämlich eine kostenlose App namens *Swift Playgrounds* mit. Darüber ist es möglich – wie der Name bereits andeutet – sogenannte Playgrounds zu erstellen und darin Swift-Code zu schreiben und ausführen zu lassen (siehe Bild 1.3). Die App kompiliert die Eingaben und gibt direkt Feedback über mögliche Syntaxfehler oder andere Probleme.

Da die App keine kompletten Projekte, sondern ausschließlich die Playgrounds verwalten kann, ist sie primär dafür gedacht, einzelne Code-Fragmente zu testen oder eine Idee für eine Funktion umzusetzen und zu überprüfen. Dabei kann die App die erzeugten Playgrounds auch mit Xcode auf dem Mac austauschen, damit diese dort weitergenutzt werden können. Mehr zu Playgrounds erfahren Sie in Abschnitt 1.5, „Playgrounds“.

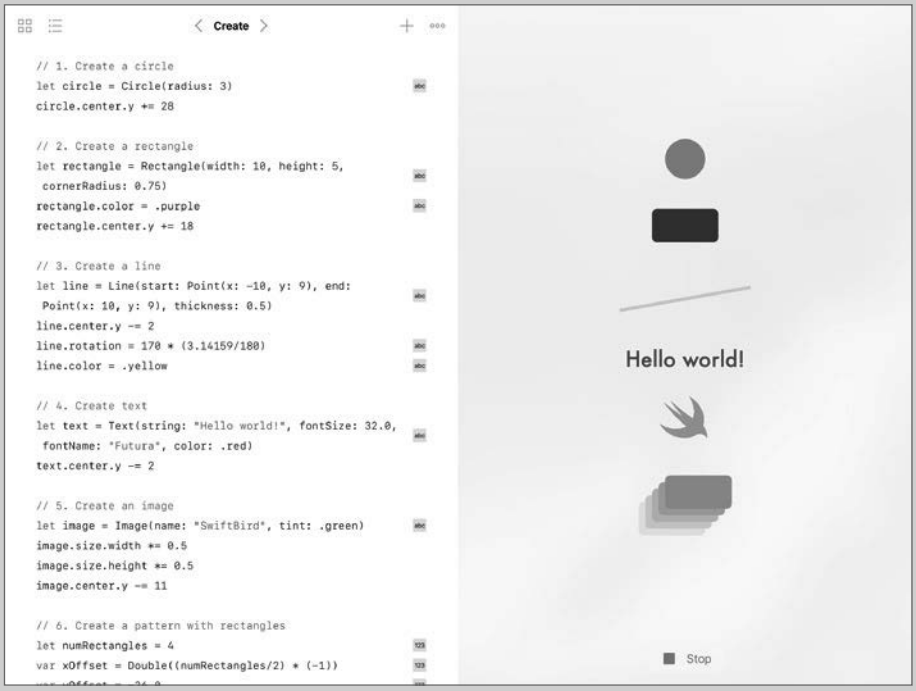


Bild 1.3 Swift Playgrounds erlaubt das Schreiben und Ausführen von Swift-Code auf dem iPad.

1.4 Installation von Swift

Je nachdem, auf welcher Plattform Sie Swift nutzen möchten – macOS oder Linux – verläuft die Installation ein wenig anders und es stehen Ihnen unterschiedliche Tools zur Arbeit mit Swift zur Verfügung (wie im vorherigen Abschnitt 1.3, „Voraussetzungen für die Nutzung von Swift“, beschrieben). Im Folgenden stelle ich Ihnen den Installationsprozess für beide Plattformen im Detail vor.

1.4.1 macOS

Um Swift unter macOS nutzen zu können, brauchen Sie im einfachsten Fall nur Folgendes zu tun: Öffnen Sie die App Store-App, suchen Sie nach Xcode und klicken Sie auf die Schaltfläche *Laden*. Die aktuelle Version von Xcode wird anschließend heruntergeladen und auf Ihrem Mac installiert (siehe Bild 1.4).

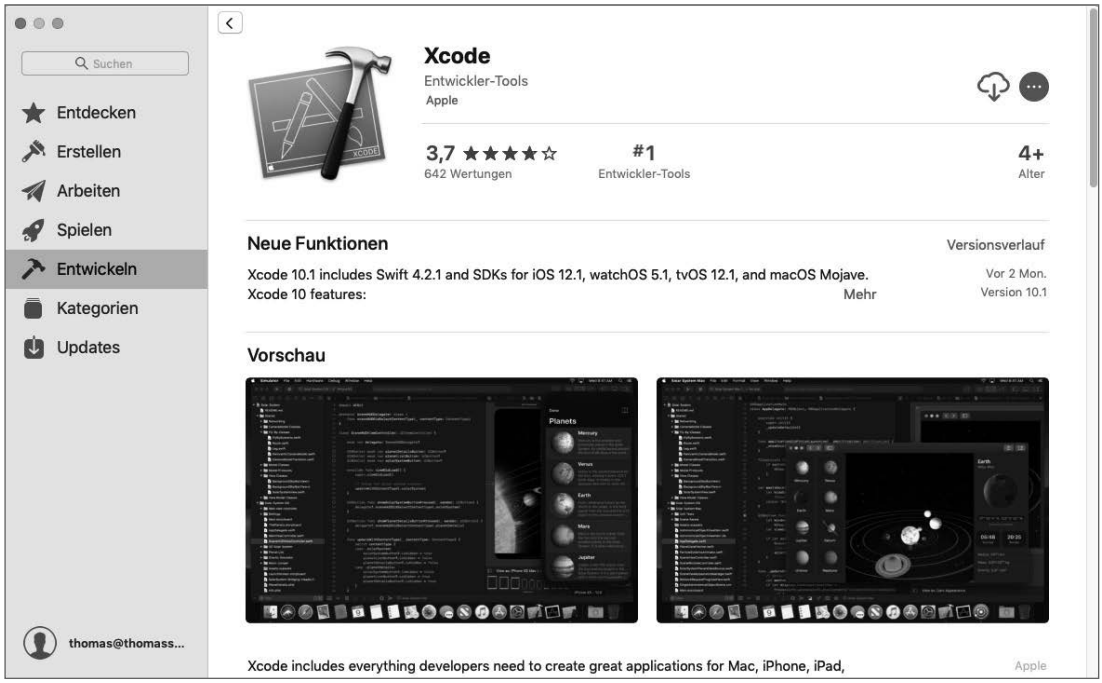


Bild 1.4 Laden Sie auf dem Mac einfach die aktuelle Version von Xcode aus dem Mac App Store, um mit der Entwicklung eigener Swift-Anwendungen zu beginnen.

Über den Mac App Store erhalten Sie immer den jeweils aktuellsten Stable-Release von Xcode. Damit können Sie direkt mit der Swift-Programmierung loslegen, allerdings nur für die jeweils aktuell freigegebene Swift-Version. Wenn Sie sich stattdessen für die Entwicklung mit einer sich noch in der Entwicklung befindlichen neuen Version von Swift interessieren, dann führt kein Weg an *Swift.org* beziehungsweise der Apple Developer-Website *(https://developer.apple.com)* vorbei. Über letztere können Sie – eine Mitgliedschaft im kostenpflichtigen Apple Developer Program vorausgesetzt – Vorabversionen von Xcode herunterladen, die meist auch noch nicht veröffentlichte und noch in der Entwicklung befindliche Aktualisierungen von Swift beinhalten. Alternativ können Sie im Downloads-Bereich von *Swift.org* auf sogenannte *Snapshots* von Swift zurückgreifen (siehe Bild 1.5). Dabei handelt es sich um Packages, die Sie auf Ihren Mac herunterladen und installieren. Anschließend können Sie in den Einstellungen von Xcode im Bereich *Components* zwischen der mit der Entwicklungsumgebung ausgelieferten Version von Swift und der von Ihnen installierten hin und her wechseln. Wählen Sie hierfür zunächst den Reiter *Toolchains* und im nächsten Schritt den gewünschten Snapshot aus (siehe Bild 1.6).

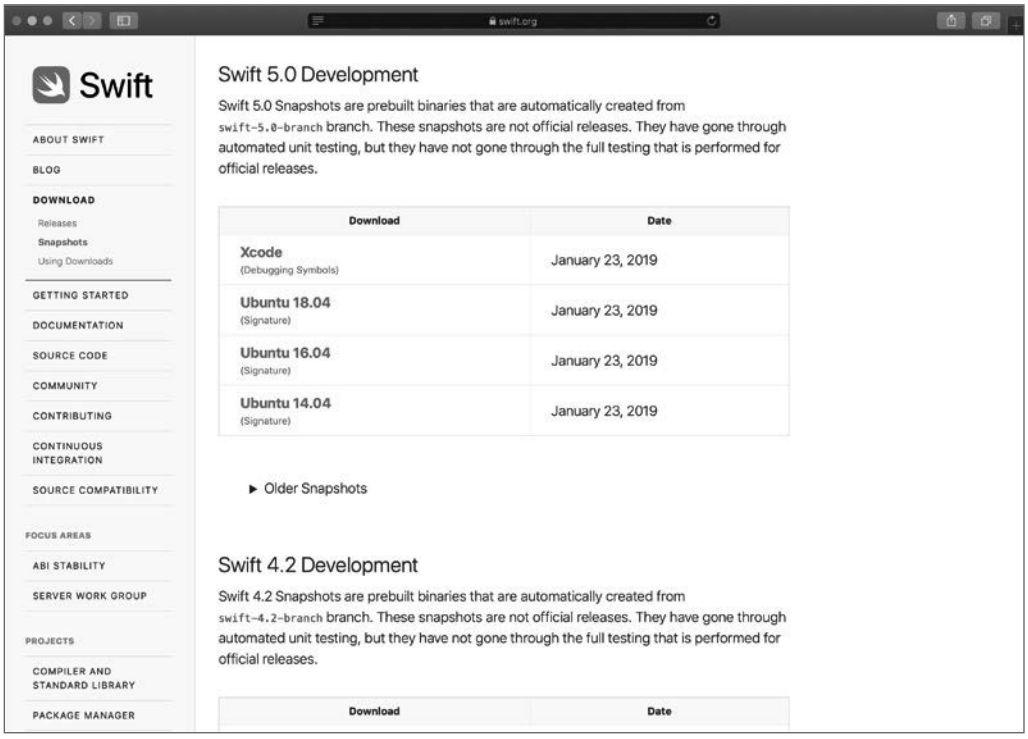


Bild 1.5 Im Downloadbereich von Swift.org finden Sie die aktuellsten Vorabversionen von Swift.

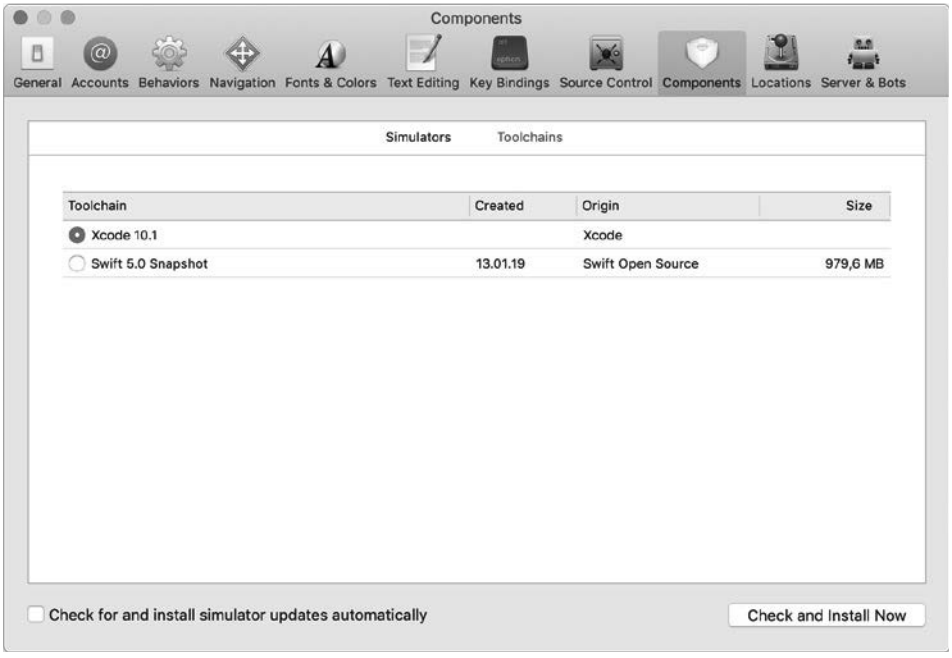


Bild 1.6 In Xcode können Sie zwischen den installierten Snapshots wechseln.

Mehr zum Aufbau, zur Funktionsweise und zur Arbeit mit der Entwicklungsumgebung Xcode erfahren Sie im gleichnamigen zweiten Teil dieses Buches.

1.4.2 Linux

Wenn Sie Swift unter Linux installieren möchten, besteht Ihr erster Gang in Richtung *Swift.org*. Dort finden Sie nach Klick auf *Download* verschiedene vorgefertigte Downloadpakete von Swift für Linux. Die Xcode-Downloads können Sie ignorieren, da Xcode ausschließlich unter macOS zur Verfügung steht.

Die verfügbaren Downloads sind in die Bereiche *Releases* und *Snapshots* unterteilt. Unter *Releases* haben Sie Zugriff auf die jeweils aktuell freigegebene Version von Swift, während Sie unter *Snapshots* Vorabversionen kommender Swift-Updates herunterladen können.

Ganz gleich, für welche Version von Swift Sie sich entscheiden, setzen sich die fertigen Downloadpakete für Linux immer aus zwei Bestandteilen zusammen, die beide separat heruntergeladen werden müssen:

- Binary
- Signature

Bei der Binary handelt es sich um eine bereits kompilierte Version von Swift für die jeweilige Linux-Distribution. Generell kann Swift auch auf anderen Linux-Distributionen installiert und ausgeführt werden, die nicht explizit auf *Swift.org* als fertige Binary angeboten werden. In diesem Fall muss man sich allerdings selbst um einen passenden Port aus den Quelldateien kümmern beziehungsweise nach anderen (seriösen) Quellen im Internet suchen. Die fertige Binary können Sie per Klick auf den Namen der gewünschten Linux-Distribution herunterladen.

Ebenso herunterladen müssen Sie die zugehörige Signatur, die Ihnen als *Signature*-Link direkt unterhalb des Binary-Links der jeweiligen Distribution zum Download angeboten wird (siehe Bild 1.7).

Bild 1.7 Unter Linux müssen Sie sowohl die eigentliche Swift-Binary als auch die Signatur über beide Links herunterladen.

Sind beide Downloads abgeschlossen, kann die eigentliche Installation von Swift unter Linux beginnen. Stellen Sie zunächst sicher, alle benötigten Abhängigkeiten und Programme heruntergeladen und installiert zu haben. Führen Sie dazu den folgenden Befehl im Terminal aus:

```
sudo apt-get install clang libicu-dev
```

Wenn Sie das allererste Mal Swift unter Linux herunterladen und installieren, müssen Sie auch die passenden PGP Keys unter Linux hinzufügen. Geben Sie dazu den nachfolgenden Befehl in das Terminal ein und führen Sie diesen anschließend aus:

```
wget -q -O - https://swift.org/keys/all-keys.asc | gpg --import -
```

Ebenso benötigen Sie – zur Überprüfung der Aktualität und Echtheit der heruntergeladenen Pakete – den passenden Public Key, der für das Signieren verwendet wurde. Diesen können Sie über den folgenden Terminal-Befehl herunterladen und importieren:

```
wget -q -O - https://swift.org/keys/automatic-signing-key-1.asc | gpg --import -
```

Bevor es nun an die eigentliche Installation geht, sollten Sie in jedem Fall die Echtheit der heruntergeladenen Signatur überprüfen. Dazu laden Sie zunächst eine Liste der bereits zurückgezogenen beziehungsweise aufgehobenen Zertifikate herunter, gegen die Sie die Signatur im zweiten Schritt dann prüfen können. Führen Sie dazu zunächst den folgenden Befehl im Terminal aus:

```
gpg --keyserver hkp://pool.sks-keyservers.net --refresh-keys Swift
```

Ist das erledigt, können Sie die Echtheit und Aktualität Ihrer heruntergeladenen Signatur mit dem folgenden Befehl überprüfen:

```
gpg --verify swift-<VERSION>-<PLATTFORM>.tar.gz.sig
```

`<VERSION>` und `<PLATTFORM>` sind dabei durch die entsprechenden Informationen zu ersetzen, die der von Ihnen heruntergeladenen Binary und der Signatur für Linux entsprechen, beispielsweise wie folgt:

```
gpg --verify swift-5.0-ubuntu18.04.tar.gz.sig
```

Wundern Sie sich nicht, falls Sie eine Warnung mit dem Text `This key is not certified with a trusted signature!` erhalten; das ist in Ordnung. Sollte aber die Verifizierung gänzlich fehlschlagen und Sie eine Meldung `BAD Signature` erhalten, sollten Sie die heruntergeladene Binary keinesfalls verwenden und überprüfen, was beim Download schiefgegangen ist beziehungsweise aus welcher Quelle Sie das Paket bezogen haben.

Nach all diesen grundlegenden Vorbereitungen geht es nun endlich an die eigentliche Swift-Binary. Entpacken Sie zunächst das heruntergeladene Paket mit dem folgenden Terminal-Befehl:

```
tar xzf swift-<VERSION>-<PLATTFORM>.tar.gz
```

Auch hier gilt es, <VERSION> und <PLATTFORM> durch die passenden Informationen aus dem heruntergeladenen Paket zu ersetzen, beispielsweise wie folgt:

```
tar xzf swift-5.0-ubuntu18.04.tar.gz
```

Zu guter Letzt sollten Sie noch die Nutzung von Swift als PATH setzen, um anschließend die Nutzung von Swift einfach mittels des Terminal-Befehls `swift` starten zu können. Dazu existiert innerhalb des eben entpackten Archivs ein Ordner *usr* und darin wiederum ein Ordner *bin*. Den Pfad zu diesem Ordner müssen Sie exportieren und der PATH-Variablen hinzufügen, um anschließend wie beschrieben Swift mittels des Befehls `swift` im Terminal ausführen und starten zu können. Im Folgenden sehen Sie den entsprechenden Befehl, mit dem Sie dieses gewünschte Verhalten umsetzen:

```
export PATH=<PFAD ZU /USR/BIN>:"${PATH}"
```

<PFAD ZU /USR/BIN> bezieht sich auf den genannten Verweis des entpackten Swift-Archivs und dessen *bin*-Ordner.

Wenn Sie nun im Terminal den Befehl `swift` eingeben, startet die REPL und Sie können damit beginnen, Swift-Code zu schreiben und auszuführen (siehe Bild 1.8).

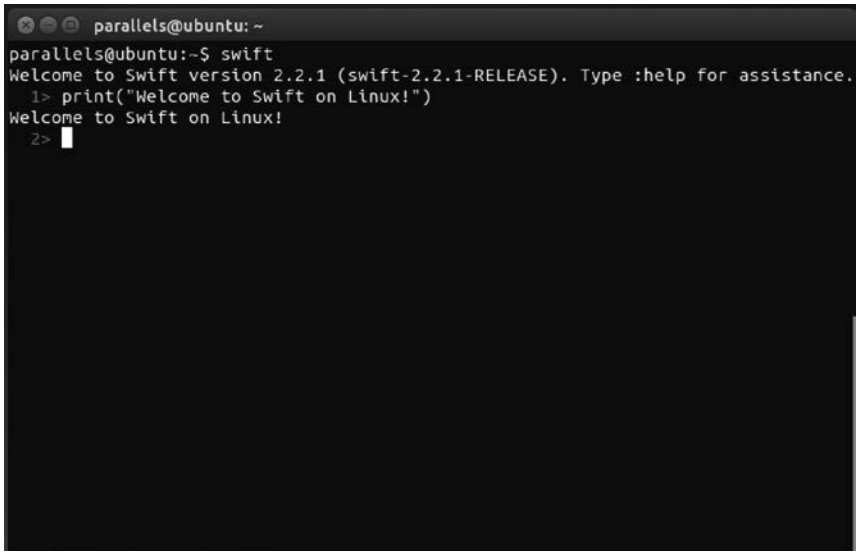


Bild 1.8 Nach der erfolgreichen Installation kann Swift mittels des Befehls „swift“ unter Linux im Terminal ausgeführt werden.

1.5 Playgrounds

Bei Playgrounds handelt es sich um ein spezielles Dateiformat zum Schreiben und Ausführen von Swift-Code. Aktuell lassen sich Playgrounds sowohl mit Xcode als auch mit der iPad-App *Swift Playgrounds* erstellen und verwenden (siehe Bild 1.9).

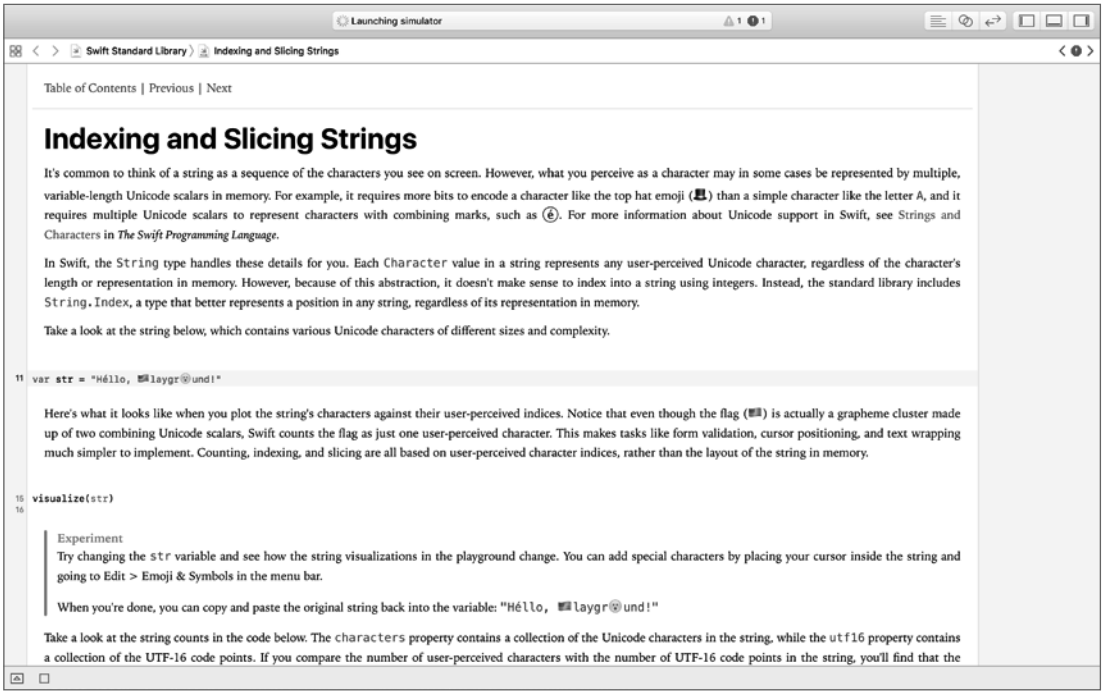


Bild 1.9 Playgrounds erlauben das Ausprobieren und Dokumentieren von Swift-Code.

Wie der Name bereits andeutet, sind Playgrounds auf das Ausprobieren von und Experimentieren mit Swift ausgelegt. Dabei wird der in den Playgrounds eingegebene Code umgehend kompiliert und das daraus entstehende Ergebnis angezeigt. Auch bringen Playgrounds sehr gute Möglichkeiten zur Dokumentation mit. So lassen sich schnell und leicht Überschriften verschiedener Ebenen formatieren und Aufzählungen umsetzen, die parallel zum geschriebenen Code angezeigt werden.

Mit diesen Eigenschaften eignen sich Playgrounds auch ideal zum Einstieg in die Swift-Entwicklung und zum Ausprobieren eigener Ideen. Auch beim Durcharbeiten dieses Buches sind Playgrounds ein geeignetes Mittel, um die Inhalte der einzelnen Kapitel und Abschnitte möglichst schnell praktisch anzuwenden und mit ihnen zu experimentieren. Darum stelle ich in diesem Abschnitt Playgrounds einmal mit ihrer grundlegenden Funktionalität vor. Dabei beschränke ich mich auf die Arbeit mit Playgrounds in Xcode, da diese umfangreicher und ausgereifter sind und deutlich mehr Möglichkeiten bieten, als das bei der Swift Playgrounds-App für das iPad der Fall ist.

1.5.1 Erstellen eines Playgrounds

Ein neuer Playground kann entweder direkt aus dem Startfenster von Xcode heraus ober über das Menü über *File → New → Playground …* erstellt werden (siehe Bild 1.10). Anschließend öffnet sich ein neues Fenster, in dem Sie den gewünschten Namen für den neu zu erstellenden Playground festlegen sowie aus dem Drop-down-Menü *Platform* eine der verfügbaren Plattformen auswählen (siehe Bild 1.11). Durch letztere Auswahl werden bereits erste Frameworks importiert, die für die Entwicklung von Apps für die jeweilige Plattform essenziell sind (was aber nicht bedeutet, dass Sie diese Frameworks auch benutzen müssen; Sie können auch stattdessen ganz grundlegenden Swift-Code schreiben und ausführen, der unabhängig von der gewählten Plattform ist, wodurch die Auswahl in diesem Fall letzten Endes irrelevant ist).

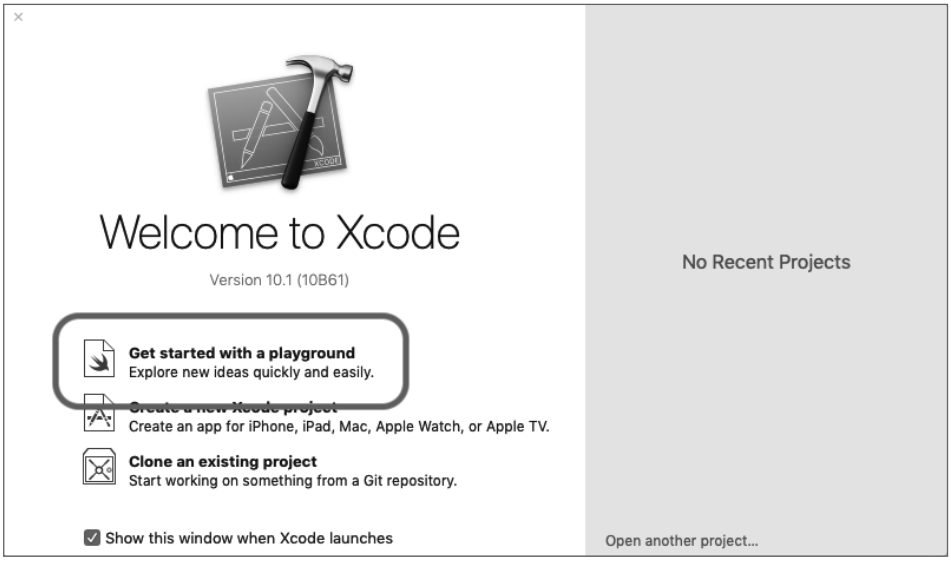


Bild 1.10 Ein neuer Playground kann direkt aus dem Startfenster von Xcode heraus erstellt werden.

Nach einem anschließenden Klick auf *Next* fragt Xcode nach dem Speicherort der neuen Playground-Datei, ein weiterer Klick auf *Create* erstellt diese und öffnet sie direkt (siehe Bild 1.12).

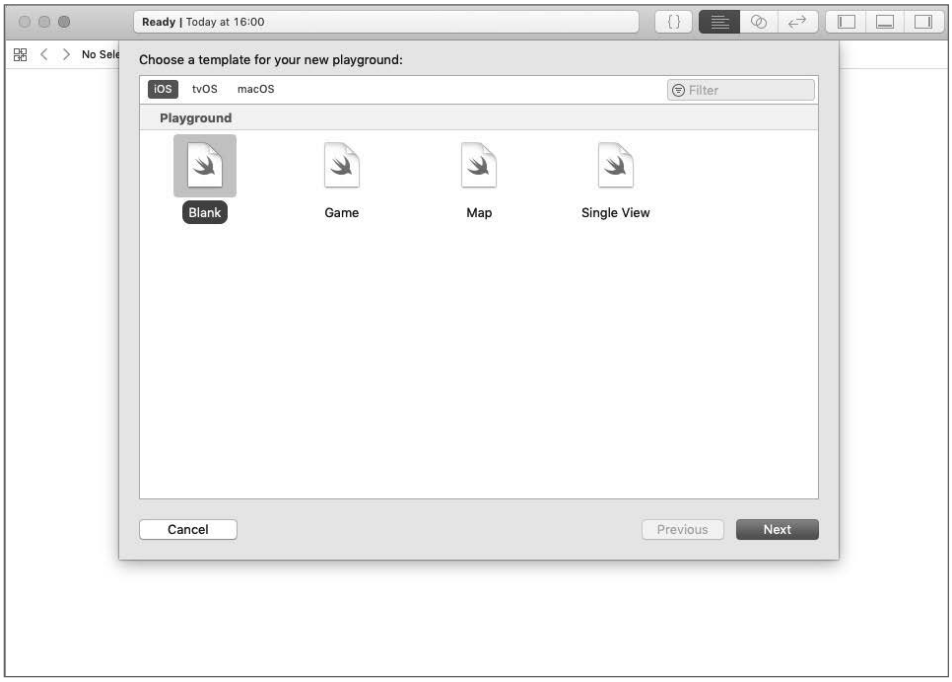


Bild 1.11 Es reicht die Eingabe eines Namens sowie die Auswahl einer Plattform, um einen neuen Playground in Xcode zu erstellen.

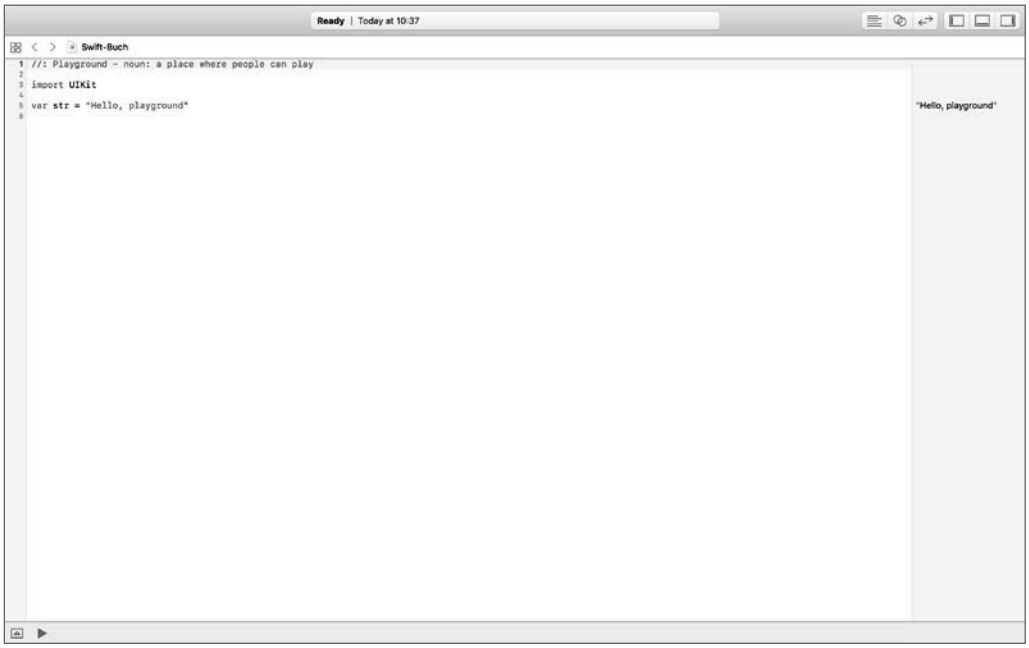


Bild 1.12 Ein neu erstellter Playground in Xcode

1.5.2 Aufbau eines Playgrounds

Der grundlegende Aufbau eines Playgrounds ist zu vergleichen mit dem eines Xcode-Projekts, gestaltet sich aber ein wenig kompakter, schlanker und übersichtlicher. Im Folgenden stelle ich Ihnen alle Elemente des Playground-Fensters im Detail vor.

Editor

Die Standardansicht, die auch zu sehen ist, wenn ein neuer Playground erstellt wurde, zeigt lediglich das Editor-Fenster an, das sich bei Playgrounds in zwei Bereiche teilt. Den größten Raum nimmt die weiße Fläche des Editors ein, in dem der eigentliche Swift-Code geschrieben wird. Daneben befindet sich auf der rechten Seite ein leicht gräulicher Bereich, der bei der Arbeit mit Xcode exklusiv in Playgrounds zur Verfügung steht und die Playgrounds auch so besonders macht. Dort nämlich werden direkt Informationen und Ergebnisse zu dem geschriebenen Code aufgeführt, sobald dieser kompiliert wird. Sie sehen so beispielsweise, wie oft eine Schleife durchlaufen wurde oder welchen Wert eine Variable besitzt. In dem Standardcode, der Teil eines jeden neu erstellten Playgrounds ist, ist das bereits sehr gut zu sehen: Darin wird eine Variable `str` erstellt und ihr der String `"Hello, playground"` zugewiesen. Genau dieser String – der Inhalt der Variablen `str` zu diesem Zeitpunkt – wird nach dem Ausführen des Codes direkt in eben dem grauen Bereich auf der rechten Seite angezeigt (siehe Bild 1.13). Um Code auszuführen, klicken Sie einfach auf die passende Play-Schaltflächeam linken Rand. Alle Befehle bis zur aktuellen Zeile werden anschließend ausgeführt. Das erlaubt es Ihnen auch, einen Playground Zeile für Zeile oder Abschnitt für Abschnitt ausführen zu lassen.

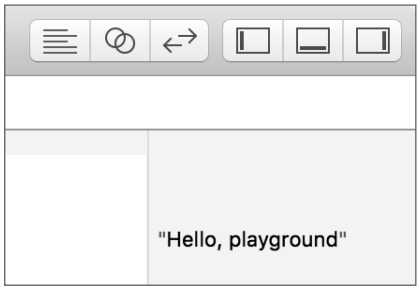


Bild 1.13
Im rechten Bereich des Playground-Editors werden Ausgaben und Informationen zum eingegebenen Swift-Code angezeigt.

Wenn Sie nun mit der Maus über solch einen Output im grauen Bereich des Editors fahren, erscheinen zwei Schaltflächen am rechten Rand neben dem jeweiligen Eintrag (siehe Bild 1.14). Bei der linken der beiden handelt es sich um die sogenannte *Quick Look*-Ansicht. Ein Klick darauf öffnet eine Vorschau für das jeweilige Element. Im Falle des Strings zeigt Quick Look einfach noch einmal den reinen Text (ohne Anführungszeichen) an. Quick Look kann aber noch mehr. Es kann beispielsweise konkrete Informationen zu Farben zurückliefern oder das einer Variablen zugewiesene Bild direkt anzeigen (siehe Bild 1.15). Damit liefert Quick Look bisweilen mehr Informationen zu einem Element als das, was im rechten grauen Bereich des Editors angezeigt wird. Welche Informationen in welcher Form zu einem Element mittels Quick Look angezeigt und ausgegeben werden, ist von Typ zu Typ unterschiedlich.

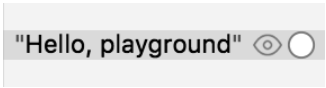


Bild 1.14 Bei einem Mouseover über ein Element im rechten grauen Editorbereich erscheinen zwei zusätzliche Schaltflächen.

Bild 1.15
Ein Klick auf die Quick Look-Schaltfläche liefert eine grafische Vorschau zum zugehörigen Element.

Neben der Quick Look-Schaltfläche befindet sich ein weiterer Button mit dem Titel *Show Result*. Dieser bindet die eben vorgestellte Quick Look-Ansicht direkt in den Quelltext des Playgrounds unterhalb des zugehörigen Elements ein (siehe Bild 1.16), ein weiterer Klick auf die Schaltfläche entfernt die Quick Look-Ansicht wieder. Der große Vorteil dieses direkten Einbindens von Quick Look liegt darin, dass die Quick Look-Ansicht selbst nun auch bei jedem Kompilieren des Playgrounds aktualisiert wird. Somit sehen Sie innerhalb des Playgrounds sofort, inwieweit sich ein bestimmtes Element verändert hat und welchen Wert es besitzt.

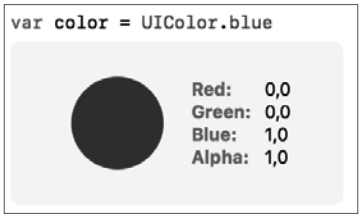


Bild 1.16
Die Quick Look-Vorschau kann direkt unterhalb des zugehörigen Elements in den Quellcode des Playgrounds eingebunden werden.

Konsole und Codeausführung

Am unteren linken Bildschirmrand des Editors finden sich zwei Schaltflächen. Mit der linken lässt sich die Konsole von Xcode ein- und ausblenden. Sie wird immer dann automatisch eingeblendet, sobald der vom Playground ausgeführte Code etwas in der Konsole ausgibt. Daneben befindet sich ein Run-Button zum Ausführen des Playground-Codes. Wird er betätigt, wird der gesamte Code des Playgrounds kompiliert. Standardmäßig geschieht das nach einem Klick auf den Play-Button am linken Rand einer Zeile, Sie können dieses Verhalten aber ändern, in dem Sie längere Zeit die Run-Schaltfläche gedrückt halten. Dann öffnet sich ein Pop-up, in dem Sie zwischen den Optionen *Automatically Run* und *Manually Run* wählen können (standardmäßig ist *Manually Run* aktiv, siehe Bild 1.17). Wenn Sie hier *Manually Run* auswählen, dann wird der Code des Playgrounds nur dann ausgeführt, wenn Sie explizit den Play-Button per einfachem Klick betätigen. Besonders auf leistungsschwächeren Rechnern mag diese Option sinnvoll sein, da das schier ununterbrochene Ausführen des Codes durchaus einige Ressourcen des Mac beanspruchen kann.

Bild 1.17
Sie können das manuelle Ausführen eines Playgrounds auf automatisch umstellen, sodass jede Änderung umgehend kompiliert wird.

Toolbar

Am oberen befindet sich die Toolbar. Diese informiert über eine Statusleiste in der Mitte über die aktuell von Xcode ausgeführten Aktionen wie beispielsweise den Start eines Simulators im Hintergrund, das Ausführen des Playground-Codes oder das Herunterladen zusätzlicher Inhalte. Auch werden Ihnen dort Warnungen und Fehler angezeigt, sollte es im Code Probleme geben (siehe Bild 1.18).

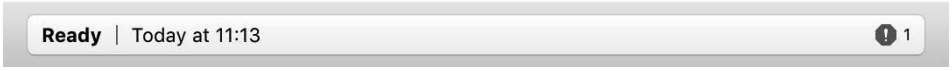


Bild 1.18 Die Statusbar informiert über aktuelle Xcode-Aktivitäten und macht auf etwaige Fehler und Probleme im Code aufmerksam.

Am rechten Rand der Toolbar finden sich noch Schaltflächen zum Anpassen des Editors (erste Dreierreihe) sowie zum Ein- und Ausblenden einzelner Bereiche (letzte Dreierreihe, siehe Bild 1.19).

Bild 1.19 Die Playground-Ansicht kann über die Schaltflächen am oberen rechten Bildschirmrand angepasst werden.

So können Sie mithilfe der letzten drei Schaltflächen eine Übersicht der Dateien des Playgrounds, die bereits bekannte Konsole sowie die Inspectors einblenden (siehe Bild 1.20).

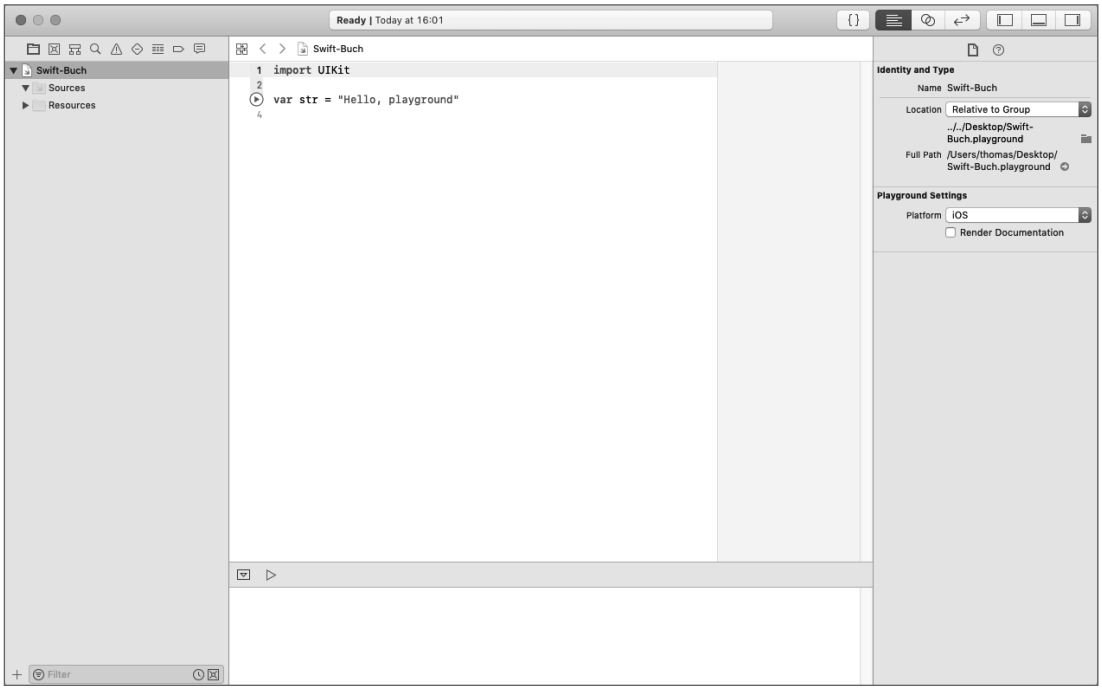


Bild 1.20 Das Playground-Fenster kann um insgesamt drei Bereiche ergänzt werden.

Navigator

Die Navigator Area lässt sich mithilfe der eben gezeigten Schaltflächen am linken Bildschirmrand des Playgrounds ein- und ausblenden. Sie gewährt Zugriff auf die Dateien des Playgrounds, informiert im Detail über Fehler und Warnungen im Code und erlaubt das Durchsuchen des Quelltexts. Dazu verfügt die Navigator Area am oberen Rand über fünf Schaltflächen, über die in die verschiedenen Abschnitte gewechselt werden kann (siehe Bild 1.21).

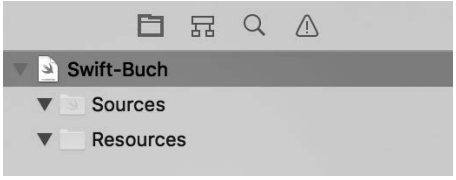


Bild 1.21
Über die Navigator Area haben Sie Zugriff auf die Dateien eines Playgrounds.

Mehr zum Erstellen und Hinzufügen von Dateien zu einem Playground erfahren Sie in Abschnitt 1.5.3, „Pages, Sources und Resources“.

Inspectors

Sie können die Inspectors über die entsprechende Schaltfläche aus der Toolbar dynamisch ein- und ausblenden. Sie liefert Informationen über die eigentliche Playground-Datei und bietet einen Schnellzugriff auf die Dokumentation von Apple nach Auswahl eines bestimmten Typs oder einer Funktion im Quellcode (siehe Bild 1.22).

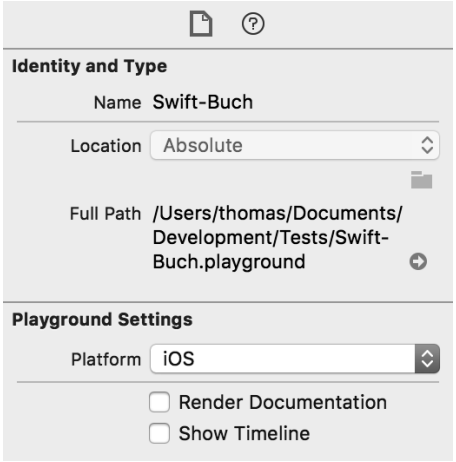


Bild 1.22
Die Inspectors liefern Informationen zum Playground und einen Schnellzugriff auf die Dokumentation von Apple.

1.5.3 Pages, Sources und Resources

Ein Playground setzt sich immer aus einer sogenannten *Page* zusammen. Das ist eine spezielle Datei, die von Xcode gelesen und bearbeitet werden kann und die direkte Ausgaben des eigenen Codes im rechten Bereich des Editors erlaubt (wie wir im vorherigen Abschnitt 1.5.2, „Aufbau eines Playgrounds“, gesehen haben). Jede Page verfügt über zwei Unterordner, um weitere Dateien zu verwalten:

- Sources
- Resources

Dem Sources-Ordner können reine Quellcode-Dateien hinzugefügt werden, die für den Playground benötigt werden. Typischerweise handelt es sich dabei um einfache Swift-Dateien mit der Dateiendung *.swift*. Beispielsweise haben Sie bereits zuvor an einer anderen Stelle Code geschrieben, den Sie nun für eine bestimmte Funktion oder für Testzwecke in einem Playground verwenden möchten. Dann können Sie – anstatt den gesamten entsprechenden Code in die Page des Playgrounds zu kopieren – die entsprechenden Swift-Dateien einfach als Quellcode-Dateien dem Sources-Ordner der Playground-Page hinzufügen.

Selbiges gilt für *Resources*, wobei dieser Ordner – wie der Name bereits andeutet – für Dateien wie Bilder oder Videos gedacht ist. Sollten Sie also für Ihren Playground zusätzliche Dateien abseits des reinen Quellcodes benötigen, können Sie diese in den Resources-Ordner der zugehörigen Page einbinden.

Um Zugriff auf die beiden genannten Ordner zu erhalten, müssen Sie zunächst über die zugehörige Schaltfläche in der Toolbar die Navigator Area einblenden. Dann sehen Sie innerhalb der Navigator Area bereits die Page des Playgrounds (diese wird an oberster Stelle angezeigt und trägt den gleichen Titel wie der Playground) sowie die beiden Unterordner *Sources* und *Resources* (siehe Bild 1.23). Sollten Sie letztere nicht sehen, müssen Sie sehr wahrscheinlich die Page mittels der vorangestellten Pfeilschaltfläche „aufklappen“, damit die Unterordner angezeigt werden.

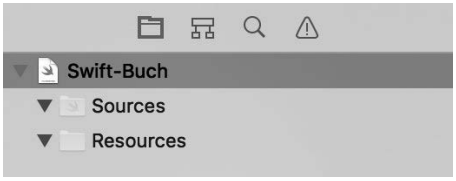


Bild 1.23
Die Navigator Area zeigt die zugrunde liegende Page mitsamt Unterordnern des Playgrounds an.

Möchten Sie nun ein Source-File oder eine Resource zu einer Playground-Page hinzufügen, wählen Sie zuvor den gewünschten Ordner, dem Sie eine Datei hinzufügen möchten, per Linksklick aus und klicken anschließend auf die Plus-Schaltfläche am unteren linken Rand der Navigator Area. Dort erscheint dann ein Pop-up-Menü mit verschiedenen Auswahloptionen (siehe Bild 1.24).

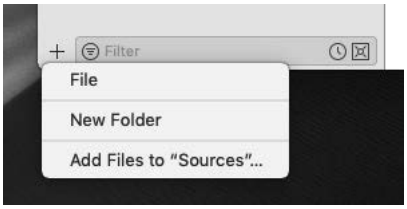


Bild 1.24
Über die Plus-Schaltfläche am unteren linken Rand der Navigator Area können einem Playground neue Dateien hinzugefügt werden.

Der erste Punkt *File* fügt dem gewählten Ordner eine neue leere Datei hinzu, darüber hinaus können Sie auch bereits vorhandene Dateien mittels der letzten Schaltfläche *Add Files to "<NAME DES Ordners>"…* von Ihrem Mac auswählen und so zu Ihrem Playground hinzufügen. Um einen Unterordner zu erstellen, wählen Sie *New Folder*.

Ebenso haben Sie die Möglichkeit, Ihren Playground um weitere Pages zu ergänzen. Hierfür wählen Sie eine bereits vorhandene Page aus (zum Beispiel den obersten Eintrag mit dem Namen Ihres Playgrounds) und klicken anschließend auf die Plus-Schaltfläche unten links. Ihnen wird dann eine Aktion mit dem Titel *New Playground Page* angeboten. Darüber erstellt Xcode beim erstmaligen Betätigen der Schaltfläche zwei neue Pages unterhalb der obersten Page und verschiebt dabei den Inhalt dieser obersten Page in eine der beiden neu erstellten (siehe Bild 1.25). Damit wird diese oberste Page zu einer Art übergeordnetem Element, das selbst direkt keinen Inhalt mehr besitzt, sondern diesen stattdessen über seine Unterseiten verwaltet. Anschließend können Sie die neu erstellten Pages noch umbenennen, indem Sie sie erst einmal mittels Linksklick selektieren, kurz warten, und anschließend erneut mit der linken Maustaste anklicken.

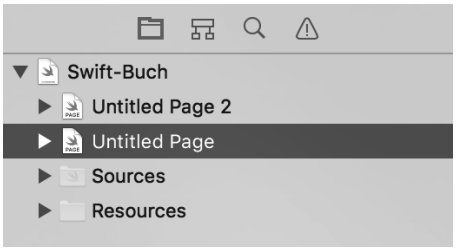


Bild 1.25
Beim erstmaligen Hinzufügen einer neuen Page werden direkt zwei Pages unterhalb des obersten Elements erstellt.

Sie können mit den beiden neu erstellten Pages nun genauso arbeiten wie zuvor mit der eigentlichen Playground-Page und können darin alle bekannten Vorteile von Playgrounds nutzen. Mehrere Pages sind vor allen Dingen dann sinnvoll, wenn Sie sehr viel Code in Ihrem Playground schreiben, um diesen besser zu strukturieren und den Playground selbst ein wenig übersichtlicher zu gestalten. Dabei können Sie Ihren Playground auch auf die gezeigte Art und Weise um weitere Pages ergänzen und so ganz Ihren Bedürfnissen anpassen.

Darüber hinaus besitzt aber auch jede neu erzeugte Page ebenfalls für sich genommen noch einmal die beiden genannten Unterordner *Sources* und *Resources*. Diese dienen dazu, bestimmte Dateien dort einzufügen, die nur im Zusammenhang mit der zugehörigen Page stehen. Falls Sie beispielsweise ein Bild nur explizit für eine ganz bestimmte Page in Ihrem Playground nutzen möchten, dann macht es Sinn, dieses Bild dem Resources-Ordner eben dieser Page hinzuzufügen und nicht der Page auf der obersten Ebene; die dort hinzugefügten Dateien stehen wiederum *allen* Pages Ihres Playgrounds zur Verfügung.

1.5.4 Playground-Formatierungen

Xcode stellt eine Reihe verschiedener Formatierungen für Playgrounds bereit, mit denen diese beispielsweise durch Überschriften und Aufzählungen optisch optimiert werden können. Der Playground kann dann dynamisch zwischen reiner Codeansicht und einer aufbereiteten formatierten Ansicht umgeschaltet werden. Letztere bietet sich dann beim Prüfen des Inhalts eines Playgrounds an, da dort die Texte, die für das Layout formatiert wurden, zwar nicht geändert werden können, dafür aber dank ihrer Formatierung eine bessere Lesbarkeit und Übersichtlichkeit bieten.

In diesem Abschnitt möchte ich Ihnen eine Auswahl der verfügbaren Formatierungen mitsamt ihrer zugehörigen Syntax vorstellen. Wichtig dabei ist, dass alle gezeigten Formatierungen innerhalb eines Kommentarblocks im Code des Playgrounds erfolgen. Es gibt zwei Möglichkeiten, derartige Kommentare in Playgrounds umzusetzen. Für einen Kommentar, der sich lediglich über eine einzige Zeile erstreckt, muss diese mit einem `//:` eingeleitet werden:

```
//: <PLAYGROUND-FORMATIERUNG>
```

Sollen hingegen mehrere Zeilen auf einmal formatiert werden, so wird ein solcher Kommentarblock mittels `/*:` eingeleitet und mit `*/` wieder abgeschlossen. In den Zeilen dazwischen folgen dann die gewünschten Formatierungen. So ergibt sich der in Listing 1.1 gezeigte Aufbau.

Listing 1.1 Formatierung eines mehrzeiligen Playgrounds

```
/*:
 <ERSTE PLAYGROUND-FORMATIERUNG>
 <ZWEITE PLAYGROUND-FORMATIERUNG>
 <DRITTE PLAYGROUND-FORMATIERUNG>
*/
```

Um zwischen den beiden Ansichten – der standardmäßigen reinen Codeansicht, in der auch die gezeigten Kommentare geschrieben werden, und der formatierten Ansicht – zu wechseln, gibt es zwei Möglichkeiten. Einerseits können Sie über das Xcode-Menü gehen und dort *Editor → Show Rendered Markup* (zum Wechseln in die Formatierungsansicht) beziehungsweise *Editor → Show Raw Markup* (zum Wechseln in die Codeansicht) auswählen (siehe Bild 1.26) oder Sie nutzen die Utilities Area. Dort findet sich in der File Inspector-Ansicht (diese erreichen Sie über die Schaltfläche ganz links am oberen Rand der Utilties Area) eine Checkbox mit dem Titel *Render Documentation* (siehe Bild 1.27). Ist diese Checkbox aktiviert, wird die Formatierungsansicht für den Playground aktiviert, andernfalls die Codeansicht.

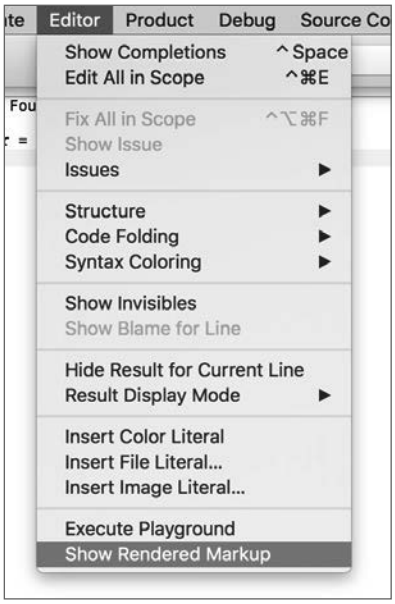


Bild 1.26
Über das Xcode-Menü kann zwischen Formatierungs- und Codeansicht eines Playgrounds gewechselt werden.

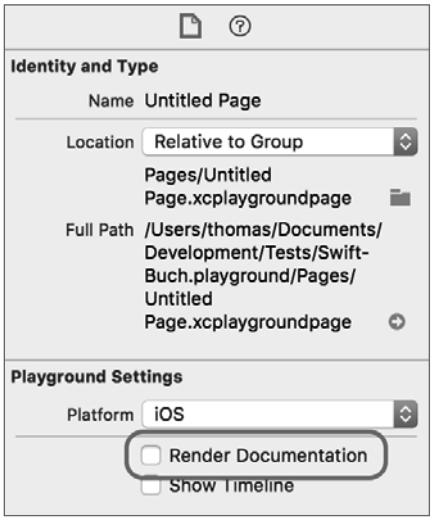


Bild 1.27
Die Checkbox „Render Documentation“ der Utilities Area erlaubt das schnelle Wechseln zwischen Formatierungs- und Codeansicht eines Playgrounds.

Überschriften

Überschriften können in Playgrounds bis zu einer dritten Ebene reichen. Sie werden mithilfe des #-Zeichens erstellt. Ein #-Zeichen steht für eine Überschrift der ersten Ebene, zwei #Zeichen stehen für eine Überschrift der zweiten und drei #-Zeichen für eine Überschrift der dritten Ebene (siehe Listing 1.2).

Listing 1.2 Formatierung von Überschriften in einem Playground

```
/*:
 # Header 1
 ## Header 2
 ### Header 3
*/
```

Ein so formatierter Playground sieht dann aus wie in Bild 1.28 zu sehen.

Bild 1.28
Formatierte Überschriften eines Playgrounds.

Trennlinie

Eine Trennlinie in einem Kommentar wird mithilfe dreier aufeinanderfolgender Bindestriche (`---`) umgesetzt, so wie beispielhaft in Listing 1.3 zu sehen.

Listing 1.3 Formatierung einer Trennlinie in einem Playground

```
/*:
 Ein Text gefolgt von einer Trennlinie...

 ---

 ...und darauffolgendem weiteren Text.
*/
```

Formatiert sieht ein solcher Kommentar in einem Playground aus wie in Bild 1.29 zu sehen.

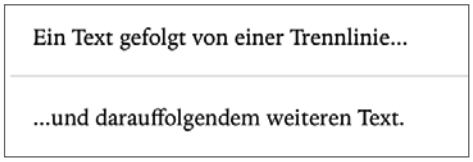


Bild 1.29 Formatierter Kommentar eines Playgrounds mit Trennlinie.

Listen

Es können sowohl einfache Aufzählungen als auch nummerierte Listen mithilfe einer passenden Playground-Formatierung in einem Kommentar umgesetzt werden.

Für eine einfache Aufzählung gibt es dabei insgesamt drei Möglichkeiten: Sie können entweder einen Stern (*), ein Plus (+) oder einen Bindestrich (-), jeweils gefolgt von einem Leerzeichen, verwenden, um darüber nacheinander in aufeinanderfolgenden Zeilen die gewünschte Aufzählung umzusetzen. Listing 1.4 zeigt ein Beispiel dazu.

Listing 1.4 Eine formatierte Aufzählung in einem Playground

```
/*:
 # Einfache Aufzählung
 - Erstes Element
 - Zweites Element
 - Drittes Element
 */
```

Welches der drei möglichen Elemente Sie für die Aufzählung verwenden, spielt keine Rolle; es wird unabhängig davon immer das gleiche Aufzählungszeichen formatiert (siehe Bild 1.30).

Bild 1.30 Eine einfache Aufzählung in einem Playground.

Sie können die drei Zeichen aber auch innerhalb einer Aufzählung mischen, um zwischen den Elementen einen zusätzlichen Abstand herzustellen. Der Code in Listing 1.5 führt dann zu dem in Bild 1.31 gezeigten Ergebnis.

Listing 1.5 Mehrere entkoppelte Aufzählungen in einem Playground

```
/*:
 # Mehrere einfache Aufzählungen
 - Erstes Element, erste Aufzählung
 - Zweites Element, erste Aufzählung
 * Erstes Element, zweite Aufzählung
 + Erstes Element, dritte Aufzählung
 + Zweites Element, dritte Aufzählung
 + Drittes Element, dritte Aufzählung
 */
```

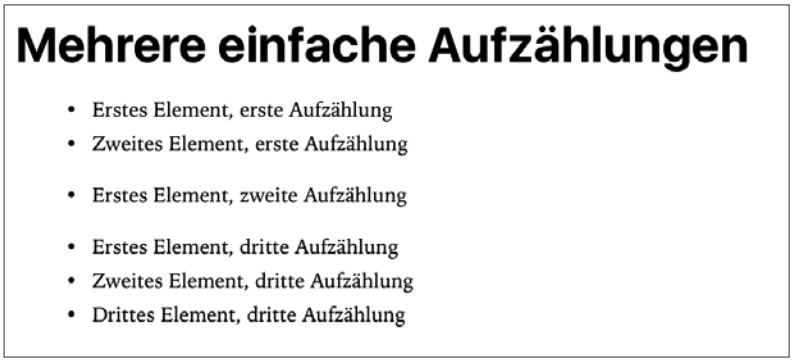


Bild 1.31 Durch parallele Verwendung der drei möglichen Aufzählungsformatierungen lassen sich bis zu drei Gruppen von Aufzählungen auf einmal erstellen.

Um eine nummerierte Liste zu erstellen, gehen Sie ähnlich vor wie bei den eben gezeigten einfachen Aufzählungen. Beginnen Sie dabei mit der Nummer, mit der die Aufzählung beginnen soll, gefolgt von einem Punkt und einem Leerzeichen, anschließend folgt der Text für das zugehörige Aufzählungselement. So fahren Sie anschließend in den darauffolgenden Zeilen für alle weiteren Elemente der nummerierten Liste fort. Listing 1.6 zeigt ein Beispiel dazu, Bild 1.32 präsentiert das zugehörige Ergebnis.

Listing 1.6 Nummerierte Liste in einem Playground

```
/*:
 # Nummerierte Liste
 1. Erstes Element
 2. Zweites Element
 3. Drittes Element
 */
```

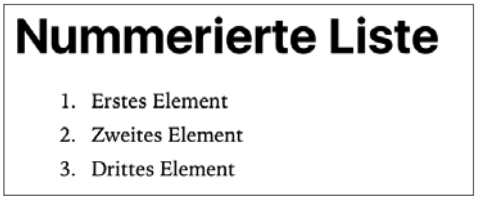


Bild 1.32 Eine nummerierte Liste in einem Playground.

Nummerierte Listen unterstützen in Playgrounds bis zu drei Einrückungsebenen, um so eine Liste zu verschachteln. Elementen der zweiten Ebene wird dafür ein Tab vorangestellt, Elementen der dritten Ebene zwei Tabs. Listing 1.7 zeigt ein Beispiel dazu, in Bild 1.33 sehen Sie das zugehörige Ergebnis.

Listing 1.7 Nummerierte Liste mit mehreren Ebenen in einem Playground

```
/*:
 # Nummerierte Liste mit Ebenen
 1. Erstes Element
 2. Zweites Element
    1. Erstes Unterelement
    2. Zweites Unterelement
        1. Erstes Unterunterelement
    3. Drittes Unterelement
 3. Drittes Element
 */
```

Bild 1.33 Eine nummerierte Liste mit mehreren Ebenen in einem Playground.

Soll eine nummerierte Liste mit einer anderen Zahl beginnen, setzen Sie diese einfach entsprechend. Alle darauffolgenden Elemente der entsprechenden Ebene werden daraufhin automatisch immer um eins hochgesetzt. Das geht sogar so weit, dass es vollkommen irrelevant ist, welche Zahl Sie ab dem zweiten Element einer Ebene im Code angeben; der Playground wird das rigoros ignorieren und ausgehend vom ersten Element der Ebene automatisch alle darauffolgenden Elemente hochsetzen. In Listing 1.8 sehen Sie ein entsprechendes Beispiel dazu, dessen durchaus überraschendes Ergebnis sehen Sie in Bild 1.34.

Listing 1.8 Nummerierte Liste mit unterschiedlichen Zahlenwerten in einem Playground

```
/*:
 # Nummerierte Liste mit automatischer Zählung
 7. Erstes Element
 2. Zweites Element
    5. Erstes Unterelement
    4. Zweites Unterelement
        3. Erstes Unterunterelement
    3. Drittes Unterelement
 8. Drittes Element
 */
```

Bild 1.34 Obwohl den Punkten „Zweites Element“, „Zweites Unterelement“, „Drittes Unterelement“ und „Drittes Element“ im Code eine andere Nummer zugewiesen wurde, wird diese vom Playground ignoriert und stattdessen automatisch ausgehend von der ersten Nummer der jeweiligen Ebene hochgezählt.

Code

Zur besseren Dokumentation können Sie auch Code in einem Playground-Kommentar formatieren. Dazu müssen Sie lediglich die gewünschte Zeile, in der Sie eine solche Code-Formatierung anwenden möchten, mit einem Tab einleiten. Anschließend können Sie den gewünschten Code schreiben (siehe Listing 1.9). Beachten Sie dabei auch, dass die Zeile *vor* dem Code-Beispiel leer sein muss, damit die Formatierung korrekt angewendet wird. Wie das Ganze dann formatiert aussehen kann, sehen Sie in Bild 1.35.

Listing 1.9 Formatierter Code in einem Playground

```
/*:
 # Code
 Hier ein Code-Beispiel:

    print("Ein Code-Kommentar")
 */
```

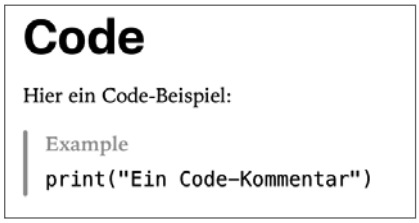


Bild 1.35
Formatierung von Code in einem Playground.

Anstatt ganze Zeilen als Code zu formatieren, können Sie aber auch in einem einfachen Kommentar einzelne Wörter oder Teile davon entsprechend hervorheben. Setzen Sie die entsprechenden Elemente dafür innerhalb von einfachen Anführungszeichen (`, siehe Listing 1.10 und Bild 1.36).

Listing 1.10 Formatierter Code im Fließtext eines Playgrounds

```
/*:
 # Code Inline
 Hier ein Code-Beispiel mit `print`:

    print("Ein Code-Kommentar")
 */
```

Bild 1.36 Code lässt sich auch innerhalb von Fließtext in einem Playground-Kommentar passend formatieren.

Kursiv und fett

Texte in Playground-Kommentaren können kursiv und fett formatiert werden. Dazu wird der entsprechende Abschnitt im Falle von kursiv entweder zwischen je einem Stern (*) oder einem Unterstrich (_) eingefügt, während es bei fett jeweils zwei Sterne (**) oder zwei Unterstriche (__) sind. In Listing 1.11 und Bild 1.37 sehen Sie dazu ein passendes Beispiel.

Listing 1.11 Kursiv und fett formatierte Textstellen in einem Playground

```
/*:
 # Kursiv und Fett
 In diesem Satz ist etwas *kursiv* und etwas **fett**.
 */
```

Bild 1.37 Textstellen lassen sich in Playground-Kommentaren sowohl kursiv als auch fett formatieren.

Links

Um Links in einem Playground-Kommentar einzufügen, geben Sie zunächst innerhalb von eckigen Klammern den Text ein, der für den Link angezeigt werden soll, gefolgt vom Link innerhalb von runden Klammern. In Listing 1.12 sehen Sie ein Beispiel dazu, das zugehörige Ergebnis zeigt Bild 1.38.

Listing 1.12 Link in einem Playground

```
/*:
 # Link
 Hier geht's zu [Swift.org](https://swift.org)
 */
```

Bild 1.38
Verlinkungen können ebenfalls in einem Playground-Kommentar gesetzt und passend hervorgehoben werden.

Verlinkungen zwischen Pages

Da Playgrounds sich durchaus auch aus mehreren verschiedenen Pages zusammensetzen können (siehe dazu auch den Abschnitt 1.5.3, „Pages, Sources und Resources“), ist es an manchen Stellen womöglich sinnvoll, zwischen den verschiedenen Pages selbst zu verlinken, damit der Nutzer eines Playgrounds direkt von einer Page zu einer anderen zugehörigen Page springen kann, ohne diese selbst in der Navigator Area suchen und aufrufen zu müssen.

Für diese Form der Verlinkungen zwischen Pages gibt es prinzipiell drei Möglichkeiten. Die dynamischste besteht darin, den Namen einer Page als Link zu verwenden. Wählt der Nutzer dann diesen Link aus, springt er zu genau derjenigen Page im Playground, die diesen Namen besitzt. In Listing 1.13 sehen Sie ein Beispiel dazu, wobei davon ausgegangen wird, dass es eine Page mit dem Titel *MoreInformation* im Playground gibt.

Listing 1.13 Verlinkung zu einer anderen Page in einem Playground

```
/*:
 # Page-Verlinkung
 Weitere Informationen finden Sie [hier](MoreInformation)
 */
```

Wie Sie sehen, ist dieses Vorgehen identisch mit dem im vorherigen Abschnitt gezeigten Setzen von Links, nur dass statt einer Webadresse der Name einer Page als Ziel des Links angegeben wird.

Daneben gibt es noch zwei andere Möglichkeiten, zu einer anderen Page zu wechseln. Diese speziellen Verlinkungen erlauben das Springen zur *vorherigen* oder zur *nächsten* Page, ausgehend von der aktuell ausgewählten Page. Das ist ideal, wenn Sie in einem Playground eine hierarchische Navigation umsetzen und immer einen Link von einer Page zur nächsten und wieder zurück anbieten möchten. Dann nämlich brauchen Sie nicht immer explizit die jeweiligen Pages mit ihren zugehörigen Namen (wie in Listing 1.13 zu sehen ist) zu verlinken, sondern Sie können stattdessen Xcode die Arbeit machen lassen. Dazu müssen Sie lediglich innerhalb des Links deklarieren, ob dieser auf die vorherige (`@previous`) oder die kommende (`@next`) Page verweisen soll (siehe Listing 1.14).

Listing 1.14 Verlinkung zur vorherigen und zur nächsten Page in einem Playground

```
/*:
 # Page-Verlinkung

 [Previous Page](@previous)

 ---

 [Next Page](@next)
 */
```

1.5.5 Swift Playgrounds-App für das iPad

Neben Xcode unterstützt auch die mit iOS 10 eingeführte Swift Playgrounds-App von Apple für das iPad die in diesem Abschnitt vorgestellten Playgrounds (siehe Bild 1.39). Was dabei die Programmierung mit Swift betrifft, so sind Ihnen auch in der Swift Playgrounds-App keinerlei Grenzen gesetzt und Sie können alle Eigenschaften und Funktionen der Sprache nutzen und auch auf dem iPad anwenden.

Bild 1.39
Mit der Swift Playgrounds-App können Playgrounds auch auf einem iPad erstellt, bearbeitet und ausgeführt werden.

Allerdings bietet die App nicht einen solch großen und mächtigen Funktionsumfang wie Xcode auf dem Mac. Zwar können Sie neue Playgrounds erstellen und bearbeiten, haben aber keinen schreibenden Zugriff auf die darunterliegende Dateistruktur. Auch das Erstellen von weiteren Pages zu einem Playground ist nicht möglich, ebenso wenig wie der Wechsel in eine Ansicht zur optimierten Darstellung der Kommentare und Formatierungen. Dafür stehen aber Quick Look sowie verschiedene Code-Schnipsel-Vorlagen bereit, die Sie direkt in Ihrem Playground einfügen und nutzen können. Auch das Hinzufügen zusätzlicher Dateien zu einem Playground auf dem iPad ist mittels der Swift Playgrounds-App möglich.

Vielmehr liegt der Schwerpunkt der App im Erlernen und Ausprobieren von Swift. Zu diesem Zweck können nämlich nicht einfach nur eigene Playgrounds erstellt und bearbeitet, sondern bereits vorgefertigte Playgrounds heruntergeladen werden, die sich wie interaktive Lehrbücher verhalten. Sie teilen den Bildschirm zumeist in zwei Bereiche: links Aufgaben und Instruktionen mitsamt einer Fläche zum Schreiben des Codes, rechts eine dynamische Vorschau, die das Ergebnis des geschriebenen Codes ausgibt (siehe Bild 1.40). Diese speziell für die App angepassten und optimierten Playgrounds können Sie über einen entsprechenden Katalog einsehen und auf Wunsch direkt herunterladen. (siehe Bild 1.41).

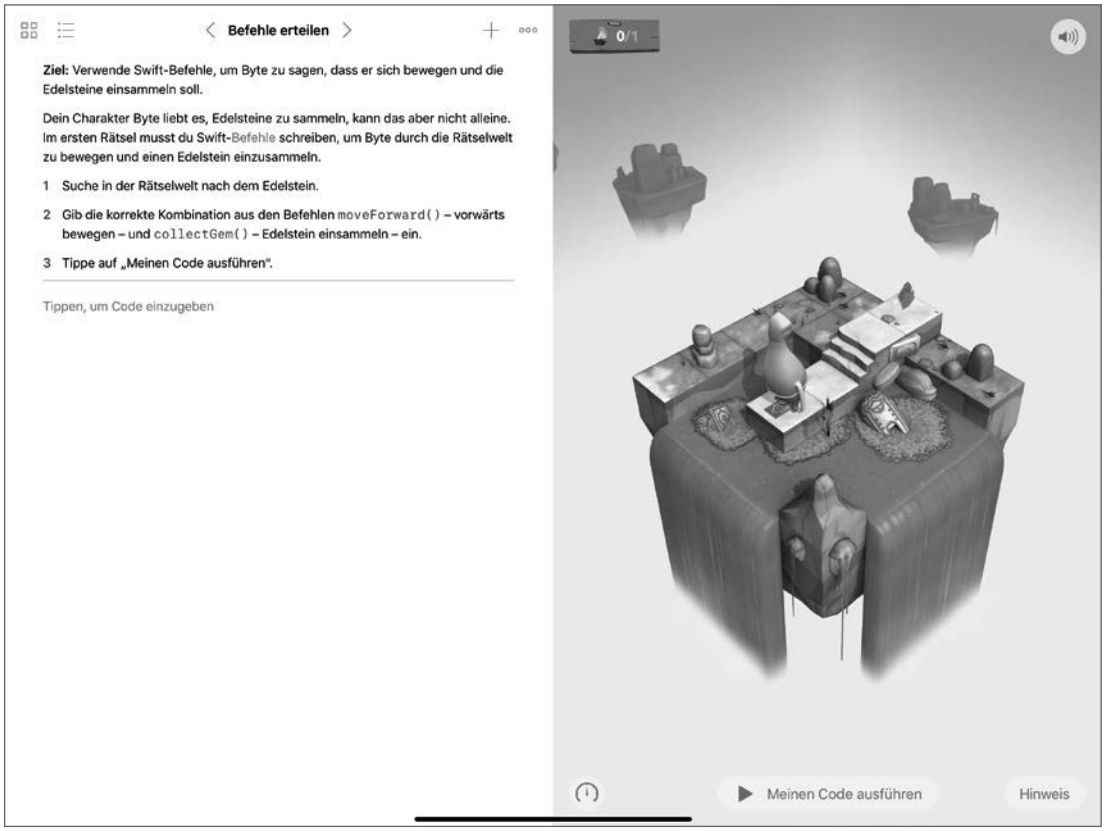


Bild 1.40 Über die Swift Playgrounds-App können angepasste Playgrounds heruntergeladen werden, die spielend das Programmieren mit Swift näherbringen.

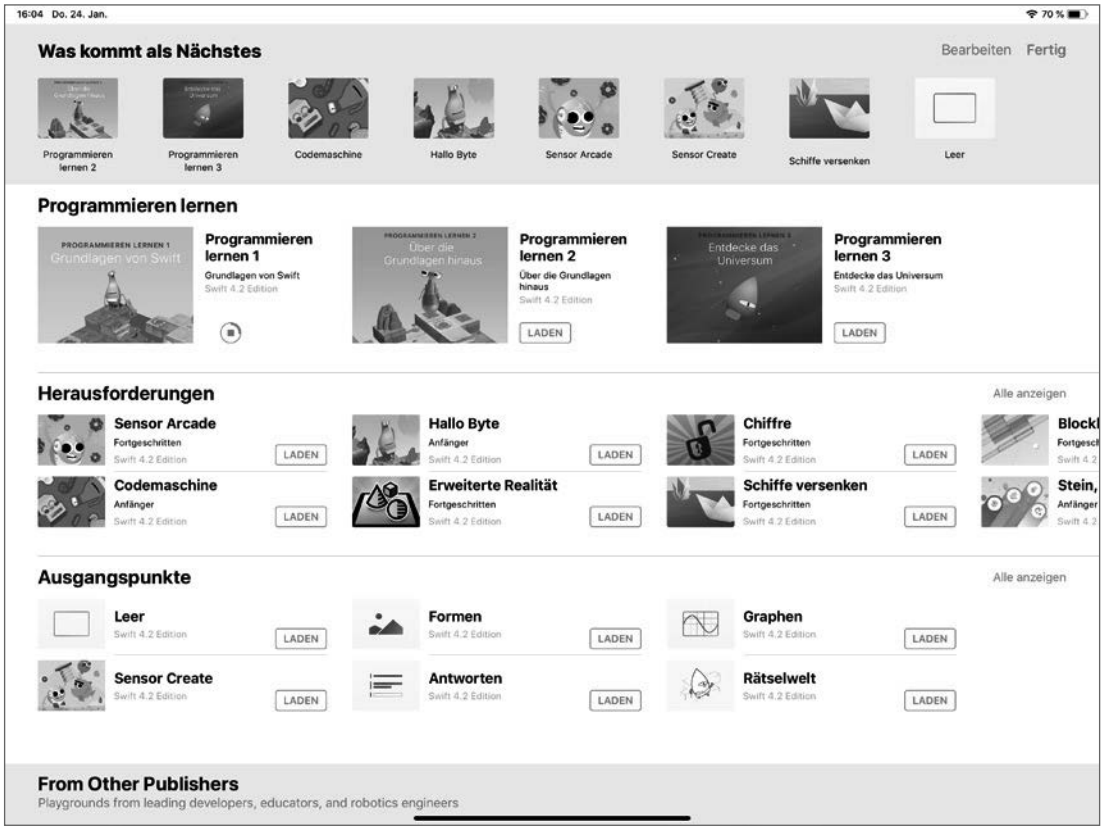


Bild 1.41 Im integrierten Katalog der Swift Playgrounds-App finden sich verschiedene Playgrounds zum Download und zum Experimentieren mit Swift.

Darüber hinaus verfügt die App noch über spezielle Vorlagen für neue Playgrounds. Neben einem einfachen „leeren“ Playground können Sie so auch welche für eigene „Rätselwelten“ sowie zur Arbeit mit Graphen oder Formen erstellen. Diese Vorlagen enthalten bereits vorgefertigten Code, den Sie in Ihrem neuen Playground verwenden und um eigene Logik erweitern können.

23 iOS – Grundlagen

iOS ist Apples mobiles Betriebssystem für iPhone, iPad und iPod touch (siehe Bild 23.1). Die erste Version erschien zusammen mit dem Ur-iPhone im Jahr 2007 und hat sich seit dieser Zeit massiv weiterentwickelt. Features wie iCloud, Siri, Continuity und Handoff eröffnen Apps ganz neue Möglichkeiten und fügen sich perfekt in das bestehende Apple-Ökosystem ein. Darüber hinaus wurde auch die Hardware stets verbessert, wodurch iPhone und iPad stets zu den leistungsstärksten Geräten ihrer jeweiligen Produktkategorie zählen.

Bild 23.1 iOS ist das Betriebssystem für iPhone, iPad und iPod touch.

iPhone und iPad sind heute in mannigfaltigen Szenarien im Einsatz. Neben dem Privatanwender halten iOS-Geräte auch immer stärker Einzug in verschiedene Business-Bereiche und kommen so unter anderem in der Logistik sowie dem Fern- und Nahverkehr zum Einsatz. Dieses starke Wachstum ist neben dem Unternehmen Apple selbst, das diese Produkt-

familie aufmerksam pflegt und stetig weiterentwickelt, all den App-Entwicklern zu verdanken, die mit kreativen und innovativen Lösungen ganz neue Anwendungsbereiche für iPhone und iPad erschlossen haben. Egal ob Schreiben, Malen, Filme machen – der App Store bietet für fast jeden Anwendungszweck eine passende Lösung. Und jeden Tag kommen neue spannende Produkte hinzu.

In diesem Kapitel stelle ich Ihnen das Betriebssystem iOS aus Entwicklersicht im Detail vor und zeige Ihnen, wie iOS-Apps funktionieren, wie sie aufgebaut sind, über welche Bestandteile sie verfügen und wie der Start einer App abläuft. Darüber hinaus lernen Sie alle essenziellen Grundlagen, um Apps für iOS entwickeln zu können. So erfahren Sie beispielsweise, was es mit View-Controllern auf sich hat und wie sie funktionieren, welche User-Interface-Elemente Ihnen von Haus aus bei der App-Entwicklung zur Verfügung stehen und wie Sie eigene Oberflächen für Ihre Apps entwerfen und testen.

In den folgenden Abschnitten erfahren Sie so alle essenziellen Grundlagen, die für die Entwicklung eigener iOS-Apps wichtig sind. Das folgende Kapitel 24, „iOS – App-Entwicklung“, baut anschließend darauf auf und stellt weitere Konzepte und tiefergehende Möglichkeiten in der App-Entwicklung für iOS vor. Wenn Sie bereits erste Erfahrungen in der iOS-Entwicklung gesammelt haben und mit den Grundlagen der Programmierung für diese Plattform vertraut sind, können Sie auch direkt in das genannte Kapitel springen um mehr über die iOS-Entwicklung zu erfahren.

23.1 Über iOS

Ganz allgemein betrachtet handelt es sich bei iPhone und iPad (und dem von Apple nicht mehr allzu aufwendig gepflegten iPod touch) um Mobilgeräte, deren Bedienung ausschließlich über einen Touchscreen erfolgt. Das ist durchaus ein besonderes und gerade bei der App-Entwicklung sehr wichtiges Merkmal. Benutzeroberflächen müssen aufgrund dessen ohne Probleme mit dem Finger zu bedienen sein. Das bedeutet, dass Schaltflächen und alle sonstigen auswählbaren Elemente groß genug sein müssen und auch nicht zu nah beieinander liegen dürfen. Wo der Mac eine pixelgenaue Auswahl mithilfe einer Maus beziehungsweise eines Trackpads erlaubt, ist eine derartig exakte Bedienung unter iOS nicht möglich.

Da iOS ausschließlich auf Mobilgeräten läuft, ist der schonende Umgang mit Ressourcen ebenfalls ein wichtiges Thema, um den Akku nicht unnötig zu belasten. Stromfressende Apps sind für den Endnutzer durchaus ein Grund, sich von diesen zu trennen und nach passenden Alternativen zu suchen. Erfreulicherweise unterstützt Sie das System hier bereits von Haus aus bestmöglich dabei, unnötig Ressourcen zu verschwenden.

Auch wenn es unter iOS inzwischen möglich ist, immens komplexe und aufwendige Apps zu kreieren, sollten Sie bei der App-Entwicklung nie den eigentlichen Fokus Ihrer App aus den Augen verlieren. Es kommt bei erfolgreichen iOS-Apps nicht darauf an, Feature über Feature in das Produkt hineinzupacken; das kann Nutzer womöglich eher überfordern und die benötigten Ressourcen einer iOS-App unnötig vergrößern. Achten Sie stattdessen darauf, sich auf *eine spezifische Aufgabe* zu konzentrieren, die Ihre App lösen soll, und lassen Sie

alles Unnötige und Überflüssige weg. Unter iOS kann dieser Ansatz maßgeblich zum Erfolg Ihrer App beitragen. Der Grund hierfür ist, dass iPhone und iPad – im Gegensatz zum Mac – meist eine kürzere Nutzungsdauer besitzen (das gilt insbesondere für das iPhone). Sie werden verwendet, um sich einer spezifischen Aufgabe anzunehmen (Mails checken, einen Post in sozialen Netzwerken absetzen, ein Foto schießen), und für genau diese Aufgabe soll eine passende App eine schnelle und unkomplizierte Lösung anbieten (siehe Bild 23.2). Lassen Sie im Zweifel also lieber Features weg, die dem Hauptzweck Ihrer Anwendung im Wege stehen könnten.

Bild 23.2 Egal ob Wetter-, Telefon-, Fotos- oder Notizen-App: Unter iOS widmet sich jede App standardmäßig einer zentralen Aufgabe, auf die der Nutzer sich fokussieren kann.

23.2 Funktionsweise einer iOS-App

Die wichtigste Basis zur Entwicklung von Apps für iOS liegt im Verständnis der grundlegenden Funktionsweise einer jeden App. In den kommenden Abschnitten erfahren Sie, aus welchen Bestandteilen und Dateien sich eine moderne iOS-App zusammensetzt, was beim Starten einer App passiert und mit welchem Framework wir es die meiste Zeit über bei der iOS-Entwicklung zu tun haben. Im Anschluss geht es weiter mit dem Erstellen eines ersten einfachen iOS-Projekts.

23.2.1 Bestandteile einer iOS-App

Jedes moderne iOS-Projekt setzt sich aus verschiedenen Bestandteilen und Dateien zusammen, die gemeinsam das Fundament und die Basis einer iOS-App bilden. Dazu gehören:

- App Delegate
- View-Controller
- Main-Storyboard

- Asset Catalog
- Info.plist

In den folgenden Abschnitten stelle ich Ihnen jedes dieser Elemente ausführlich vor und erkläre seine jeweilige Bedeutung. Im weiteren Verlauf dieses Kapitels werden Sie an passender Stelle noch mehr über sie erfahren.

23.2.1.1 App Delegate

Beim sogenannten *App Delegate* handelt es sich um eine Instanz, die zum `UIApplicationDelegate`-Protokoll konform ist. Die Methoden dieses Protokolls werden automatisch zu einem gegebenen Zeitpunkt vom System aufgerufen und informieren über verschiedene Events im Zusammenspiel mit der Nutzung einer App. So gibt es beispielsweise Methoden, die über den Start und das Beenden einer App informieren. Sie können diese Methoden implementieren, um zu den entsprechenden Events eigene Aktionen auszuführen.

Der App Delegate stellt damit in gewisser Weise den Dreh- und Angelpunkt einer iOS-App dar. Er wird als Erstes aktiv, sobald eine App gestartet wird, und informiert anschließend über alle Events, die in Bezug auf die App auftreten. Mehr über den App Delegate und die Methoden, die er bereitstellt, erfahren Sie in Abschnitt 23.2.2, „App-Start“ und Abschnitt 23.4, „Der UIApplicationDelegate“.

23.2.1.2 View-Controller

Ein *View-Controller* entspricht einer Ansicht, die ein Nutzer beim Verwenden einer iOS-App auf dem Display zu Gesicht bekommt. Er enthält die verschiedenen Bedienelemente wie Schaltflächen, Switches oder Tabellen, mit denen Apps gesteuert werden und über die sie Informationen für den Nutzer anzeigen.

Jede iOS-App besitzt wenigstens einen solchen View-Controller, in der Regel sind es aber deutlich mehr (eben für jede Art von Ansicht, die eine App zur Verfügung stellt). Ein großer Teil der Arbeit als App-Entwickler besteht im Erstellen und Gestalten solcher View-Controller. Mehr über dieses Element erfahren Sie in Abschnitt 23.5, „UIViewController im Detail“.

23.2.1.3 Main-Storyboard

Mithilfe sogenannter *Storyboards* bilden Sie Benutzeroberflächen Ihrer iOS-App mithilfe eines grafischen Editors (dem sogenannten *Interface Builder*) ab. In einem Storyboard können Sie ein oder mehrere View-Controller erstellen und mit den gewünschten Interface-Elementen wie Schaltflächen, Texten, Tabellen und so weiter versehen. Statt Ihre App-Oberflächen aufwendig händisch zu programmieren, kann Ihnen ein Storyboard hier viel Arbeit abnehmen und dank der grafischen Oberfläche direkt ein Gefühl dafür vermitteln, wie Ihre App tatsächlich bei der Ausführung aussehen wird.

Jedes neu erstellte iOS-Projekt bringt von Haus aus ein solches Storyboard mit dem Namen *Main* mit. Das können Sie direkt nutzen, um die verschiedenen View-Controller und Oberflächen für Ihre App zu gestalten und sie miteinander zu verbinden. Dieses Main-Storyboard dient gleichzeitig als Einstiegspunkt für Ihre App, indem das System einen der View-Controller daraus automatisch beim App-Start lädt und anzeigt. Mehr über den Start einer iOS-App erfahren Sie in Abschnitt 23.2.2, „App-Start“, weitere Informationen zur Arbeit mit Storyboards in der iOS-Entwicklung liefert Abschnitt 23.5, „UIViewController im Detail“.

23.2.1.4 Asset Catalog

Asset Catalogs dienen in der Entwicklung für die verschiedenen Apple-Plattformen zur Speicherung von Grafiken und Bildern, die Sie innerhalb einer App verwenden möchten. Unter iOS werden sie in jedem Fall dazu eingesetzt, das App-Icon einer Anwendung in allen benötigten Größen für die verschiedenen Geräteklassen (iPhone, iPad) einzubinden. Mehr zu Asset Catalogs und dem App-Icon einer iOS-App erfahren Sie in Abschnitt 23.8, „App-Icon“.

23.2.1.5 Info.plist

In der *Info.plist*-Datei werden verschiedene grundlegende Informationen und Einstellungen zu einer iOS-App gespeichert. Dazu gehören beispielsweise der Name der App, der Bundle Identifier und die Versionsnummer. Viele dieser Informationen werden bereits beim Erstellen eines neuen iOS-Projekts festgelegt, sie können aber jederzeit später noch angepasst und geändert werden.

Mehr zu den Einstellungsmöglichkeiten einer iOS-App und der *Info.plist*-Datei erfahren Sie in Abschnitt 23.9, „Target-Einstellungen“.

23.2.2 App-Start

Einer der wichtigsten Prozesse bei der Verwendung einer App ist deren Start. Hier werden alle grundlegenden Einstellungen festgelegt und die erste Ansicht geladen, über die der Nutzer mit der Bedienung der App startet.

Beim Start einer iOS-App spielen zwei Elemente eine entscheidende Rolle: der App Delegate und das Main-Storyboard. Zunächst wird eine Instanz des App Delegate vom System erstellt, anschließend wird im Main-Storyboard nach dem sogenannten *initialen View-Controller* gesucht. Hierbei handelt es sich um einen View-Controller innerhalb des Main-Storyboards, der beim Start einer App erstellt, geladen und auf dem Display angezeigt werden soll. Dieser initiale View-Controller stellt somit den Startpunkt einer App für den Nutzer dar.

Doch wie kommt es dazu? Wie wird eine App Delegate-Instanz automatisch vom System erzeugt, und woher weiß iOS, welchen View-Controller es laden und anzeigen soll? Beide Fragen werden in den kommenden beiden Abschnitten beantwortet.

23.2.2.1 Erzeugen des App Delegate

Der App Delegate wird in jedem iOS-Projekt mittels einer eigenen Klasse abgebildet. Diese hört standardmäßig auf den Namen `AppDelegate` und ist konform zum `UIApplicationDelegate`-Protokoll. Die standardmäßige Deklaration dieser Klasse in einem iOS-Projekt sehen Sie in Listing 23.1.

Listing 23.1 Deklaration der `AppDelegate`-Klasse

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    // Implementierung des App Delegate
}
```

Auffällig bei dieser Klassendeklaration ist das vorangestellte Schlüsselwort `@UIApplicationMain`. Eine damit gekennzeichnete Klasse wird beim Start der App vom System automatisch als App Delegate initialisiert. Es ist daher gar nicht notwendig, selbst eine Instanz dieser `AppDelegate`-Klasse zu erzeugen, um sie verwenden zu können, sobald sie nur mit dem Keyword `@UIApplicationMain` versehen ist.

UIApplication und UIResponder

Der Befehl `@UIApplicationMain` erzeugt nicht nur eine Instanz des App Delegate, sondern auch der Klasse `UIApplication`. Hierbei handelt es sich um ein Singleton, das zentral für Kontrolle und Koordination von unter iOS ausgeführten Apps verantwortlich ist. In gewisser Weise ist eine Instanz der `UIApplication`-Klasse somit das zentrale Herzstück einer jeden iOS-App.

Der App Delegate ist direkt mit der Singleton-Instanz von `UIApplication` verknüpft. Diese besitzt eine Property namens `delegate` vom Typ `UIApplicationDelegate` (also jenem Protokoll, zu dem der App Delegate konform ist). Der Befehl `@UIApplicationMain` erzeugt nicht nur Instanzen von `UIApplication` und der projektspezifischen App Delegate-Klasse, sondern weist die App Delegate-Instanz auch noch der `delegate`-Property von `UIApplication` zu. Es wird also durch `@UIApplicationMain` unmittelbar eine Kopplung zwischen `UIApplication` und App Delegate hergestellt.

In Listing 23.1 ist Ihnen darüber hinaus womöglich aufgefallen, dass die `AppDelegate`-Klasse von der Klasse `UIResponder` abgeleitet ist. Es handelt sich hierbei um eine abstrakte Klasse, die für das Event-Handling in iOS verantwortlich ist. Wo immer Events auftreten können (beispielsweise durch Tastatureingaben oder Tippen auf einen Button) spielt `UIResponder` eine entscheidende Rolle. Jedes Element einer iOS-App, das auf mögliche Events reagieren kann, ist von dieser Klasse abgeleitet. Mehr zu `UIResponder` und Event-Handling erfahren Sie unter anderem in Abschnitt 23.6, „Oberflächen gestalten mit UIView“, sowie in den weiteren Abschnitten, in denen es um das Reagieren auf Nutzereingaben und -aktionen geht. ■

23.2.2.2 Laden des initialen View-Controllers

Jede iOS-App benötigt einen sogenannten *initialen View-Controller*. Es handelt sich dabei um eine Klasse, die der Startansicht einer App entspricht. Sie ist das, was der Nutzer nach Starten einer App als Erstes zu sehen bekommt.

Um einen solchen initialen View-Controller zu erstellen, kommt in der Regel das Main-Storyboard zum Einsatz. Darin wird ein View-Controller erstellt, der als initialer View-Controller deklariert wird. Um diese Deklaration durchzuführen, wählt man im Storyboard zunächst den View-Controller aus, der beim Starten der App geladen und angezeigt werden soll, und wechselt anschließend in den Attributes Inspector. Dort findet sich eine Checkbox mit dem Titel *Is Initial View Controller*. Sie muss aktiviert sein, damit ein View-Controller als initialer View-Controller des zugrunde liegenden Storyboards deklariert wird. Dieser View-Controller wird parallel dazu noch mit einem Pfeilsymbol am linken Rand gekenn-

zeichnet, um darüber zusätzlich zu signalisieren, welcher View-Controller der initiale ist (siehe Bild 23.3). Pro Storyboard-Datei kann nur ein View-Controller als initial gekennzeichnet sein.

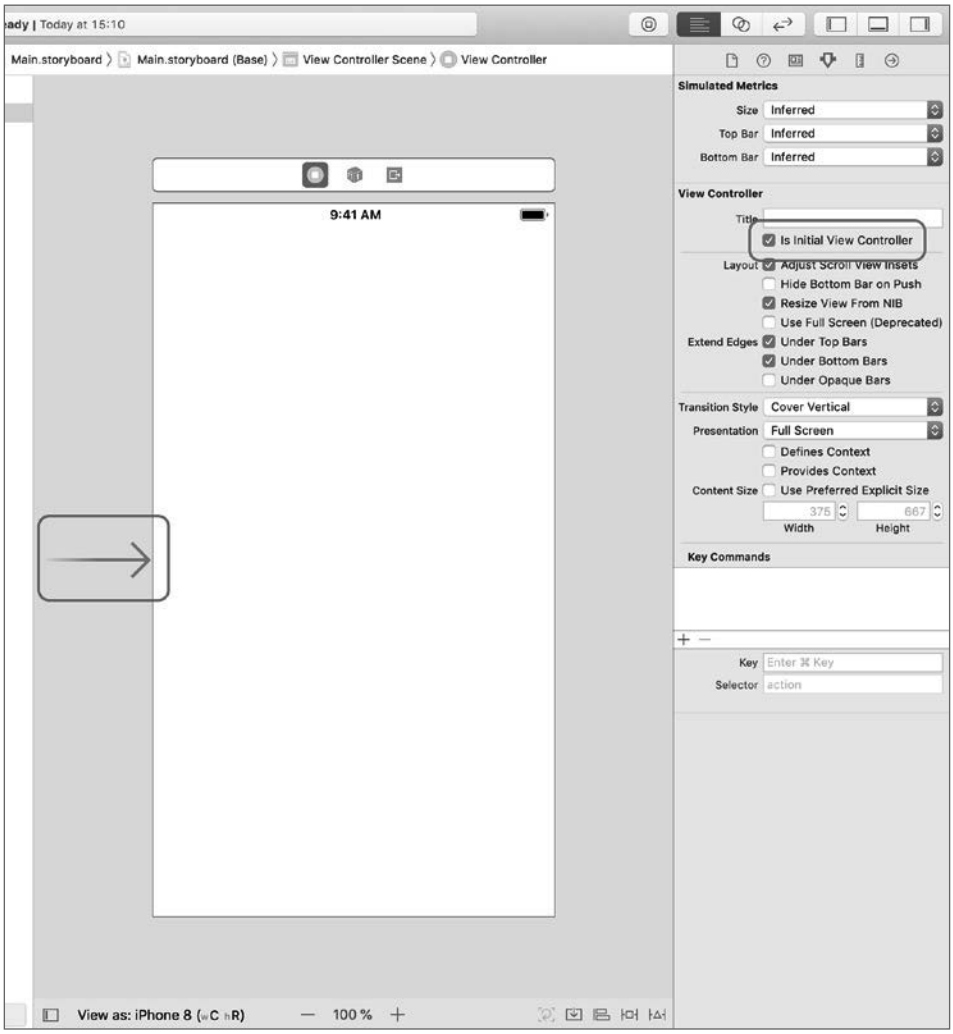


Bild 23.3 Mithilfe der Checkbox „Is Initial View Controller" definiert man im Storyboard, welcher View-Controller initial geladen und angezeigt werden soll.

Abschließend muss in den Einstellungen der iOS-App noch festgelegt werden, *welches* Storyboard als Einstiegspunkt dient und dessen initialer View-Controller beim Start geladen und angezeigt wird. Dazu wählt man das iOS-Target im Xcode-Projekt aus und scrollt zum Bereich *Deployment Info*. Dort findet man ein Feld *Main Interface*, in das der Name der gewünschten Storyboard-Datei eingetragen werden muss (siehe Bild 23.4).

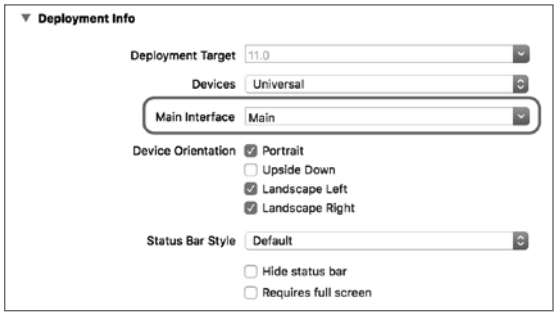


Bild 23.4
In den Target-Einstellungen der iOS-App geben Sie im Feld „Main Interface" den Namen des Storyboards an, dessen initialer View-Controller beim Starten der App geladen und angezeigt werden soll.

Standardmäßig kommt an dieser Stelle immer das Main-Storyboard zum Einsatz. Solange Sie dieses nicht umbenennen oder weitere Storyboard-Dateien hinzufügen, von denen Sie eines stattdessen als Startpunkt für Ihre App verwenden möchten, brauchen Sie an diesen Standardeinstellungen nichts zu ändern.

Das erste Projekt

Falls Ihnen an dieser Stelle noch nicht klar ist, wie die Arbeit mit dem Main-Storyboard und dem initialen View-Controller in der Praxis funktioniert, ist das nicht schlimm. Dieser Abschnitt soll zunächst einmal einen theoretischen Überblick über den Ablauf bieten, den eine iOS-App beim Starten durchläuft. In Abschnitt 23.3, „Ein erstes iOS-Projekt", betrachten wir die beschriebenen Elemente noch einmal in der Praxis und ich verdeutliche noch einmal die eben beschriebene Funktionsweise. ■

23.2.3 Das UIKit-Framework

Die Basis für alle iOS-spezifischen Aktionen, Klassen und Typen stellt das *UIKit*-Framework dar. Darin finden sich alle Elemente, die speziell für die Entwicklung von iOS-Apps von zentraler Bedeutung sind. Dazu gehören beispielsweise die bereits in den vorherigen Abschnitten vorgestellten Typen `UIApplicationDelegate`, `UIApplication` und `UIResponder`. Aber auch Typen wie `UIViewController` (zum Abbilden von View-Controllern) und `UIView` (zum Erstellen von Interface-Elementen) sowie eine Vielzahl weiterer sind im UIKit-Framework untergebracht. In diesem Kapitel sowie auch im folgenden Kapitel 24, „iOS – App-Entwicklung", werden ausschließlich die Typen und Funktionen des UIKit-Frameworks im Fokus stehen.

Aufgrund seiner Wichtigkeit in der iOS-Entwicklung wird UIKit an allen Stellen importiert, an denen Zugriff auf die entsprechenden Elemente des Frameworks benötigt wird (so zum Beispiel im App Delegate oder jeder View-Controller-Klasse). In den folgenden Abschnitten sowie in Kapitel 24, „iOS – App-Entwicklung", werden Sie eine Vielzahl von Typen kennenlernen, die das UIKit-Framework für iOS-Entwickler zur Verfügung stellt.

23.3 Ein erstes iOS-Projekt

In diesem Abschnitt führe ich Sie durch das Erstellen eines ersten simplen iOS-Projekts, um Sie mit den zuvor beschriebenen Elementen vertraut zu machen und Ihnen ein Gefühl für die Entwicklung von iOS-Apps zu vermitteln. Die Beispiel-App, die wir zu diesem Zweck erstellen, gibt schlicht den Text *Hello World!* auf dem Bildschirm aus. Das mag noch nicht sonderlich komplex sein, gibt Ihnen aber einen ersten guten Überblick über die Bestandteile eines iOS-Projekts und die Arbeit mit Xcode. Packen wir's an. ☺

23.3.1 Auswahl einer Template-Vorlage

Starten Sie Xcode und wählen Sie im Begrüßungsfenster *Create a new Xcode project* aus (siehe Bild 23.5). Alternativ können Sie auch im Xcode-Menü *File → New → Project…* wählen oder das Tastaturkürzel **Shift+cmd+N** verwenden.

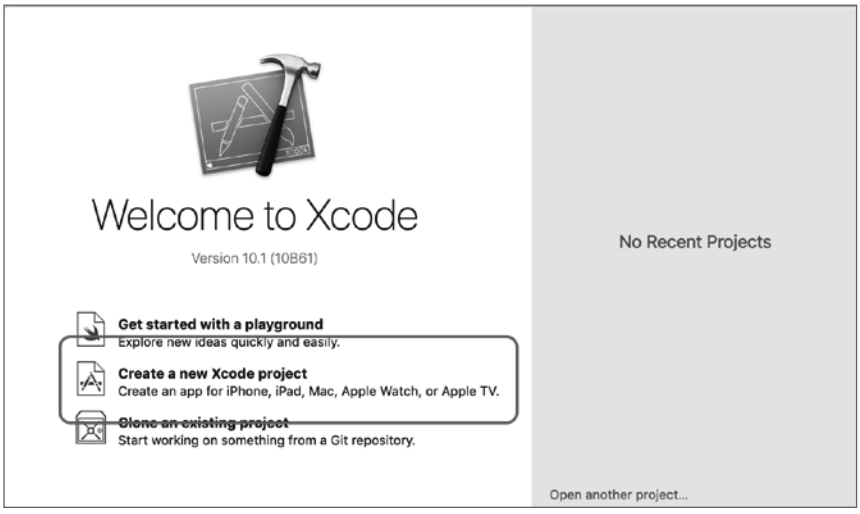


Bild 23.5 Über das Begrüßungsfenster von Xcode können Sie ein neues Projekt erstellen.

Im Anschluss öffnet sich die Template-Auswahl zur Erstellung eines neuen Projekts. Wählen Sie dort zunächst im oberen Reiter als Plattform *iOS* aus. Im Abschnitt *Application* finden Sie anschließend alle Vorlagen, die Sie zum Erstellen eines neuen iOS-Projekts verwenden können (siehe Bild 23.6). Wählen Sie für dieses erste Beispiel den Punkt *Single View App* aus. Diese Vorlage bietet sich generell für alle Arten von Projekten an, bei denen Sie nur mit den grundlegendsten Elementen beginnen und bei der weiteren Gestaltung Ihrer App die größtmögliche Flexibilität genießen möchten.

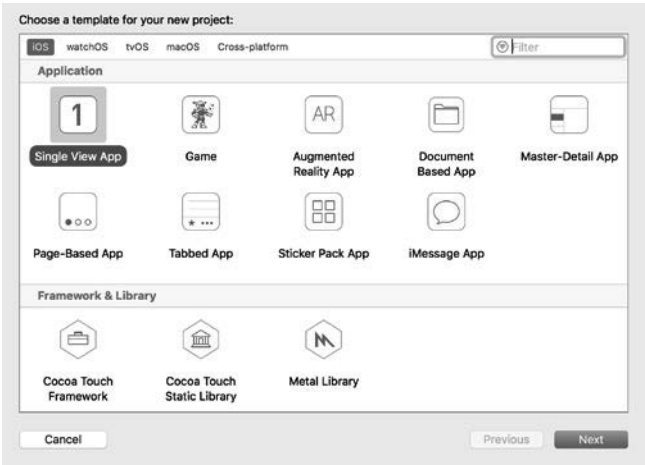


Bild 23.6 Im Abschnitt „Application“ des Reiters „iOS“ finden Sie alle Vorlagen, um mit der Entwicklung einer neuen iOS-App beginnen zu können.

Nach einem anschließenden Klick auf *Next* legen Sie im nächsten Fenster die grundlegenden Optionen für das neue Projekt fest. Dazu gehört vor allen Dingen der *Product Name*, bei dem es sich um den Namen Ihrer neuen App handelt (in diesem Beispiel gebe ich dort *Hello World* ein). Haben Sie sich bereits mit einem Entwickler-Account in Xcode registriert, können Sie diesen in der Auswahlbox unter dem Titel *Team* auswählen. Zusätzlich können Sie den *Organization Name* sowie den *Organization Identifier* für Ihre App festlegen. Im Bereich *Language* entscheiden Sie sich für *Swift* (siehe Bild 23.7). Die Checkboxen *Use Core Data*, *Include Unit Tests* und *Include UI Tests* können Sie für dieses Beispiel-Projekt deaktivieren.

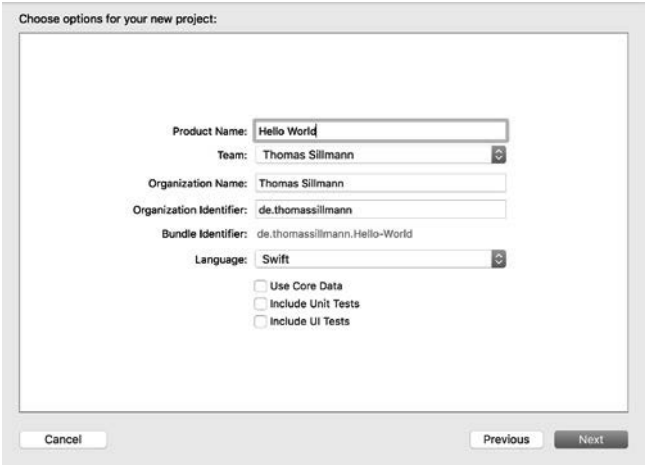


Bild 23.7 In diesem Fenster geben Sie die grundlegenden Informationen zu Ihrem neuen iOS-Projekt ein.

Das Erstellen des Projekts schließen Sie anschließend mit einem Klick auf die Schaltfläche *Next* ab. Zum Schluss wählen Sie noch den gewünschten Speicherort für das Xcode-Projekt

auf Ihrem Mac aus und bestätigen diese Auswahl per Klick auf die Schaltfläche *Create* (siehe Bild 23.8). Daraufhin begrüßt Sie das Projektfenster von Xcode (siehe Bild 23.9).

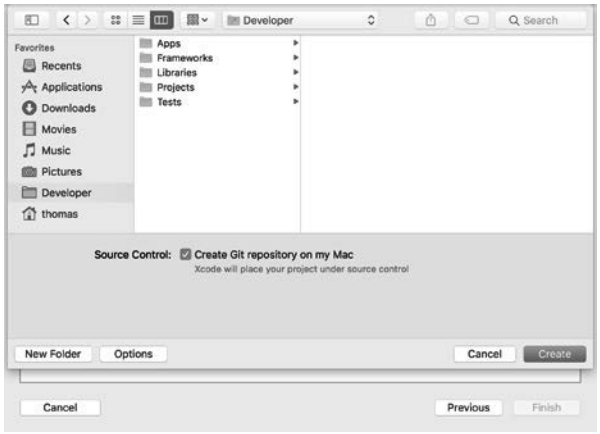


Bild 23.8 Wählen Sie einen Speicherort für das neue Xcode-Projekt und bestätigen Sie die Auswahl mithilfe der Schaltfläche „Create“. Die standardmäßig aktive Checkbox „Create Git repository on my Mac“ können Sie aktiviert lassen.

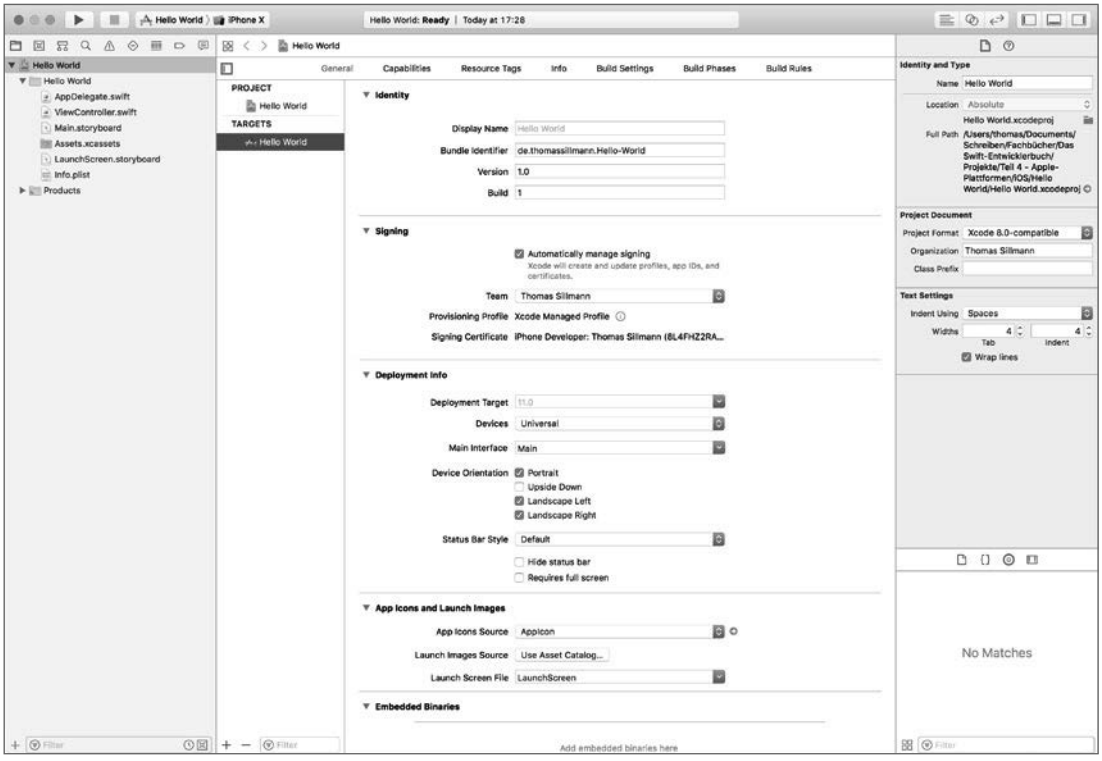


Bild 23.9 Nach dem erfolgreichen Erstellen des neuen Projekts wird es direkt von Xcode geöffnet.

23.3.2 Rundgang durch die erstellten Dateien

Betrachten wir nun zunächst einmal die verschiedenen Dateien, die Xcode automatisch mitsamt diesem ersten neuen iOS-Projekt erstellt hat. Diese werden im linken Bereich – der sogenannten *Navigator Area* – innerhalb des Project Navigators aufgeführt. In Bild 23.10 sehen Sie eine Detailansicht des entsprechenden Ausschnitts.

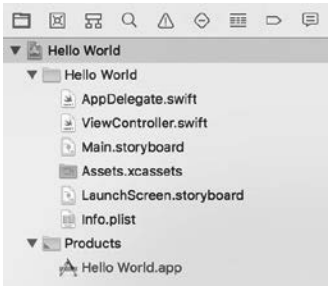
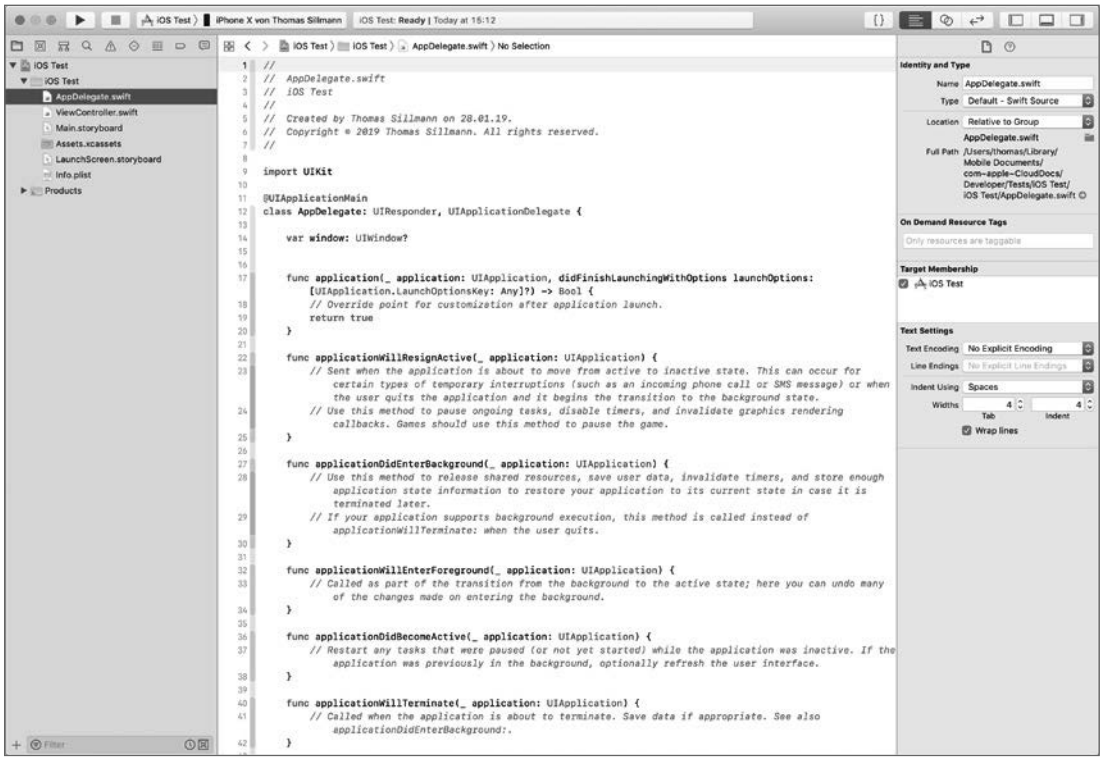


Bild 23.10
Hier sehen Sie die Dateien des neu erstellten iOS-Xcode-Projekts.

AppDelegate.swift

Zunächst ist da die *AppDelegate.swift*-Datei. Wählt man diese im Project Navigator aus, wird im mittleren Bereich von Xcode – der sogenannten *Editor Area* – der zugehörige Code angezeigt (siehe Bild 23.11). Dieser dürfte bei Ihnen ähnlich aussehen wie in Listing 23.2.




```
//
//  AppDelegate.swift
//  iOS Test
//
//  Created by Thomas Sillmann on 28.01.19.
//  Copyright © 2019 Thomas Sillmann. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for
            certain types of temporary interruptions (such as an incoming phone call or SMS message) or when
            the user quits the application and it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering
            callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough
            application state information to restore your application to its current state in case it is
            terminated later.
        // If your application supports background execution, this method is called instead of
            applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        // Called as part of the transition from the background to the active state; here you can undo many
            of the changes made on entering the background.
    }

    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the application was inactive. If the
            application was previously in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate. See also
            applicationDidEnterBackground:.
    }
```

Bild 23.11 Nach Auswahl der AppDelegate.swift-Datei wird im mittleren Editor-Bereich der zugehörige Quelltext angezeigt.

Listing 23.2 Standard-Code der `AppDelegate`-Klasse

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state.
This can occur for certain types of temporary interruptions (such as an incoming
phone call or SMS message) or when the user quits the application and it begins the
transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate
graphics rendering callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate
timers, and store enough application state information to restore your application to
its current state in case it is terminated later.
        // If your application supports background execution, this method is called
instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        // Called as part of the transition from the background to the active state;
here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the
application was inactive. If the application was previously in the background,
optionally refresh the user interface.
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if
appropriate. See also applicationDidEnterBackground:.
    }

}
```

In dieser von Xcode standardmäßig generierten Klasse fallen mehrere Dinge auf. Zunächst ist die Klasse mit dem in Abschnitt 23.2.2.1, „Erzeugen des App Delegate“, vorgestellten `@UIApplicationMain`-Schlüsselwort versehen, was bedeutet, dass es sich bei ihr sowohl um den Einstiegspunkt der App wie auch um den Delegate des `UIApplication`-Singleton handelt. Letzteres wird auch durch die Zuweisung des `UIApplicationDelegate`-Protokolls bei der Klassendeklaration deutlich. Neben einer `window`-Property wurden auch bereits verschiedene Methoden des `UIApplicationDelegate`-Protokolls innerhalb der `AppDelegate`-Klasse implementiert und mit passenden Kommentaren versehen. Das soll dabei helfen, die

wichtigsten dieser Methoden kennenzulernen und zu verstehen, wofür sie gut sind. Beispielsweise findet sich hier die Methode `applicationWillResignActive(_:)`, die immer dann vom System aufgerufen wird, wenn die App inaktiv wird (weil zum Beispiel ein eingehender Anruf die Anwendung unterbricht oder der Nutzer auf den Home-Bildschirm zurückkehrt). Mehr zu den verfügbaren Methoden des App Delegate und ihrem jeweiligen Anwendungszweck erfahren Sie in Abschnitt 23.4, „Der UIApplicationDelegate“.

Die window-Property

Alle Ansichten einer iOS-App, die in Form von View-Controllern abgebildet und umgesetzt werden, werden in einer `UIWindow`-Instanz eingefügt und darin angezeigt. Ein `UIWindow` stellt somit die eigentliche Basis für die Anzeige einer iOS-App dar, ohne ein `UIWindow` kann eine App nicht auskommen.

Im Gegensatz zu macOS setzen Apps unter iOS meist nur auf eine einzige `UIWindow`-Instanz. Deren Inhalt wird über die View-Controller definiert, die innerhalb dieses Fensters eingeblendet werden. Möchte man beispielsweise eine neue Ansicht laden, stellt man diese in der Regel über einen View-Controller bereit, den man dann auf dem bereits existierenden Fenster einblendet. Entsprechend hat man es in der iOS-Entwicklung so gut wie nie direkt mit der Manipulation von `UIWindow`-Instanzen zu tun.

Durch die Verwendung von Storyboards wird für den initialen View-Controller auch automatisch vom System eine passende `UIWindow`-Instanz erzeugt, in die dieser View-Controller eingebunden wird; wir selbst brauchen dafür nicht das Geringste zu tun. Dieser View-Controller, der direkt zur Anzeige innerhalb eines `UIWindow` verwendet wird, wird auch als *Root-View-Controller* bezeichnet. Im Code kann er über die `rootViewController`-Property einer `UIWindow`-Instanz ausgelesen und gesetzt werden (siehe Bild 23.12).

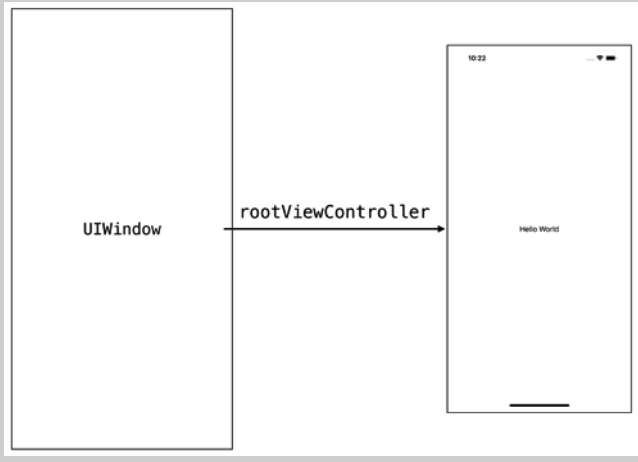


Bild 23.12 Der initiale View-Controller wird mithilfe von Storyboards automatisch als Root-View-Controller für die ebenfalls automatisch erzeugte `UIWindow`-Instanz gesetzt.

Bei der Verwendung von Storyboards wird der initiale View-Controller automatisch dem zugrunde liegenden `UIWindow` als Root-View-Controller zugewiesen. Die `UIWindow`-Instanz selbst wird in diesem Fall an den App Delegate gekoppelt, weshalb sich beim Erstellen eines neuen iOS-Projekts eine entsprechende `window`-Property in der `AppDelegate`-Klasse findet; sie verweist auf die `UIWindow`-Instanz.

Verzichtet man auf Storyboards, muss man beim Start der App selbst dafür sorgen, dass eine `UIWindow`-Instanz erzeugt und ein Root-View-Controller geladen wird. Das tut man typischerweise innerhalb der `AppDelegate`-Methode `application(_:didFinishLaunchingWithOptions:)` (mehr zu den verschiedenen Methoden des App Delegate erfahren Sie in Abschnitt 23.4, „Der UIApplicationDelegate“). In Listing 23.3 sehen Sie ein Beispiel, das alle notwendigen programmatischen Schritte zum Erzeugen und Anzeigen eines `UIWindow` aufführt.

Listing 23.3 Programmatisches Erstellen und Laden einer `UIWindow`-Instanz

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    // Erstellen einer Instanz des Root-View-Controller.
    let rootViewController = UIViewController()

    // Erstellen der UIWindow-Instanz und Zuweisung zur window-Property.
    window = UIWindow(frame: UIScreen.main.bounds)

    // Zuweisen des Root-View-Controller zum UIWindow.
    window?.rootViewController = rootViewController

    // Anzeigen des UIWindow.
    window?.makeKeyAndVisible()

    return true
}
```

Hier noch ein paar Anmerkungen zum Code aus Listing 23.3:

Bei der Initialisierung einer `UIWindow`-Instanz kommt der Initializer `init(frame:)` zum Einsatz. Dieser erwartet als Parameter die Größe und Position des Fensters für die App. Im gezeigten Beispiel wurden hier die Ausmaße des zugrunde liegenden Displays übergeben (`UIScreen.main.bounds`). Mehr zum Erstellen von Ansichten und zur Arbeit mit Größen in der iOS-Entwicklung erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“.

Um ein Fenster einzublenden, müssen Sie auf der entsprechenden `UIWindow`-Instanz die Methode `makeKeyAndVisible()` aufrufen. In dem gezeigten Beispiel geschieht das, nachdem die `UIWindow`-Instanz erzeugt, der `window`-Property zugewiesen und ein Root-View-Controller erstellt und konfiguriert wurde.

Bei der Arbeit mit Storyboards werden Ihnen all diese Schritte bereits im Vorhinein abgenommen.

Den vorgegebenen Code kann man prinzipiell so stehen lassen, da er keinerlei eigene Logik enthält und so keine Auswirkungen auf die Funktionsweise der neu erstellten App hat. Ich persönlich würde Ihnen aber empfehlen, alle Methoden konsequent zu löschen, die Sie nicht implementieren und somit auch nicht benötigen. Für dieses Beispiel ist dieses Vorgehen aber irrelevant.

ViewController.swift

Als Nächstes betrachten wir die *ViewController.swift*-Datei, innerhalb derer die `ViewController`-Klasse deklariert wird. Der Code darin dürfte in etwa so aussehen wie der in Listing 23.4 gezeigte. Die Klasse ist von `UIViewController` abgeleitet und es werden die beiden Methoden `viewDidLoad()` und `didReceiveMemoryWarning()` dieser Superklasse darin überschrieben. Auch hier gilt, dass Sie für den Moment theoretisch beide Methoden aus der Klasse entfernen können, da diese in der `ViewController`-Subklasse keine eigene Logik mit sich bringen. Xcode fügt sie lediglich vorab bereits ein, davon ausgehend, dass Sie sie möglicherweise selbst implementieren.

Listing 23.4 Standard-Code der `ViewController`-Klasse

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}
```

Die `ViewController`-Klasse ist in diesem neuen iOS-Projekt bereits als initialer View-Controller konfiguriert. Der Code, der in dieser Klasse implementiert ist, wird somit im Zusammenspiel mit der Startansicht der App ausgeführt. Entsprechend werden darin alle Properties und Methoden untergebracht, die für die Startansicht relevant sind. Gleichzeitig wird die `ViewController`-Klasse als Root-View-Controller für die `window`-Property des App Delegate gesetzt (siehe hierzu auch den vorherigen Abschnitt „AppDelegate.swift“).

Warum der View-Controller als Startpunkt dient und an welcher Stelle diese Konfiguration festgelegt ist, erfahren Sie im folgenden Abschnitt „Main.storyboard“.

Main.storyboard

Innerhalb der *Main.storyboard*-Datei wird das Interface der neu erstellten App auf Basis einer grafischen Oberfläche abgebildet. Ruft man diese Datei auf, wird innerhalb der Editor Area ein Fenster für einen einzelnen View-Controller angezeigt, der auf der linken Seite mit einem Pfeil versehen ist (siehe Bild 23.13). Wie Sie bereits erfahren haben, handelt es sich hierbei um den initialen View-Controller dieses Storyboards. Wählt man nun diesen View-Controller per Klick in die obere schmale Leiste (in der *View Controller* steht) aus und wechselt in der Inspectors Area von Xcode in den Identitiy Inspector, sieht man, dass diesem View-Controller die `ViewController`-Klasse aus der *ViewController.swift*-Datei zugewiesen

ist (siehe Bild 23.14). Dieses Interface ist somit bereits direkt mit der in „ViewController.swift“ vorgestellten Klasse verknüpft.

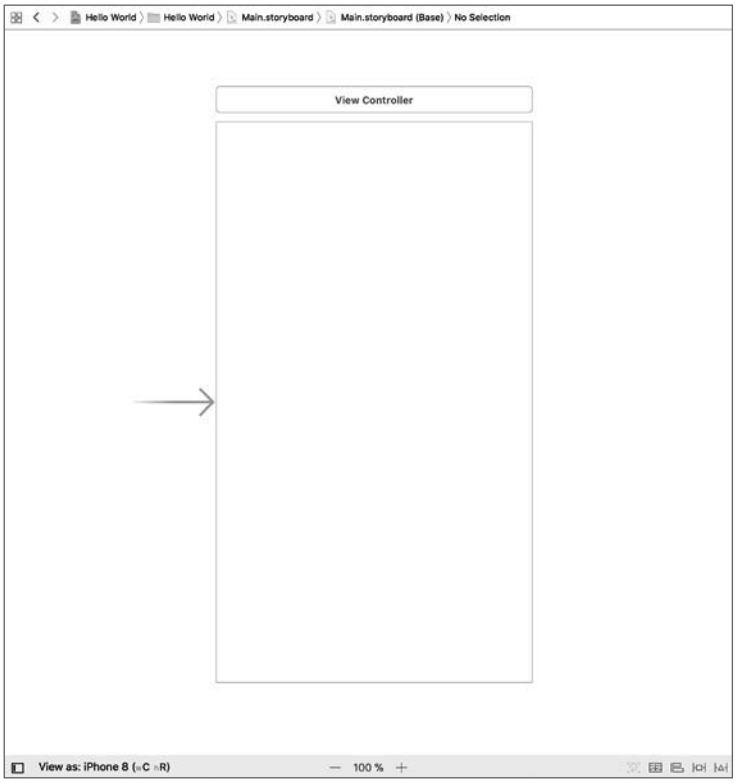


Bild 23.13 Die Main.storyboard-Datei enthält einen ersten initialen View-Controller, der beim Start der App geladen und angezeigt wird.

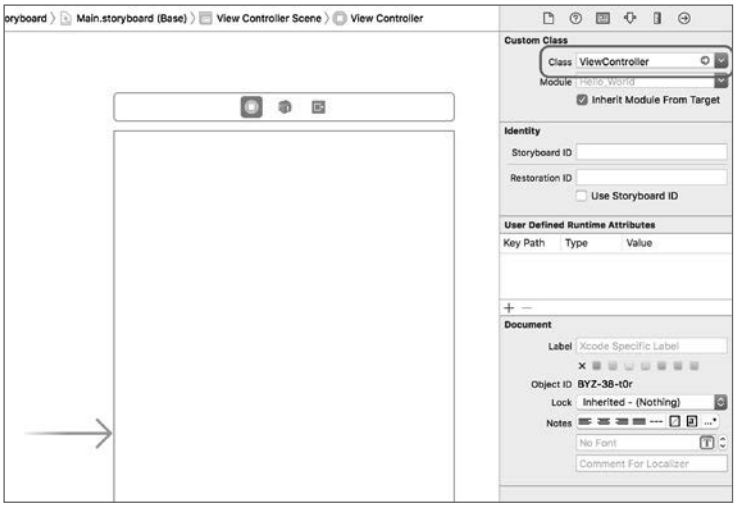


Bild 23.14 Das View-Controller-Interface in der Main.storyboard-Datei ist bereits mit der ViewController-Klasse verknüpft.

Weitere Dateien

Daneben verfügt das neu erstellte Xcode-Projekt noch über einige weitere Dateien. *Assets.xcassets* in ein sogenannter Asset Catalog und dient zum Speichern von Bildern, die innerhalb der App genutzt werden sollen, und zur Erstellung eines App-Icons. Die Datei *Launch.storyboard* enthält ebenfalls einen View-Controller, der beim Starten der App angezeigt wird (also noch bevor der initiale View-Controller aus der *Main.storyboard*-Datei geladen und angezeigt wird). Er dient als Lückenfüller für die Zeit zwischen dem Starten der App und deren eigentlicher Verfügbarkeit. Und in der *Info.plist*-Datei werden verschiedene Informationen zur App gespeichert, beispielsweise der Product Name oder die Versionsnummer. Diese Elemente werden an gegebener Stelle im Buch noch genauer vorgestellt.

Das letzte Element des neu erstellten iOS-Xcode-Projekts befindet sich im Ordner *Products* und trägt den gleichen Namen, der bei der Erstellung des Projekts als Product Name definiert wurde, mit der Dateiendung *.app*. Diese Datei wird erzeugt, sobald das Xcode-Projekt das erste Mal erfolgreich gebaut wurde, und braucht uns in der Regel nicht weiter zu interessieren. Sie ist in gewisser Weise ein Verweis auf die von uns erstellte App.

23.3.3 Hello World

Kommen wir nun zur eigentlichen Aufgabe und kreieren unsere erste iOS-App, die den Text „Hello World!“ auf dem Bildschirm ausgibt. Dazu müssen wir keine einzige Zeile Code schreiben. Stattdessen wechseln wir in die Datei *Main.storyboard* und rufen dort die *Objects Library* von Xcode über die entsprechende Schaltfläche am oberen rechten Rand von Xcode auf (siehe Bild 23.15).

Die Objects Library verfügt über zwei Ansichtsmodi, zwischen denen Sie über die Schaltfläche am oberen linken Rand wechseln können. In Bild 23.15 ist die sogenannte *Icon View* zu sehen, Bild 23.16 zeigt die alternative (und standardmäßig in Xcode aktive) *List View*.

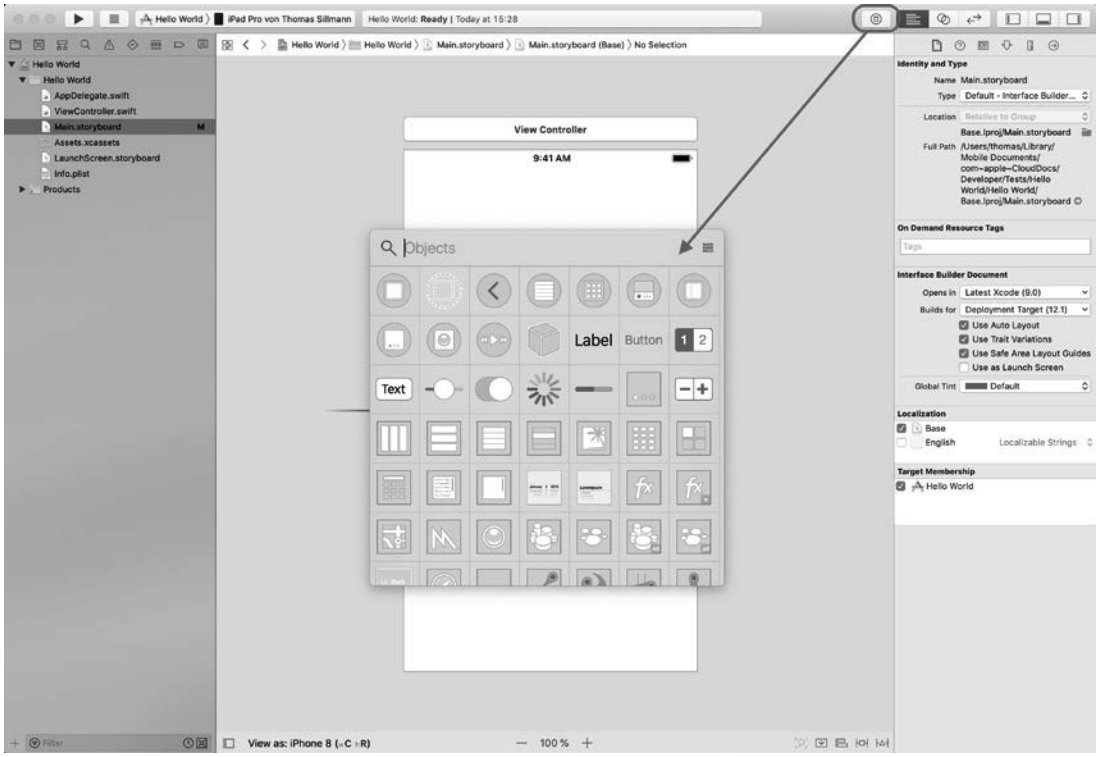


Bild 23.15 Die Objects Library blenden Sie über die gekennzeichnete Schaltfläche am oberen rechten Rand von Xcode ein.

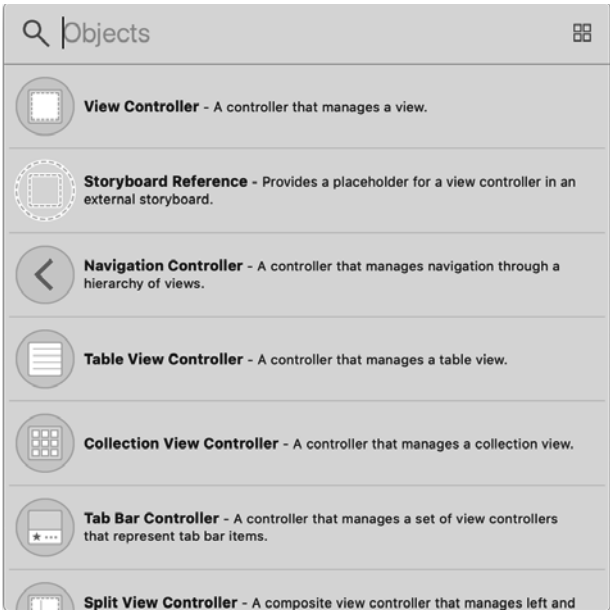


Bild 23.16 Über die Schaltfläche oben rechts lässt sich der Ansichtsmodus der Objects Library wechseln. Auf diesem Screenshot ist die sogenannte „List View“ aktiv.

Suchen Sie nun innerhalb der Objects Library nach dem Element *Label* (siehe Bild 23.17). Sie können dazu auch das Suchfeld am oberen Rand der Objects Library benutzen, um nach dem gewünschten Element zu suchen und alle anderen auszublenden.

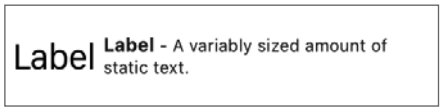


Bild 23.17 Das Label-Element wird in Form eines einfachen Textes in der Objects Library dargestellt.

Haben Sie das Label-Objekt gefunden, klicken Sie es mit der linken Maustaste an, halten diese anschließend gedrückt und bewegen die Maus auf die große weiße Fläche des View-Controllers, der in der *Main.storyboard*-Datei zu sehen ist. Wenn Sie die linke Maustaste nun wieder loslassen, wird das Label an der Stelle im View-Controller platziert, über der sich der Mauszeiger befindet. Sie können anschließend jederzeit das Label-Objekt frei im View-Controller bewegen und an einer anderen Stelle positionieren. Blaue Hilfslinien helfen Ihnen dabei, das Element optimal auszurichten (siehe Bild 23.18). Platzieren Sie für diese erste Beispiel-App das Label in etwa im mittleren Bereich des View-Controllers.

Bild 23.18 Die blauen Hilfslinien helfen bei der optimalen Positionierung des Labels.

Das so platzierte Label ist nun Teil der Ansicht, die beim Starten dieser App angezeigt wird. Bleibt zum Abschluss nur noch das Ändern des Textes des Labels. Das können Sie auf zwei verschiedene Arten durchführen.

Möglichkeit eins besteht darin, das platzierte Label im View-Controller doppelt anzuklicken. Anschließend können Sie den gewünschten Text für das Label – in diesem Beispiel „Hello World!“ – direkt eintragen (siehe Bild 23.19).

Bild 23.19
Den Text des Labels können Sie direkt im View-Controller ändern, indem Sie das Element doppelt anklicken.

Alternativ dazu wählen Sie das Label-Element aus und öffnen den Attributes Inspector in der Inspectors Area. Dort finden Sie an zweiter Stelle ein Textfeld, das den Text des Labels enthält. Ändern Sie ihn dort und drücken anschließend die Return-Taste oder klicken an eine andere Stelle, wird der Text des Labels entsprechend aktualisiert (siehe Bild 23.20).

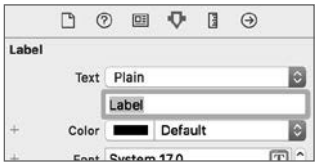


Bild 23.20
Der Text eines Labels kann auch innerhalb des Attributes Inspector geändert werden.

Nutzen Sie eine der genannten Möglichkeiten, um den Text des Labels so zu „Hello World!“ zu ändern. Anschließend können Sie noch einmal die Position des Labels mithilfe der blauen Hilfslinien verschieben und optimieren.

Damit wäre unsere erste einfache iOS-App bereits vollständig fertiggestellt! Um das Ergebnis einmal live zu betrachten, führen wir die App in einem der verfügbaren iOS-Simulatoren aus. Klicken Sie dazu im oberen linken Bereich von Xcode auf die Schaltfläche neben dem Namen Ihrer App. Anschließend öffnet sich eine Liste, in der im Bereich *iOS Simulators* alle in Xcode eingerichteten Simulatoren aufgeführt werden, in denen die App ausgeführt werden kann (siehe Bild 23.21).

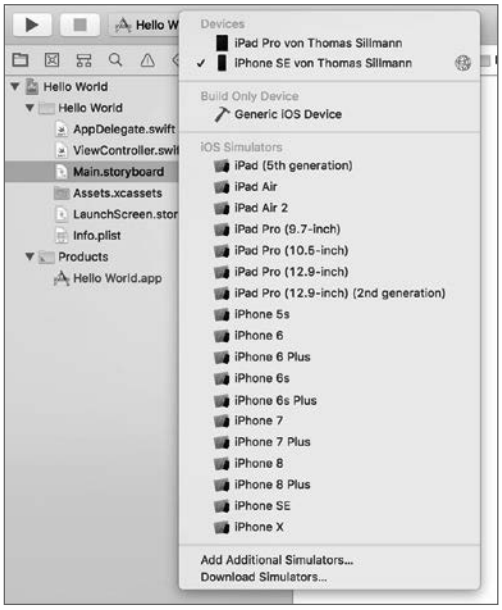


Bild 23.21
Xcode führt im Bereich „iOS Simulators“ alle installierten und eingerichteten Simulatoren auf, in denen eine iOS-App ausgeführt werden kann.

Wählen Sie in diesem Fenster einen der verfügbaren Simulatoren und klicken Sie anschließend auf die Run-Schaltfläche links oben (der Button sieht aus wie der Play-Button in einer Video- oder Musik-App). Anschließend „baut“ Xcode die App, startet den ausgewählten Simulator und darin die App (siehe Bild 23.22). Dieser Vorgang kann – je nach Hardware-Konfiguration des zugrunde liegenden Mac – durchaus einen Moment dauern (gerade das Starten der Simulatoren kann etwas Zeit in Anspruch nehmen).

Bild 23.22
Unsere erste iOS-App läuft im Simulator!

Herzlichen Glückwunsch! Bis zu diesem Punkt haben Sie bereits einmal einen ersten Blick auf die Bestandteile eines iOS-Projekts geworfen, Ihr erstes Interface erstellt und Ihre App erfolgreich ausgeführt. Die folgenden Abschnitte dieses Kapitels vertiefen die hier angeschnittenen Themen und führen Sie weiter in die Grundlagen der App-Entwicklung für iOS ein.

23.4 Der UIApplicationDelegate

Die Singleton-Instanz der Klasse `UIApplication` ist das Herzstück jeder iOS-App. Sie wird mithilfe des Befehls `@UIApplicationMain` automatisch erzeugt und ihr wird ein Delegate-Objekt zugewiesen, der sogenannte *App Delegate*. `UIApplication` ruft diesen Delegate auf, sobald wichtige Ereignisse in Bezug auf den Lebenszyklus einer iOS-App auftreten.

Jedes neu erstellte iOS-Projekt besitzt mit `AppDelegate` eine Klasse, die als App Delegate des `UIApplication`-Singleton fungiert. Damit eine Klasse als App Delegate fungieren kann, muss sie zwei Anforderungen erfüllen:

- Sie muss konform zum `UIApplicationDelegate`-Protokoll sein.
- Sie muss mit dem Schlüsselwort `@UIApplicationMain` versehen werden.

Beides ist bei neu erstellten Projekten bei der Deklaration der `AppDelegate`-Klasse der Fall, so wie in Listing 23.5 zu sehen.

Listing 23.5 Deklaration eines App Delegates

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    <Implementierung der Klasse>

}
```

Wie beschrieben, ruft das automatisch erzeugte `UIApplication`-Singleton diesen App Delegate nun auf, wenn bestimmte Ereignisse im Lebenszyklus einer iOS-App auftreten. Doch was genau bedeutet das? Was ist der *Lebenszyklus* einer iOS-App? Und wie sieht der genau aus?

23.4.1 Lebenszyklus einer iOS-App

Der Lebenszyklus einer iOS-App beschreibt diverse Ereignisse, die während der Verwendung einer App auftreten (können). Beispiele für solche Ereignisse sind das Starten einer App, das Beenden einer App, das Wechseln einer App in den Hintergrund oder das Unterbrechen einer laufenden App. Für jedes dieser Ereignisse (und noch einige mehr) definiert das `UIApplicationDelegate`-Protokoll entsprechende Methoden, die automatisch vom `UIApplication`-Singleton aufgerufen werden, sobald sie eintreten. Möchten Sie somit auf ein bestimmtes Ereignis reagieren (beispielsweise beim Beenden der App noch ungesicherte Daten speichern), können Sie die passende `UIApplicationDelegate`-Methode in Ihrer App Delegate-Klasse implementieren und darin die gewünschten Befehle unterbringen.

Bevor wir uns einmal verschiedene Methoden des `UIApplicationDelegate`-Protokolls ansehen, zeige ich Ihnen in Bild 23.23, welche Ereignisse im Lebenszyklus einer iOS-App typischerweise auftreten. Im Folgenden finden Sie eine Beschreibung der verschiedenen Zustände, die eine iOS-App während ihres Lebenszyklus annehmen kann:

- *Not running:* Die App wird nicht ausgeführt und ist auch nicht im Hintergrund aktiv.
- *Inactive:* Die App wird ausgeführt und befindet sich im Vordergrund (sprich sie wird auf dem Bildschirm angezeigt), nimmt augenblicklich aber keine Ereignisse wie Touch-Eingaben entgegen. Dieser Status kann beispielsweise eintreten, wenn ein Telefonanruf während der Verwendung der App eingeht. Spiele können diesen Zustand nutzen, um das aktuelle Geschehen anzuhalten und zu pausieren.
- *Active:* Die App wird ausgeführt und befindet sich im Vordergrund. Sie kann in vollem Umfang verwendet werden.
- *Background:* Die App führt Code aus, befindet sich aber nicht im Vordergrund. Dieser Zustand ist beispielsweise aktiv, wenn die App vom Vordergrund durch Wechseln auf den Home-Bildschirm in den Hintergrund wechselt. Ihr steht dann eine kurze Zeitspanne zur Verfügung, um abschließende Befehle durchzuführen, ehe sie in den Suspended-Status wechselt (beispielsweise um noch nicht gesicherte Informationen zu speichern).

  Darüber hinaus gibt es noch weitere Situationen, in denen iOS eine App im Hintergrund starten kann, um sie Aktionen durchführen zu lassen (ohne dass die App dazu in den Vordergrund wechselt). Dazu gehören das Abschließen von im Hintergrund durchgeführten Downloads oder das Reagieren auf eine empfangene Push Notification. Mehr zu diesen spezifischen Hintergrundaktionen erfahren Sie an entsprechenden Stellen im Buch, die diese Themen behandeln.

- *Suspended:* Die App befindet sich noch im Speicher, führt aber keinen Code aus. Von diesem Status aus kann die App jederzeit vom System beendet werden, um Platz für andere Apps zu schaffen. Sie wechselt dann zurück in den ursprünglichen *Not running*-Status.

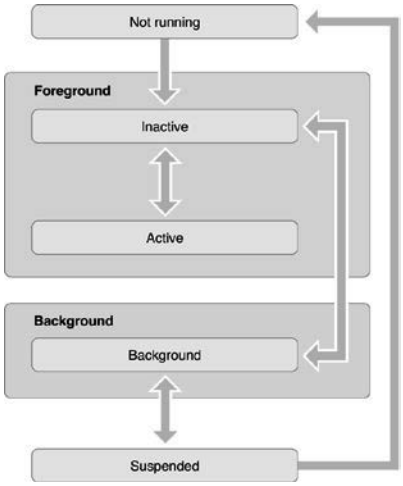


Bild 23.23
Der Lebenszyklus einer iOS-App (Bild: Xcode-Dokumentation).

Wie eingangs beschrieben, werden diese verschiedenen Zustände des Lebenszyklus einer iOS-App sowie der Wechsel zwischen ihnen in Form von Methoden abgebildet, die im `UIApplicationDelegate`-Protokoll definiert sind. Im Folgenden stelle ich Ihnen diese Methoden vor und erläutere, wann sie vom System aufgerufen werden:

- Start einer App
  - `application(_:willFinishLaunchingWithOptions:)`: Diese Methode wird aufgerufen, sobald eine App startet, der Startvorgang aber gerade erst begonnen hat und somit noch nicht abgeschlossen ist.
  - `application(_:didFinishLaunchingWithOptions:)`: Diese Methode wird aufgerufen, nachdem der Startvorgang einer App vollständig abgeschlossen ist.
- Wechsel in den Vordergrund
  - `applicationDidBecomeActive(_:)`: Diese Methode wird aufgerufen, sobald sich eine App im Vordergrund befindet und auf dem Bildschirm des iOS-Geräts angezeigt wird.
- Wechsel in den Inaktiv-Status
  - `applicationWillResignActive(_:)`: Diese Methode wird aufgerufen, wenn eine aktive App in den inaktiven Zustand wechselt.
- Wechsel in den Hintergrund
  - `applicationDidEnterBackground(_:)`: Diese Methode wird aufgerufen, sobald eine App vom Inaktiv-Status in den Hintergrund wechselt. Sie wird dann nicht länger auf dem Bildschirm des iOS-Geräts angezeigt.
- Wechsel zurück in den Vordergrund
  - `applicationWillEnterForeground(_:)`: Diese Methode wird aufgerufen, wenn sich eine App im Hintergrund befindet und währenddessen wieder zurück in den Vordergrund wechselt.

- Beenden einer App
  - `applicationWillTerminate(_:)`: Diese Methode wird aufgerufen, sobald eine App komplett beendet wird. Diese Methode wird nicht aus dem Suspended-Zustand heraus aufgerufen.

Betrachtet man einmal den Code der `AppDelegate`-Klasse eines neu erstellten iOS-Projekts in Xcode, so stellt man fest, dass für einen Großteil der hier aufgeführten Methoden bereits Platzhalter existieren (siehe Listing 23.6). Diese enthalten zusätzlich kurze Beschreibungstexte in Form von Kommentaren, die ebenfalls noch einmal erläutern, wofür diese Methoden gut sind und wann sie vom System aufgerufen werden.

Listing 23.6 Standardimplementierung der `AppDelegate`-Klasse

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state.
This can occur for certain types of temporary interruptions (such as an incoming
phone call or SMS message) or when the user quits the application and it begins the
transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate
graphics rendering callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate
timers, and store enough application state information to restore your application to
its current state in case it is terminated later.
        // If your application supports background execution, this method is called
instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        // Called as part of the transition from the background to the active state;
here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the
application was inactive. If the application was previously in the background,
optionally refresh the user interface.
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if
appropriate. See also applicationDidEnterBackground:.
    }

}
```

Sie können diese Platzhalter nutzen, um bei den gewünschten Methoden Ihre eigene Logik zu implementieren, oder – falls Sie sie (noch) nicht brauchen – sie einfach aus der `AppDelegate`-Klasse löschen. Eine fehlende Implementierung einer der beschriebenen Methoden teilt dem System lediglich mit, dass Sie bei einem Wechsel in den entsprechenden Zustand selbst keine zusätzlichen Befehle ausführen möchten (was vollkommen in Ordnung ist). Ich persönlich empfehle Ihnen, nur dann Methoden des `UIApplicationDelegate` in Ihrem App Delegate zu implementieren, wenn Sie sie auch tatsächlich benötigen und eigenen Code darin ausführen. Andernfalls können Sie den Code der `AppDelegate`-Klasse soweit reduzieren, dass nur noch die Deklaration und die `window`-Property übrig bleibt (siehe Listing 23.7).

Listing 23.7 Reduzierte Version der `AppDelegate`-Klasse

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

}
```

23.4.2 Die window-Property

Bei der Verwendung von Storyboards zum Laden des initialen View-Controllers wird automatisch eine Instanz der Klasse `UIWindow` erzeugt. Diese Klasse wird in iOS-Apps genutzt, um darin die Inhalte von View-Controllern anzuzeigen und auf dem Bildschirm eines iOS-Geräts einzublenden. Der `window`-Property des App Delegate wird hierbei automatisch jene `UIWindow`-Instanz zugewiesen, die durch die Verwendung von Storyboards vom System erzeugt wurde. Sie können somit diese Property nutzen, um auf die Informationen des `UIWindow` zuzugreifen.

In der Regel brauchen Sie sich in der iOS-Entwicklung nur wenig bis gar nicht mit der Klasse `UIWindow` zu beschäftigen, da die angezeigten Oberflächen ausschließlich über View-Controller und deren Logik abgebildet werden. Mehr zu View-Controllern erfahren Sie in Abschnitt 23.5, „UIViewController im Detail“.

23.4.3 Einsatzzweck des App Delegate

Theoretisch können Sie iOS-Projekte umsetzen, ohne auf die beschriebenen Funktionen des App Delegate zurückzugreifen. Wenn Sie sich weder für Statusänderungen beim Lebenszyklus Ihrer App interessieren, noch auf die `window`-Property zugreifen müssen, brauchen Sie die App Delegate-Klasse in Ihrem Projekt nicht weiter zu beachten. Die Hauptaufgabe des App Delegate besteht darin, Sie über Ereignisse ausgehend vom System zu informieren, und wenn Sie diese Information nicht benötigen, brauchen Sie sich auch keine Gedanken um eine mögliche Implementierung des App Delegate zu machen (und das ist dann auch vollkommen in Ordnung).

Aus eigener Erfahrung kann ich Ihnen aber sagen, dass in eigentlich jeder App wenigstens ein Teil der Methoden des `UIApplicationDelegate`-Protokolls benötigt wird. Sei es, dass

beim Starten der App einige grundlegende Einstellungen gesetzt oder beim Beenden ungesicherte Daten gespeichert werden sollen, oft führt für diese Aufgaben kein Weg am App Delegate vorbei. Welche Methoden Sie hierbei für Ihre individuellen Zwecke nutzen, hängt davon ab, bei welchem Ereignis Sie einschreiten und eigene Befehle implementieren wollen.

23.5 UIViewController im Detail

Die Klasse `UIViewController` gehört zu den wichtigsten Klassen in der iOS-Entwicklung. Wie der Name bereits andeutet, handelt es sich bei ihr um ein Controller-Element, das mit einer View verknüpft ist. Sie dient dazu, dem Nutzer eine Ansicht einer App zu präsentieren und deren Inhalte dynamisch anzupassen oder auf Nutzereingaben zu reagieren (beispielsweise die Betätigung eines Buttons). Die erste Ansicht, die dem Nutzer beim Starten einer App präsentiert wird, ist dementsprechend ebenfalls direkt ein solcher View-Controller.

Die Klasse `UIViewController` dient als Basisklasse für alle Ansichten, die dem Nutzer während der Verwendung einer App präsentiert werden. Es gibt noch diverse spezifischere Subklassen, die für spezielle Aufgaben ausgelegt sind (beispielsweise das Darstellen einer Tabelle oder die Umsetzung einer Navigationsstruktur); diese werden in separaten Abschnitten in Kapitel 24, „iOS – App-Entwicklung“, im Detail beleuchtet. An dieser Stelle soll es nur um die Klasse `UIViewController` und deren grundlegende Funktionsweise gehen, da das Verständnis darüber essenziell ist, um professionelle Apps für iOS entwickeln zu können.

23.5.1 Aufbau

Jede `UIViewController`-Instanz besitzt eine Property `view` vom Typ `UIView`. Hierbei handelt es sich um die eigentliche Ansicht, die der Nutzer zu sehen bekommt, wenn ein View-Controller geladen und angezeigt wird. Wir können die Hintergrundfarbe dieser View ändern oder sie um zusätzliche Subviews (sprich andere View-Elemente, die wir auf einer View platzieren) ergänzen. Letzteres haben wir beispielhaft in Abschnitt 23.3.3, „Hello World“, durchgeführt, als wir ein Label auf der View des View-Controllers platziert haben; das Label stellte dort eine Subview dar.

Views

Views sind in der iOS-Entwicklung ein Thema für sich. Mehr zu den verschiedenen Views, die in der Programmierung zur Verfügung stehen, sowie die Arbeit mit ihnen erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“.

Dieser Aufbau kann sehr schön im Interface Builder von Xcode nachvollzogen werden. Öffnen Sie dazu ein bestehendes iOS-Projekt oder erstellen Sie ein neues auf Basis einer *Single View App*. Greifen Sie darin anschließend auf die *Main.storyboard*-Datei zu und klicken Sie

den initialen View-Controller an einer beliebigen Stelle an. Werfen Sie nun einen Blick in die *Document Outline Area*. Es handelt sich hierbei um eine zusätzliche Ansicht im linken Bereich des Interface Builders. Sollte diese nicht angezeigt werden, können Sie sie über die Schaltfläche am unteren linken Rand des Editor-Fensterns ein- und ausblenden (siehe Bild 23.24).

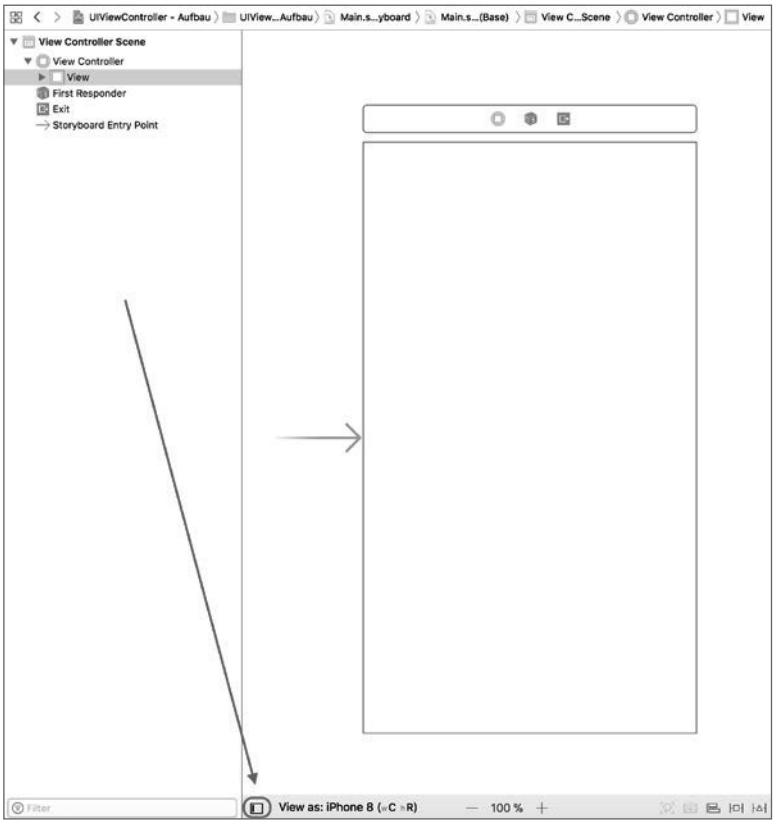


Bild 23.24 Die links angezeigte Document Outline Area kann über die Schaltfläche am unteren linken Rand des Editor-Fensters ein- und ausgeblendet werden.

In der Document Outline Area sehen Sie nun den Aufbau des ausgewählten View-Controllers (siehe Bild 23.25). Hier ist zu sehen, dass jeder View-Controller ein View-Element besitzt. Bei der View handelt es sich standardmäßig um die große weiße Fläche, die zusammen mit dem View-Controller im Storyboard angezeigt wird. Das bestätigt auch ein Blick in die Inspectors Area und den Identity Inspector, nachdem man die weiße Fläche mit der linken Maustaste angeklickt hat. Dort findet sich dann ganz oben im Feld *Class* die Information, dass es sich bei diesem Element um eine `UIView` handelt (siehe Bild 23.26).

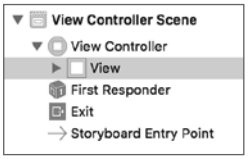


Bild 23.25
Die Document Outline Area zeigt den Aufbau eines View-Controllers. Hier ist zu sehen, dass eine View direkt dem View-Controller zugeordnet ist.

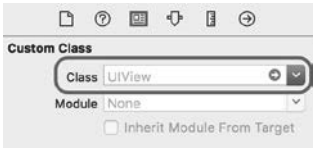


Bild 23.26
Die Inspectors Area verrät, dass es sich bei dem im Interface Builder angezeigten Element eines View-Controllers um eine UIView handelt.

Die Verknüpfung dieser `UIView`-Ansicht im Interface Builder mit der `view`-Property der `UIViewController`-Klasse kann ebenfalls konkret nachvollzogen werden. Dazu klickt man auf das gelbe Symbol für den View-Controller (entweder in der Symbolleiste über der View oder in der Document Outline Area) und wechselt anschließend in den Connections Inspector der Inspectors Area. Dort ist im Abschnitt *Outlets* zu sehen, dass die View mit der `view`-Property des View-Controllers verknüpft ist (siehe Bild 23.27).

Bild 23.27
Der Connections Inspector eines View-Controllers verdeutlicht die Verknüpfung mit der angezeigten View.

Der View-Controller kümmert sich darum, dass beim Laden und Einblenden die zugrunde liegende View mit all ihren Subviews auf dem Bildschirm angezeigt wird. Sobald ein View-Controller vollständig geladen ist, wird die Methode `viewDidLoad()` der `UIViewController`-Klasse aufgerufen.

In vielen Szenarien in der iOS-Entwicklung wird diese Methode in eigenen `UIViewController`-Subklassen überschrieben, um zusätzliche Initialisierungen im Zusammenspiel mit dem View-Controller durchzuführen (beispielsweise das Übersetzen von einem auf der View angezeigten Label oder das Laden aktueller Inhalte von einem Webservice). Die Methode wird auch standardmäßig beim Erstellen einer neuen `UIViewController`-Subklasse in Xcode hinzugefügt, wie in Listing 23.4 in Abschnitt 23.3.2, „Rundgang durch die erstellten Dateien“, bereits zu sehen war.

Mehr über die Methode `viewDidLoad()` und den Lebenszyklus eines View-Controllers erfahren Sie in Abschnitt 23.5.4, „Lebenszyklus eines View-Controllers“.

View-Controller programmatisch erstellen

In der Regel werden View-Controller auf Basis einer Nib- oder Storyboard-Datei erstellt. Ist das der Fall, wird der `view`-Property des View-Controllers automatisch die View aus der entsprechenden Interface-Datei zugewiesen.

Möchte man aber stattdessen einen View-Controller komplett im Code erzeugen, ohne dass es für diesen eine zugehörige Interface-Datei gibt, ist es wichtig, in der Implementierung dieses View-Controllers die Methode `loadView()` zu überschreiben. Diese Methode wird automatisch aufgerufen, sobald ein View-Controller versucht, auf seine `view`-Property zuzugreifen und dabei feststellt, dass diese noch keinen Wert besitzt. Selbst aufrufen sollte man die Methode `loadView()` niemals.

In der eigenen `UIViewController`-Subklasse erzeugt man sodann innerhalb der Methode `loadView()` die gewünschte `UIView`-Instanz, die im Zusammenspiel mit dem View-Controller angezeigt werden soll, und weist sie der `view`-Property zu. Der Aufruf von `super` innerhalb von `loadView()` sollte vermieden werden.

Mehr über Views in der iOS-Entwicklung erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView".

23.5.2 Ansicht eines View-Controllers anpassen

Die Ansicht, die jeder View-Controller mit sich bringt, kann individuell angepasst werden. Im einfachsten Fall geschieht diese Gestaltung über eine Interface-Datei wie ein Storyboard. Darüber kann die Ansicht mit allen Elementen in der gewünschten Anordnung kreiert werden.

Herzstück hierbei ist die *Objects Library* von Xcode. Sie lässt sich über eine gleichnamige Schaltfläche am oberen rechten Rand von Xcode einblenden (siehe Bild 23.28).

Innerhalb der Objects Library finden sich Vorlagen für verschiedene View-Controller und Views, die in einem Interface platziert werden können. Um ein Element dem Interface hinzuzufügen, ziehen Sie es einfach aus der Objects Library auf die gewünschte Stelle. Auf diese Art und Weise können Sie Ihren View-Controller beispielsweise um Schaltflächen, Schalter, Labels, Textfelder und vieles mehr ergänzen.

In der Objects Library wird primär zwischen zwei Arten von Interface-Elementen unterschieden:

- *View-Controller:* Diese Elemente werden zu Beginn innerhalb der Objects Library aufgelistet und sind in Form von gelbfarbigen Elementen umgesetzt. Der initiale View-Controller einer Storyboard-Datei ist beispielsweise ein solches Element. Sie dienen dazu, in einem Interface mehr als einen View-Controller zu gestalten, wofür für jeden zu gestaltenden View-Controller ein entsprechendes View-Controller-Element im Interface platziert werden muss.

  View-Controller können ausschließlich auf einer freien Fläche innerhalb der Interface-Datei und nicht innerhalb eines bestehenden View-Controllers platziert werden.
- *Views:* Mit diesen Elementen gestalten Sie das Aussehen eines View-Controllers. Sie können ausschließlich in der freien Fläche eines View-Controllers und an sonst keiner anderen Stelle einer Interface-Datei platziert werden. Der Großteil der Elemente in der Objects Library gehört zu dieser Kategorie. Dazu zählen beispielsweise Labels, Schaltflächen und Slider.

Views

Mehr zu den verschiedenen in iOS verfügbaren Views und ihrer Funktionsweise erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView".

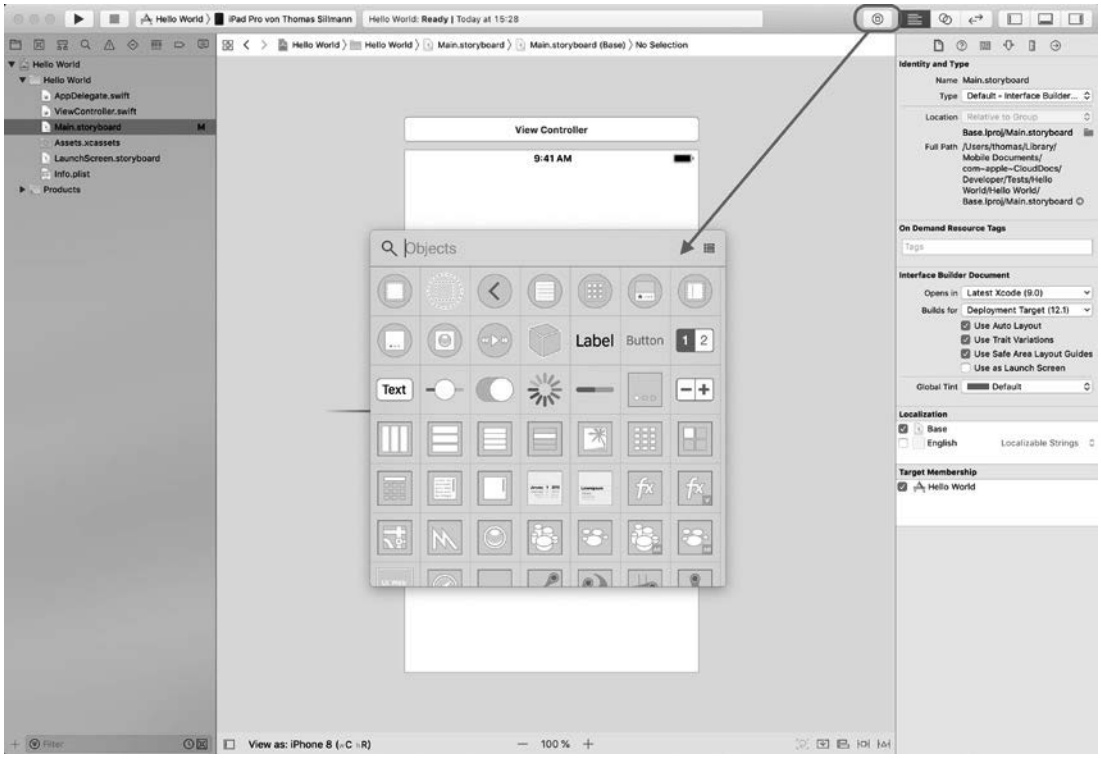


Bild 23.28 Die Objects Library erreicht man über die gekennzeichnete Schaltfläche am oberen rechten Rand von Xcode.

Festlegen der View-Positionierung mit Auto Layout

Mithilfe von *Auto Layout* legen Sie fest, wie die Views eines View-Controllers positioniert werden sollen. Das ist unter iOS deshalb so wichtig, weil es eine Vielzahl unterschiedlicher Geräte mit verschiedenen Displaygrößen und Seitenverhältnissen gibt. Möchten Sie beispielsweise ein Label mittig platzieren, befindet sich der Mittelpunkt bei einem iPhone X an einer anderen Stelle als bei einem iPhone SE oder einem iPad Pro (siehe Bild 23.29). Hinzu kommt die Möglichkeit, iOS-Geräte drehen und so zwischen einem Portrait- und Landscape-Modus wechseln zu können. Auch hier ändert sich das zugrunde liegende Koordinatensystem, in dem eine Ansicht in iOS aufgebaut wird, und die Mitte rückt an eine andere Position.

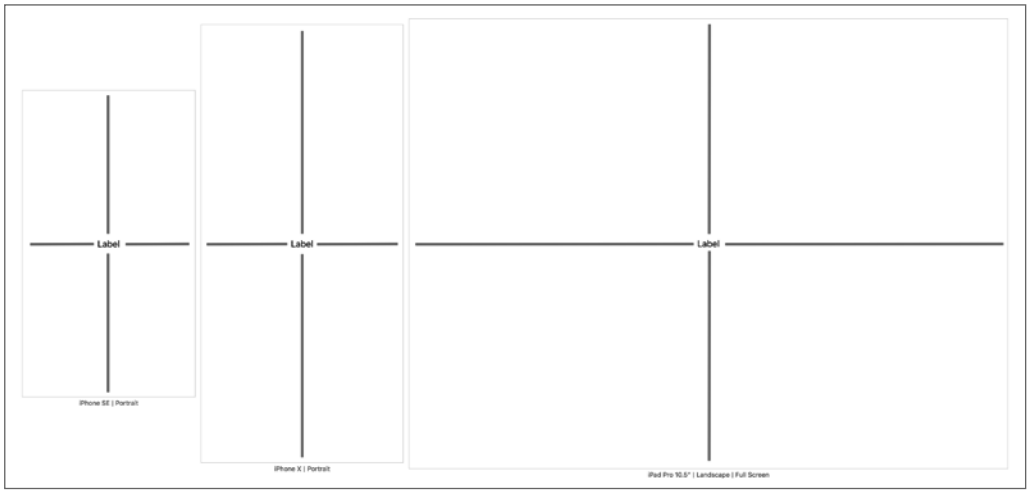


Bild 23.29 Mitte ist nicht gleich Mitte; je nach Device und Orientierung ändern sich die Abstände zu einer bestimmten Position auf dem Display.

Das Koordinatensystem in iOS

Views werden mithilfe von X- und Y-Koordinaten platziert. Hierbei ist zu beachten, dass die Y-Achse unter iOS am oberen Rand beginnt. Eine View mit der Y-Koordinate 0 wird somit ganz oben platziert (siehe Bild 23.30).

Bild 23.30
Im Koordinatensystem von iOS beginnt die Y-Achse am oberen Rand, nicht am unteren.

Auto Layout hilft dabei, für jede einzelne View Regeln für die korrekte Positionierung festzulegen, um sie auf unterschiedlichen Geräten in unterschiedlichen Ausrichtungen immer korrekt anzuzeigen. Dazu kommen sogenannte *Constraints* zum Einsatz. Ein Constraint entspricht einer solchen Regel, zum Beispiel: Richte die View horizontal zentriert aus. Zusammen mit einem zweiten Constraint zum vertikalen Zentrieren einer View hat man die nötigen Regeln definiert, um ein Element mittig zu platzieren. Es gibt noch eine Vielzahl weiterer Regeln für Constraints. Dazu gehören:

- Festlegen der Abstände einer View zum oberen, unteren, linken und rechten Rand
- Festlegen von Breite und Höhe einer View
- Festlegen des Abstands zu einem umgebenden View-Element

Wie Sie solche Auto Layout-Constraints in Storyboards setzen, soll anhand mehrerer praktischer Beispiele demonstriert werden. Das Grundvorgehen ist hierbei immer identisch. Sobald Sie also das grundlegende Prinzip einmal verstanden haben, können Sie selbst mit den verschiedenen Arten von Constraints experimentieren, um Ihre Views optimal auszurichten.

Beginnen wir zunächst mit dem Erstellen eines neuen Xcode-Projekts auf Basis einer *Single View App*. Anschließend rufen wir die Datei *Main.storyboard* auf und platzieren dort ein Label am oberen rechten Rand. Die blauen Hilfslinien helfen dabei, die optimale Position für das Label zu finden (siehe Bild 23.31).

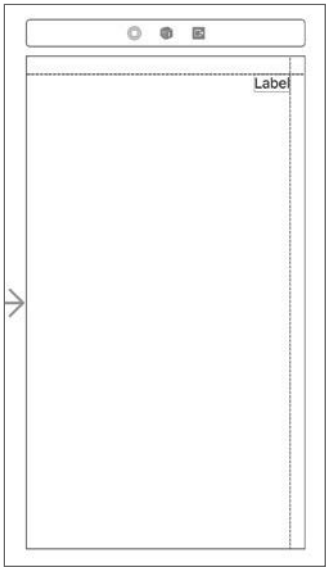


Bild 23.31
Dem initialen View-Controller der Main.storyboard-Datei wird ein Label am oberen rechten Rand hinzugefügt.

Dieses Label ist nun ideal für das Gerät positioniert, das als Basis für das Layout des Storyboards verwendet wird. Welches Gerät das ist, können sie am unteren linken Rand des Editors sehen. Dort steht die Meldung *View as*, gefolgt vom entsprechenden Gerätenamen (siehe Bild 23.32).

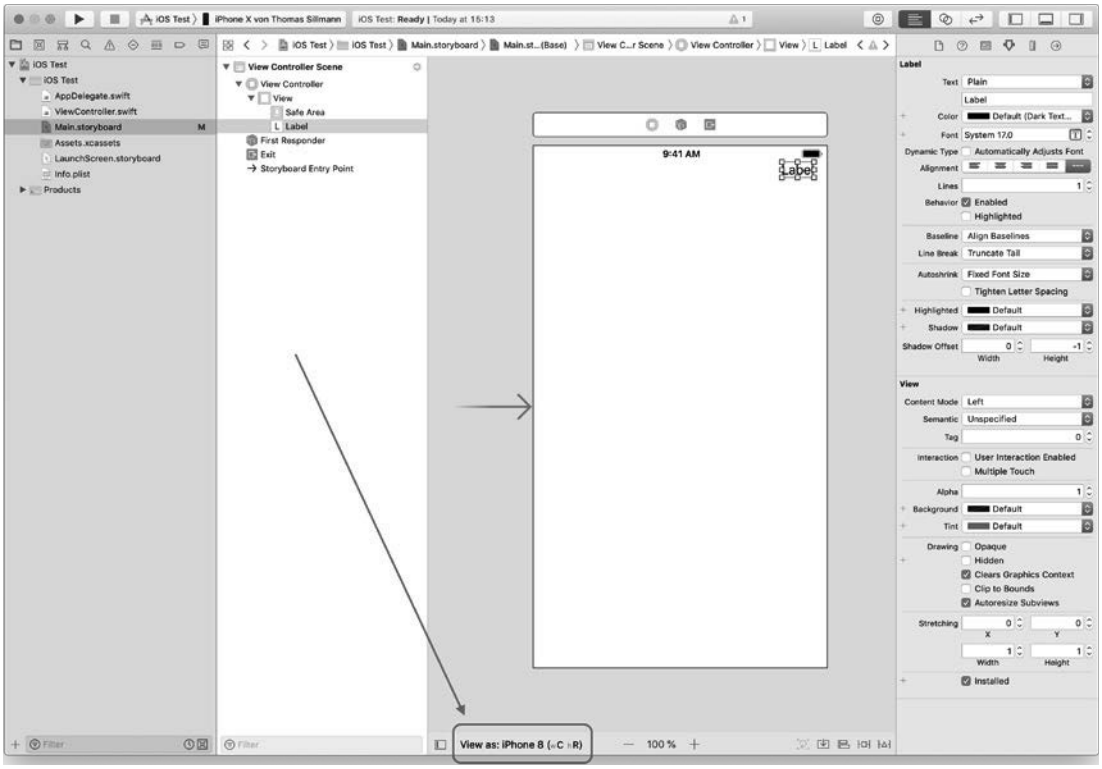


Bild 23.32 Am unteren Rand der Editor-Ansicht können Sie sehen, welches iOS-Gerät als Basis für die View-Controller des angezeigten Storyboards dient; in diesem Fall handelt es sich um das iPhone SE.

Führen Sie die App nun in dem für dieses Gerät passenden Simulator aus, werden Sie feststellen, dass das eingefügte Label perfekt und wie gewünscht am oberen rechten Rand angezeigt wird. Das liegt daran, dass die für die View festgelegte Position für das Gerät optimiert ist. Ein Test in einem anderen Simulator für ein Gerät mit einem anderen Display dürfte hingegen schnell Ernüchterung bringen; dort wird das Label sich entweder zu sehr Richtung Mitte oder sogar außerhalb des sichtbaren Bereichs befinden (siehe Bild 23.33).

Doch warum ist das so? Bis jetzt wurde das Label schlicht an einer bestimmten Stelle im Interface Builder platziert. Wird nun die entsprechende Ansicht auf einem Endgerät geladen, werden exakt die für die Platzierung des Elements verwendeten X- und Y-Koordinaten zur Positionierung genutzt. In dem gezeigten Beispiel ist der Wert der X-Achse das Problem. Während auf einem kleinen iPhone SE ein geringer Wert bereits dafür sorgt, dass eine View am rechten Rand angezeigt wird, braucht es bei einem größeren Gerät wie dem iPhone 8 Plus schon einen deutlichen höheren Wert.

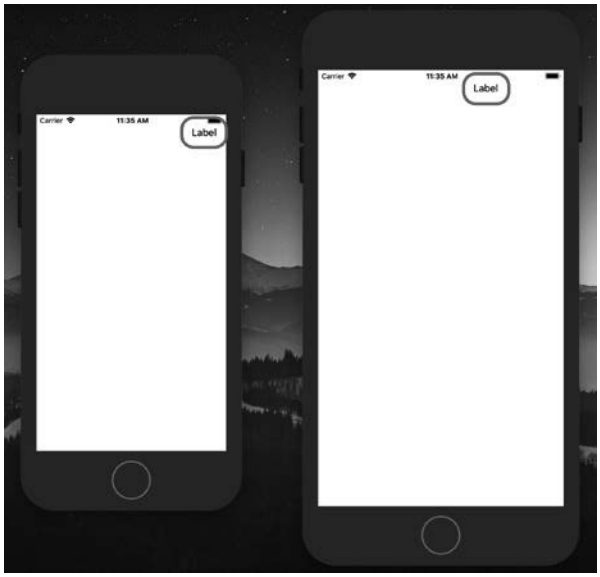


Bild 23.33 Im iPhone SE-Simulator wird das platzierte Label perfekt angezeigt, im iPhone 8 Plus-Simulator hingegen ist der Abstand zum rechten Rand deutlich größer.

Größe und Position einer View anzeigen und bearbeiten

Wählt man eine View im Interface Builder aus, lassen sich Informationen zu Größe und Position des Elements in der Inspectors Area auslesen und verändern. Dazu ruft man über die entsprechende Schaltfläche den sogenannten *Size Inspector* auf (siehe Bild 23.34). Dort finden sich im Abschnitt *View* die gesetzten Werte für die X- und Y-Koordinaten sowie die Breite und Höhe.

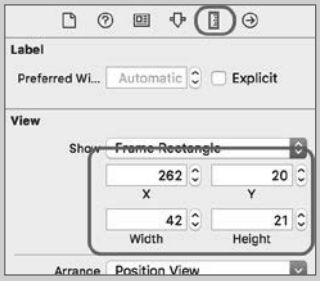


Bild 23.34
Größe und Position einer View können im Size Inspector eingesehen und bearbeitet werden.

Möchte man die Werte für diese Eigenschaften punktgenau festlegen, bietet es sich an, sie direkt an den entsprechenden Stellen im Size Inspector einzutragen anstatt die View möglicherweise aufwendig im View-Controller zu verschieben oder mithilfe der Maus die Größe zu ändern.

An dieser Stelle kommt das sogenannte *Auto Layout* ins Spiel, um das beschriebene Problem zu lösen. Es muss für das Label eine Regel definiert werden, die besagt, dass der Abstand zum rechten Rand immer gleich groß sein soll. Das sorgt dafür, dass das Label an diesem Rand „kleben" bleibt und nicht – wie im iPhone 8 Plus-Simulator in Bild 23.33 zu sehen – mitten in der Ansicht schwebt.

Um einen passenden Constraint für diese Regel zu setzen, gehen Sie wie folgt vor: Wählen Sie das Label im Interface Builder aus und ziehen Sie anschließend von dort mit gedrückt gehaltener rechter Maustaste eine Verbindung zum rechten Rand der zugrunde liegenden View, so wie in Bild 23.35 zu sehen. Lassen Sie nun die rechte Maustaste los, sobald die zugrunde liegende View blau hervorgehoben wird und ein Pop-up-Menü erscheint (siehe Bild 23.36).

Bild 23.35
Ziehen Sie eine Verbindung von einer View zu einem anderen View-Element (in diesem Fall die zugrunde liegende View des View-Controllers), um Auto Layout-Constraints zu setzen.

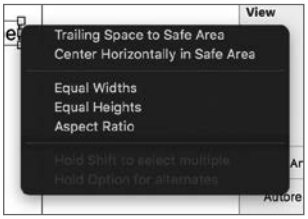


Bild 23.36
Über das Pop-up-Menü können Sie einen passenden Constraint auswählen.

Dieses Pop-up-Menü bietet Ihnen verschiedene Constraints, die Sie im Zusammenspiel zwischen der View, mit dem die gezogene Verbindung gestartet wurde (dem Label), und der View, mit der die Verbindung beendet wurde (der zugrunde liegenden View des View-Controllers), setzen können. Dabei wird auch die gewählte Richtung berücksichtigt. Da die Verbindung zum rechten Rand der View des View-Controllers erfolgte, beziehen sich die angezeigten Constraints auch primär auf genau diesen rechten Rand.

In diesem Pop-up-Menü steht beispielsweise an erster Stelle der Punkt *Trailing Space to Safe Area* zur Verfügung. Wird dieser ausgewählt, wird der aktuelle Abstand vom Label zum rechten Rand der zugrunde liegenden View als fix definiert, sodass das Label immer genau diesen Abstand zum rechten Rand besitzt; genau das, was wir in der aktuellen Situation benötigen! Wenn Sie diesen Punkt auswählen, verschwindet das Pop-up-Menü und der entsprechende Constraint wird gesetzt. Diesen erkennen Sie an der blauen Linie, die erscheint, sobald Sie das Label im Interface Builder auswählen (siehe Bild 23.37).

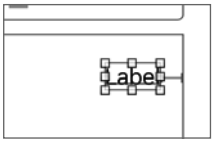


Bild 23.37
Die blaue Linie, die vom rechten Rand des Labels ausgeht, stellt den soeben gesetzten Constraint dar.

Auf diese Art und Weise lassen sich dem Label weitere Constraints hinzufügen, beispielsweise durch Ziehen einer Verbindung an den *oberen* Rand und Auswahl des Punkts *Top Space to Safe Area*. Das legt fest, dass der Abstand des Labels zum oberen Rand ebenfalls immer gleich groß bleibt.

Mit diesen beiden Regeln ist sichergestellt, dass das Label immer am oberen rechten Rand angezeigt wird, egal wie groß oder klein das Display ist, auf dem die App ausgeführt wird (siehe Bild 23.38).

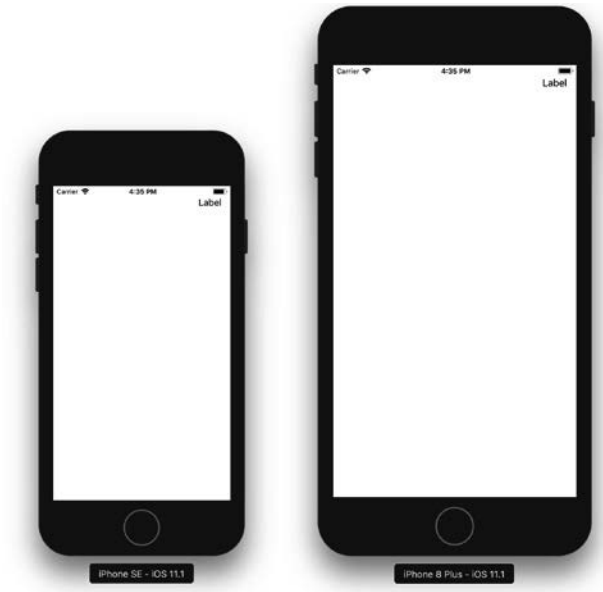


Bild 23.38 Das Label wird nach Setzen der Auto Layout-Constraints korrekt in der oberen rechten Ecke angezeigt, egal bei welcher Display-Größe und -Orientierung.

Safe Area

Die sogenannte *Safe Area* spielt seit der Einführung des iPhone X in iOS eine wichtige Rolle. Da dessen Display über abgerundete Ecken verfügt und somit die Gefahr besteht, dass View-Elemente wie Labels in diesen abgerundeten Bereichen untergehen, muss eine Lösung her, die derartige Probleme verhindert. Hierzu definiert die Safe Area für das iPhone X-Display einen viereckigen Bereich, innerhalb dessen man problemlos View-Elemente platzieren kann (siehe Bild 23.39). Durch das Setzen von Constraints in Bezug auf diese Safe Area stellt man sicher, dass die View-Elemente auch auf einem iPhone X mit abgerundetem Display korrekt angezeigt werden. Bild 23.40 verdeutlicht das hierbei zugrunde liegende Problem.

Bild 23.39 Die Safe Area ist ein von Apple für das Display eines iOS-Geräts festgelegter optimaler Bereich, in dem View-Elemente platziert werden sollten. Sie stellt sicher, dass gerade auf Geräten mit abgerundetem Display wie dem iPhone X alle Views korrekt platziert und nicht aus Versehen in den Ecken abgeschnitten werden.

Bild 23.40 Werden View-Elemente auf einem Gerät mit abgerundetem Display wie dem iPhone X nicht an der Safe Area ausgerichtet, verschieben sie sich möglicherweise in die Randbereiche, wo sie meist schlecht lesbar sind oder abgeschnitten werden.

Ob die Safe Area bei der Arbeit mit Auto Layout zum Einsatz kommt, verrät die Checkbox *Safe Area Layout Guide* im Size Inspector der Inspectors Area. Um deren aktuellen Status zu sehen, klicken Sie auf die zugrunde liegende View des gewünschten View-Controllers, für den Sie den Wert der Checkbox prüfen möchten, und rufen anschließend den Size Inspector auf. Ist der Haken für *Safe Area Layout Guide* gesetzt, wird die Safe Area beim Setzen von Constraints berücksichtigt, andernfalls nicht (siehe Bild 23.41). Bei neuen Projekten ist dieser Haken standardmäßig aktiviert.

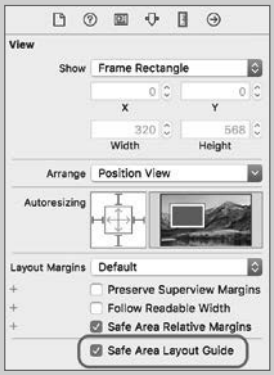


Bild 23.41
Die Checkbox „Safe Area Layout Guide“ bestimmt, ob beim Setzen von Constraints für einen View-Controller die Safe Area berücksichtigt wird oder nicht.

In der Regel sollten Sie immer Gebrauch von der Safe Area machen. Das Verhalten der Constraints bei iOS-Geräten ohne abgerundetem Display bleibt unverändert, gleichzeitig unterstützen Sie mit Beachtung der Safe Area auch direkt neue Geräte wie das iPhone X.

Das nächste Beispiel soll die Positionierung eines View-Elements in der Mitte eines iOS-Geräts demonstrieren. Fügen Sie dazu einem View-Controller erneut ein Label hinzu und fixieren Sie es mithilfe der blauen Linien genau in der Mitte der Ansicht (siehe Bild 23.42). Ziehen Sie anschließend eine Verbindung vom Label zum oberen Rand, bis die zugrunde liegende View des View-Controllers wieder blau hervorgehoben wird. Wenn Sie nun die rechte Maustaste loslassen, erscheint erneut das Pop-up-Menü zum Setzen von Constraints. Wählen Sie dort den Punkt *Center Vertically in Safe Area* aus (siehe Bild 23.43). Ziehen Sie anschließend eine Verbindung vom Label zum rechten oder linken Rand der zugrunde liegenden View des View-Controllers, lassen Sie dann die rechte Maustaste los und wählen Sie aus dem Pop-up-Menü *Center Horizontally in Safe Area* aus.

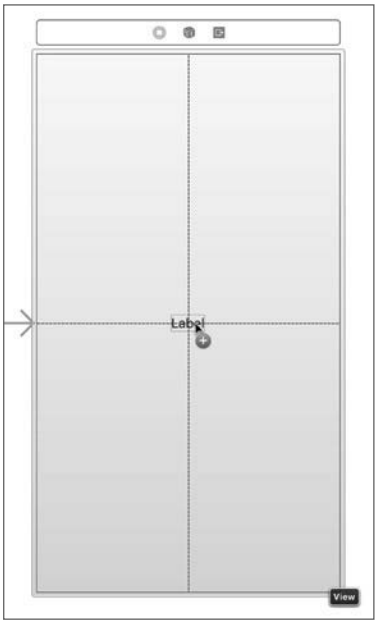


Bild 23.42
Platzieren Sie ein Label genau in der Mitte der View eines View-Controllers.

Bild 23.43
Mithilfe der Center-Constraints können Sie View-Elemente mittig platzieren.

Wenn Sie die App nun auf verschiedenen iOS-Geräten und Display-Ausrichtungen testen, werden Sie feststellen, dass das Label immer zentriert an der von Ihnen definierten Stelle angezeigt wird (siehe Bild 23.44).

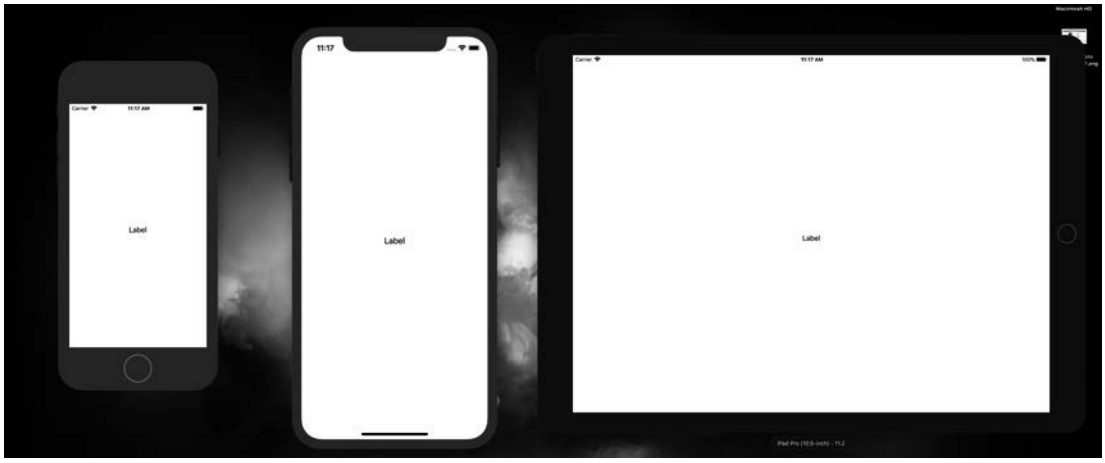


Bild 23.44 Egal ob iPhone SE, iPhone X oder iPad Pro im Landscape-Modus: Das Label wird immer zentriert auf dem Bildschirm angezeigt.

Auf die beispielhaft skizzierte Art und Weise können Sie allen Views eines View-Controllers passende Constraints zuweisen, um zu definieren, wie diese zueinander stehen und positioniert werden sollen. Die Möglichkeiten hierbei sind vielfältig. Beispielsweise lässt sich über Constraints auch definieren, dass verschiedene View-Elemente immer die gleiche Höhe oder Breite besitzen sollen.

Mehr über Auto Layout, den Umgang mit Constraints und die verschiedenen zur Verfügung stehenden Möglichkeiten zur Positionierung von Views erfahren Sie in Kapitel 28, „Cross-Platform".

Übung macht den Meister

Ich empfehle Ihnen, mit Auto Layout und den verschiedenen Constraints in Ruhe zu experimentieren und sich dabei immer eine spezifische Aufgabe zu stellen. Betrachten Sie eine View und überlegen Sie, wie diese positioniert werden und sich im Verhältnis zu den anderen View-Elementen des entsprechenden View-Controllers verhalten soll. Planen Sie aufgrund dessen die passenden Constraints und setzen Sie diese anschließend um.

Mehr Informationen zu Auto Layout erhalten Sie in Kapitel 28, „Cross-Platform".

23.5.3 Verbindung zwischen Interface und Code

Das Erstellen und Gestalten der Interfaces einer App ist selbstredend eine wichtige Aufgabe bei der Entwicklung von iOS-Apps. Das Interface allein nützt aber in der Regel nicht viel, wenn es nicht mit einer passenden Logik – sprich dem *Code* – gekoppelt wird.

Doch was bedeutet diese Kopplung, und wozu ist sie gut? Nehmen wir als Beispiel ein Label, das Sie – wie in den Beispielen aus den vorherigen Abschnitten gezeigt – einem View-Controller hinzufügen. Dieses Label soll eine Info über einen angestoßenen Download ausgeben, indem es den Download-Fortschritt in Prozent anzeigt.

Bisher haben wir Views wie das Label lediglich einem View-Controller hinzugefügt und dort platziert, ohne diese Elemente dynamisch zu verändern. So behält ein Label die gesamte Zeit über den Text, den wir ihm im Storyboard zugewiesen haben. Die Anwendungen enthielten bisher keine zugehörige Logik, mit der man auf ein solches Label vom Code aus zugreift und dynamisch den gewünschten Wert setzt. Genau ein solches dynamisches Vorgehen macht aber jede App aus und ist der Grundstein für die Programmierung von Apps.

Um also ein statisches Interface wie einen in einer Storyboard-Datei kreierten View-Controller dynamisch über den Code konfigurieren und anpassen zu können, müssen diese beiden Elemente – Interface und Code – miteinander verbunden werden. Damit solch eine Verbindung möglich ist, braucht es zwei Dinge:

- `UIViewController`-*Subklasse:* Sie benötigen eine eigene Subklasse von `UIViewController`, die Ihren eigenen Code enthält und über die Sie das Verhalten Ihrer eigenen View-Controller bestimmen.
- *Interface:* Mithilfe eines Interfaces eines View-Controllers, wie Sie es beispielsweise in einem Storyboard erstellen können, gestalten Sie das Aussehen und den Aufbau eines View-Controllers und bestimmen, welche View-Elemente er enthält und anzeigt.

Haben Sie ein neues iOS-Projekt auf Basis einer *Single View App* erstellt, sind diese beiden Elemente bereits vorhanden. In der Datei *ViewController.swift* ist eine `UIViewController`-Subklasse `ViewController` definiert, die *Main.storyboard*-Datei enthält das zugehörige Interface für diesen View-Controller.

Doch wie genau sind diese beiden Elemente nun miteinander verbunden? Das verrät ein Blick in die *Main.storyboard*-Datei. Wählen Sie dort den initialen View-Controller aus, indem Sie in die obere Leiste und anschließend auf das runde gelbe Symbol links außen klicken (siehe Bild 23.45).

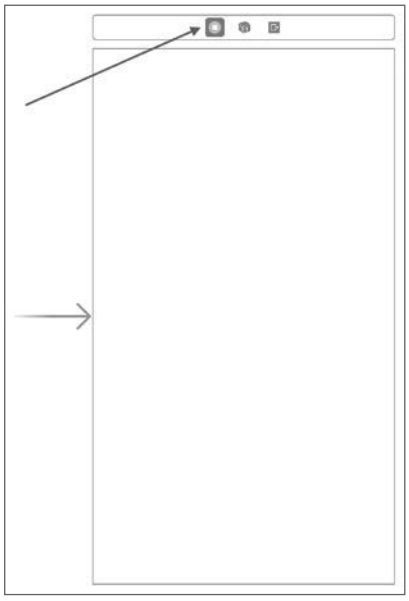


Bild 23.45
Einen View-Controller wählen Sie im Storyboard über das runde Symbol am oberen linken Rand aus.

Wechseln Sie dann in den Identity Inspector der Inspectors Area. Dort finden Sie ein Textfeld mit dem Titel *Class*. Darin wird die Klasse eines Elements aus dem Storyboard definiert. Da Sie den View-Controller ausgewählt haben, zeigt Ihnen der Identity Inspector in diesem Feld entsprechend den Namen der zugehörigen View-Controller-Klasse an. Dort ist *ViewController* eingetragen (siehe Bild 23.46).

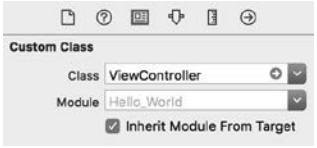


Bild 23.46
Der Identity Inspector verrät, welche Klasse dem ausgewählten Storyboard-Element zugewiesen ist.

Durch die Information im Textfeld *Class* wird die Verbindung zwischen Interface (dem View-Controller, den Sie in der Storyboard-Datei ausgewählt haben) und dem Code (jener Klasse, die Sie im Identity Inspector für das ausgewählte Element eintragen) hergestellt. Bei neuen Projekten auf Basis einer Single View App nimmt Xcode also bereits die Arbeit der Kopplung von Interface und Code für uns ab, indem es die `ViewController`-Klasse aus der *ViewController.swift*-Datei dem initialen View-Controller im Main-Storyboard zuweist.

Wenn Sie selbst für eine neue Ansicht eine weitere `UIViewController`-Subklasse sowohl im Code als auch im Storyboard erstellen, müssen Sie das beschriebene *Class*-Feld nutzen, um die beiden Elemente miteinander zu koppeln. Fehlt diese Verbindung, kann man im Code nicht auf die im Storyboard erstellten Views des View-Controllers zugreifen und das Interface verfügt umgekehrt über keine Logik, um dynamische Befehle auszuführen.

Die folgenden Abschnitte 23.5.3.1, „Outlets“, und Abschnitt 23.5.3.2, „Actions“, zeigen, was Ihnen die Verbindung von Interface und Code konkret bringt und wie Sie diese Kopplung in Ihren Apps nutzen können.

Das Interface ist optional

View-Controller müssen nicht zwingend über ein Storyboard gestaltet werden, sondern können auch vollständig im Code erstellt werden. Views wie Labels oder Buttons können also auch direkt im Code einem View-Controller hinzugefügt werden, ohne dass dafür eine Gestaltung über ein Storyboard zwingend notwendig ist (so wie es in den bisherigen Beispielen dieses Kapitels gezeigt wurde).

Wenn Sie einen View-Controller auf diese Art und Weise vollständig im Code erstellen und gestalten, braucht es selbstredend auch nicht die beschriebene Kopplung mit einem Interface im Storyboard; das gibt es dann schließlich nicht. Mehr über das programmatische Generieren von Views erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“.

23.5.3.1 Outlets

Ein *Outlet* beschreibt die Verbindung einer View aus einer Interface-Datei (wie einem Storyboard) mit dem Code. Eine solche Kopplung sorgt dafür, dass Sie von Ihrem Code aus auf die View zugreifen, sie auslesen und verändern können.

Betrachten wir als Beispiel ein Label, dass Sie in einer Storyboard-Datei einem View-Controller hinzugefügt haben. Das Label ist damit Teil des View-Controllers und wird bei Ausführung der App auch angezeigt, aber Sie haben keine Möglichkeit, dieses Label im Code Ihrer zugehörigen View-Controller-Klasse (siehe den vorherigen Abschnitt 23.5.3, „Verbindung zwischen Interface und Code“) anzusprechen, um beispielsweise den Text oder die Farbe zu ändern; dafür brauchen Sie ein Outlet.

Um ein Outlet zu erstellen, haben Sie zwei Möglichkeiten. Ich möchte Ihnen zu Beginn die „aufwendigere“ Variante vorstellen, da durch sie deutlicher wird, wie diese Kopplung von View und View-Controller zustande kommt.

Demonstrieren möchte ich dieses Vorgehen direkt an einem Beispiel auf Basis einer neuen *Single View App*. Fügen Sie zunächst dem initialen View-Controller in der *Main.storyboard*-Datei ein Label hinzu und positionieren Sie es an einer beliebigen Position (siehe Bild 23.47). Ziel dieser Beispiel-App soll es sein, beim Laden des View-Controllers den Text des Labels in „Outlet“ zu ändern.

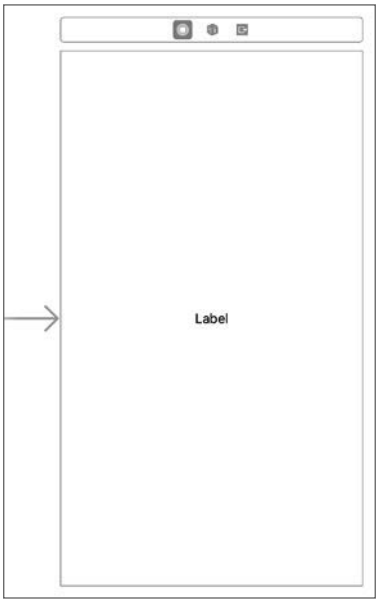


Bild 23.47
Dem initialen View-Controller der Beispiel-App fügen Sie ein Label hinzu, für das ein Outlet im Code erstellt werden soll.

Damit das gelingt, müssen wir im Code der `ViewController`-Klasse auf das Label zugreifen können; wir benötigen also ein *Outlet* des Labels. Um ein solches Outlet zu erstellen, deklariert man zunächst eine Property vom Typ der Outlet-View. In diesem Beispiel geht es um ein Label, dass dem Typ `UILabel` entspricht (mehr zu den verfügbaren Views in iOS und deren Klassen erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“).

Beachten Sie bei der Deklaration einer solchen Outlet-Property zwei Dinge:

- Weisen Sie das Schlüsselwort `weak` für eine Weak-Reference zu. Jeder View-Controller besitzt eine `view`-Property, bei der es sich um eine Strong-Reference handelt (siehe hierzu auch Abschnitt 23.5.1, „Aufbau“). Da jede weitere View, die Sie einem View-Controller in einem Storyboard hinzufügen, der `view`-Property als sogenannte Subview hinzugefügt wird, brauchen Sie diese nicht nochmals stark zu referenzieren. Solange die `view`-Property existiert (und das tut sie, solange auch der View-Controller aktiv ist), existieren auch alle ihre Subviews (mehr zu Views und Subviews erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“).
- Deklarieren Sie die Property als Implicitly Unwrapped Optional. Der Grund hierfür ist, dass bei Initialisierung eines View-Controllers die zugrunde liegende View noch nicht erstellt ist und somit `nil` entspricht. Das Gleiche gilt auch für alle Subviews, die Sie über das Storyboard einem View-Controller hinzufügen. Erst mit Erstellen der View werden auch die Subviews erzeugt und stehen dann uneingeschränkt zur Verfügung; darum die Deklaration als Implicitly Unwrapped Optional.

In Listing 23.8 sehen Sie die passende Implementierung der `ViewController`-Klasse aus der *ViewController.swift*-Datei (der von Xcode erzeugte Standard-Code der Klasse wurde von mir entfernt, da er aktuell nicht gebraucht wird). Sie enthält die Deklaration einer `UILabel`-Property namens `label`, die als Verbindung zu dem Label aus dem Storyboard dienen soll. Wie eben beschrieben, ist sie mit dem Schlüsselwort `weak` sowie als Implicitly Unwrapped Optional deklariert.

Listing 23.8 Deklaration einer Property für ein geplantes `UILabel`-Outlet

```
class ViewController: UIViewController {

    weak var label: UILabel!

}
```

Diese Property können Sie nun nutzen, um innerhalb Ihrer View-Controller-Klasse auf die Label-Instanz zuzugreifen und sie zu verändern. Doch bisher handelt es sich bei der Property um nichts weiter als eben das: eine einfache Property. Auch wenn sie vom Typ `UILabel` ist, ist sie bisher noch nicht mit dem Label gekoppelt, das wir für den View-Controller im Storyboard erstellt haben. Damit das so ist, müssen wir eine Verbindung zwischen der Property und dem Label im Storyboard herstellen.

Wie gehen wir hierfür vor? Zunächst müssen wir die Deklaration unserer `label`-Property noch minimal anpassen und ihr das Schlüsselwort `@IBOutlet` voranstellen (siehe Listing 23.9). Zunächst hat dieses Schlüsselwort keinerlei Auswirkungen auf die Funktionsweise der Property an sich. Es gibt lediglich an, dass die Property mit einem Element aus einer Interface-Datei (wie einem Storyboard) verbunden werden kann.

Listing 23.9 Deklaration einer Property als Outlet

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

}
```

Ist dieses Schlüsselwort gesetzt, fällt auch direkt eine Änderung im Editor-Fenster auf: Am linken Rand des Editors auf Höhe der Property erscheint ein Kreis (siehe Bild 23.48). Dieser weist darauf hin, dass die Property mit einer Interface-Datei verbunden werden kann.

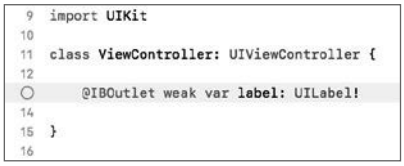


Bild 23.48
Am rechten Rand auf Höhe der Property-Deklaration erscheint ein Kreis, sobald wir das Schlüsselwort @IBOutlet hinzufügen.

Mit dieser Deklaration der ViewController-Klasse wechseln wir zurück in die *Main.storyboard*-Datei und wählen den initialen View-Controller aus, indem wir in die obere Leiste und dann auf das View-Controller-Symbol klicken (siehe hierzu auch Abschnitt 23.5.3, „Verbindung zwischen Interface und Code“). Anschließend rufen wir den sogenannten Connections Inspector in der Inspectors Area auf (siehe Bild 23.49). Dort findet sich im Abschnitt *Outlets* ein Eintrag namens *label*. Der ist dort nicht zufällig aufgetaucht, sondern es handelt sich um die `label`-Property, die wir im Code des zugehörigen View-Controllers mithilfe des Schlüsselworts `@IBOutlet` deklariert haben. Ohne dieses Schlüsselwort würde das Label nicht in dieser Liste auftauchen.

Bild 23.49
Aufgrund der Deklaration der label-Property als @IBOutlet taucht ein Verweis auf diese Property im Connections Inspector des View-Controllers auf.

Diesen *label*-Eintrag im Connections Inspector können Sie nun nutzen, um von dem Kreis am rechten Rand bei gedrückt gehaltener linker Maustaste eine Verbindung zu dem Label-Element im Interface des View-Controllers zu ziehen (siehe Bild 23.50). Sobald Sie mit der Maus über dem Label-Element sind, wird dieses blau hervorgehoben. Damit wird signalisiert, dass Sie das `label`-Outlet mit diesem Interface-Element koppeln können.

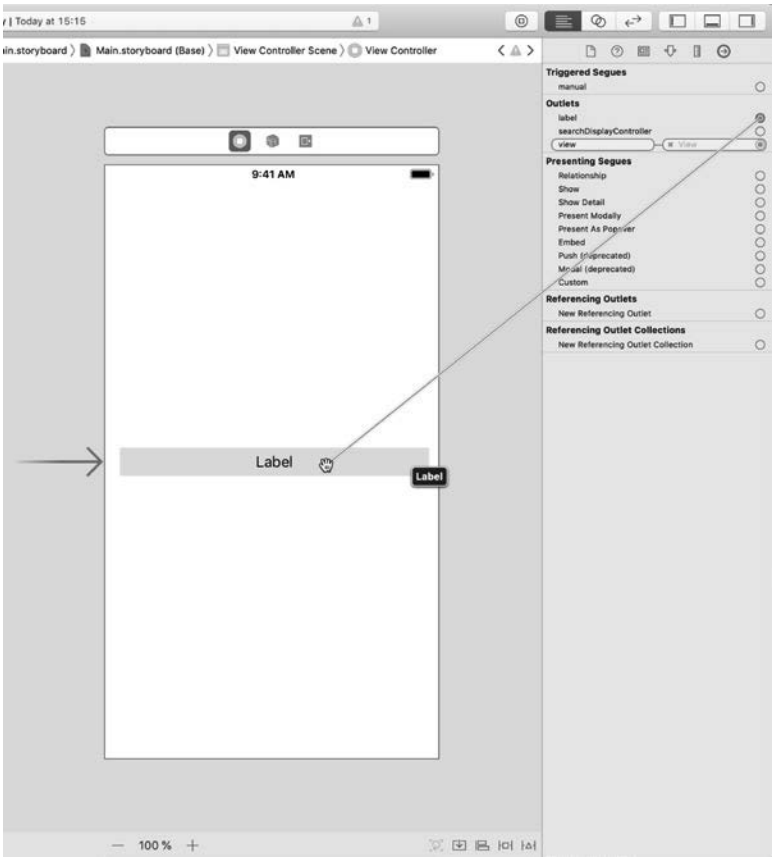


Bild 23.50 Ziehen Sie vom Connections Inspector eine Verbindung zum Label, um die label-Property mit dem Interface-Element zu verbinden.

Diese Kopplung ist nur möglich, weil Sie die `label`-Property als `@IBOutlet` deklariert haben und es sich bei der Property wie auch dem Interface-Element um ein und denselben Typ (nämlich `UILabel`) handelt.

Sobald Sie die linke Maustaste über dem Label-Element im Storyboard loslassen, stellt Xcode die Verbindung zwischen diesen beiden Elementen her. Das ist auch sehr schön im Connections Inspector zu sehen. Darin ist das *label*-Outlet nun hervorgehoben und der Kreis am rechten Rand ist gefüllt, was auf eine bestehende Verbindung von Outlet (sprich Code) und Interface hinweist (siehe Bild 23.51). Auch im Code der `ViewController`-Klasse ist der zugehörige Kreis der `label`-Property nun ausgefüllt, um auf eine bestehende Verbindung mit dem Interface hinzuweisen (siehe Bild 23.52).

Bild 23.51
Die bestehende Verbindung zwischen Code und Interface ist sowohl im Connections Inspector ...

```
 9  import UIKit
10
11  class ViewController: UIViewController {
12
13      @IBOutlet weak var label: UILabel!
14
15  }
16
```

Bild 23.52
... als auch in der Implementierung der ViewController-Klasse zu sehen.

Wenn Sie nun im Code der `ViewController`-Klasse auf die `label`-Property zugreifen, greifen Sie auf das im Storyboard hinzugefügte Label zu. Um das zu demonstrieren und, wie eingangs beschrieben, den Text des Labels nach Laden des View-Controllers in „Outlet“ zu ändern, überschreiben wir die `viewDidLoad()`-Methode in der Implementierung der ViewController-Klasse (mehr über die Methode `viewDidLoad()` und den Lebenszyklus eines View-Controllers erfahren Sie in Abschnitt 23.5.4, „Lebenszyklus eines View-Controllers“). Um den Text des Labels zu ändern, greifen wir auf die `text`-Property der Klasse `UILabel` zu. Die passende vollständige Implementierung der ViewController-Klasse finden Sie in Listing 23.10.

Listing 23.10 Ändern des Label-Textes nach Laden des View-Controllers

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        label.text = "Outlet"
    }

}
```

Wenn Sie das Projekt nun ausführen, stellen Sie fest, dass das Label nach Starten der App statt des vorgegebenen Textes aus dem Storyboard den String „Outlet“ anzeigt (siehe Bild 23.53).

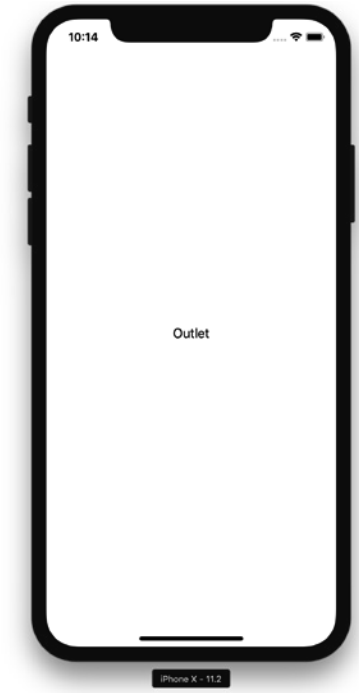


Bild 23.53
Nach dem Laden der Beispiel-App entspricht der Wert des Labels dem, der im Code der ViewController-Klasse gesetzt wurde.

Komfortablere Erstellung von Outlets

Wie eingangs erwähnt, handelt es sich bei dem gezeigten Verfahren um das „aufwendigere“ zur Erstellung von Outlets. Mithilfe von Xcode geht das Ganze auch deutlich einfacher und schneller vonstatten. Dazu gehen Sie wie folgt vor: Öffnen Sie zunächst die *Main.storyboard*-Datei und wählen Sie den View-Controller aus, für den ein Outlet erstellt werden soll. Öffnen Sie anschließend den Assistant Editor von Xcode per Klick auf die entsprechende Schaltfläche am oberen rechten Rand. Wählen Sie – falls nicht bereits standardmäßig der Fall – für die rechte Hälfte des Assistant Editor-Fensters den Code der `ViewController`-Klasse aus, sodass Interface und Code nebeneinander angezeigt werden (siehe Bild 23.54).

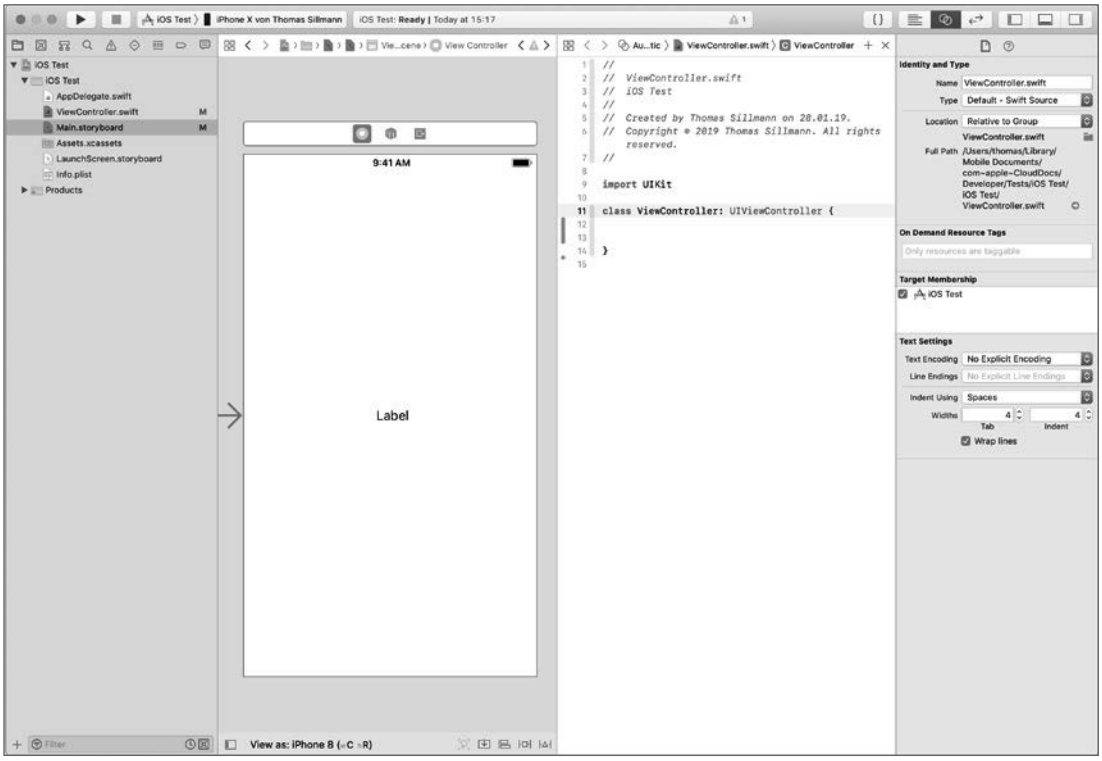


Bild 23.54 Zeigen Sie mithilfe des Assistant Editors Interface und Code nebeneinander an.

Klicken Sie nun im Storyboard auf die View, für die Sie im View-Controller-Code ein Outlet erzeugen möchten (in unserem Beispiel also für das Label), halten Sie die rechte Maustaste gedrückt und ziehen Sie eine Verbindung zur Implementierung der View-Controller-Klasse. Es erscheinen daraufhin blaue Hilfslinien, die Ihnen anzeigen, wo Xcode ein Outlet für die View im Code erstellen wird, sobald Sie die rechte Maustaste loslassen (siehe Bild 23.55).

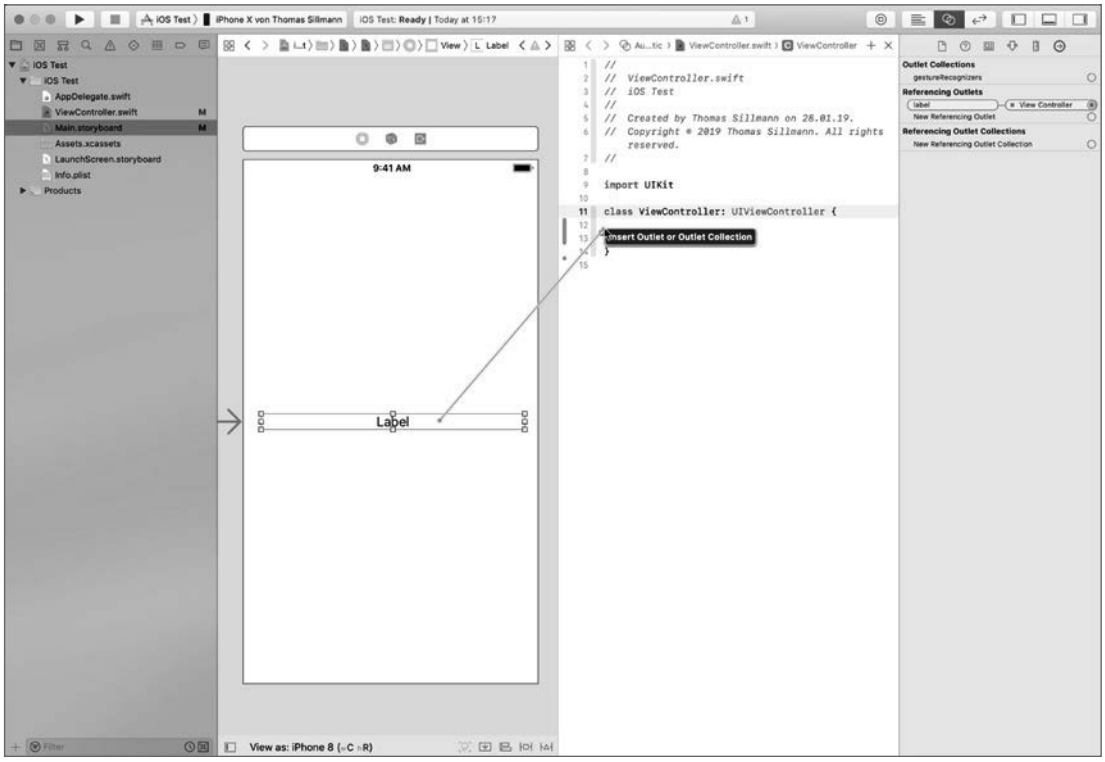


Bild 23.55 Ziehen Sie mit gedrückt gehaltener rechter Maustaste eine Verbindung vom Label in den Code, um so automatisch ein Outlet zu erstellen.

Wenn Sie die rechte Maustaste losgelassen haben, öffnet sich ein Pop-up-Menü, in dem Sie einige Informationen für das zu erstellende Outlet eingeben (siehe Bild 23.56). Dazu gehören:

- *Connection:* Die Art von Verbindung, die Sie von der View im Code erzeugen möchten. Für Outlets wählen Sie hier den gleichnamigen Punkt *Outlet* aus.
- *Object:* Hier wird Ihnen das Objekt angezeigt, für das die Verbindung hergestellt werden soll, in diesem Fall der *View Controller*.
- *Name:* Hier tragen Sie den gewünschten Namen für die Outlet-Property ein, beispielsweise *label*.
- *Type:* Hier bestimmen Sie den Typ der Outlet-Property. Dieses Feld ist mit dem Typ des gewählten View-Elements (in diesem Fall `UILabel`) vorbelegt und muss in der Regel nicht geändert werden.
- *Storage:* Wählen Sie hier die Regel zur Speicherverwaltung für das Outlet aus. Für Weak-References wählen Sie den Punkt *Weak*.

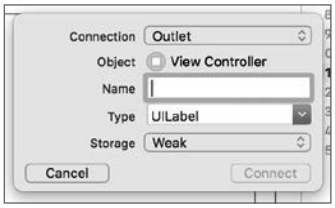


Bild 23.56
Mithilfe dieses Pop-up-Menüs deklarieren Sie das zu erstellende Outlet, das Xcode dem Code des View-Controllers hinzufügen soll.

Sind alle Informationen hinterlegt, klicken Sie auf die Schaltfläche *Connect*. Xcode erzeugt anschließend automatisch den Code für das gewünschte Outlet und fügt es der Klasse hinzu (siehe Bild 23.57).

```
 9  import UIKit
10
11  class ViewController: UIViewController {
12
13      @IBOutlet weak var label: UILabel!
14
15  }
16
```

Bild 23.57
Xcode hat das Outlet automatisch für uns erzeugt.

Neben dem Outlet hat Xcode auch gleich die Verbindung zwischen Code und Interface gesetzt. Mit diesem einfachen Schritt spart man sich somit die eigene Deklaration einer Outlet-Property sowie das gegenseitige Verbinden der Elemente aus Code und Interface.

Outlets für alle Arten von Views verfügbar

In diesem Beispiel wurde das Erstellen von Outlets auf Basis eines Labels demonstriert. Auf genau die gleiche Art und Weise können aber auch Outlets für alle anderen verfügbaren Arten von Views erzeugt werden, ganz gleich ob Buttons, Switches, Image-Views oder Tabellen. Wann immer Sie vom Code eines View-Controllers aus auf eine im Interface erstellte View zugreifen möchten, kreieren Sie ein Outlet dafür.

23.5.3.2 Actions

Mithilfe bestimmter View-Elemente löst ein Nutzer typischerweise Ereignisse aus. Ein Beispiel hierfür ist ein Button. Es handelt sich hierbei – genau wie bei einem Label – um eine View, allerdings zeigt ein Button nicht einfach nur etwas an. Nutzer können auf ihn tippen, woraufhin passende Befehle von der App ausgeführt werden sollen.

Ähnlich wie bei einem Outlet (siehe Abschnitt 23.5.3.1, „Outlets“) muss zu diesem Zweck eine Verbindung zwischen dem Button, der Teil des Interfaces ist, und dem Code der View-Controller-Klasse hergestellt werden. Allerdings basiert diese Verbindung nicht darauf, auf die Eigenschaften des Buttons zuzugreifen (Outlet), sondern darauf, das Tippen auf den Button durch den Nutzer abzufangen und darauf zu reagieren. Dieses Vorgehen wird als sogenannte *Action* bezeichnet.

Welche Views verfügen über eine Action?

Nicht jede in der iOS-Entwicklung zur Verfügung stehende View kann eine Action auslösen. Als Beispiel kann hierfür ein Label herhalten: Es zeigt etwas auf dem Bildschirm an, kann aber sonst keine weiteren Aktionen auslösen. Entsprechend kann für ein Label auch keine Action erzeugt werden.

Um zu überprüfen, ob eine View mit einer Action versehen werden kann, hilft ein Blick in die Dokumentation von Xcode. Klassen, die von `UIControl` abgeleitet sind, können auch mit einer Action gekoppelt werden (siehe Bild 23.58). Dazu gehören unter anderem die Klassen `UIButton` (für Schaltflächen), `UISwitch` (für Schalter) oder `UIStepper` (zum Verändern eines Werts).

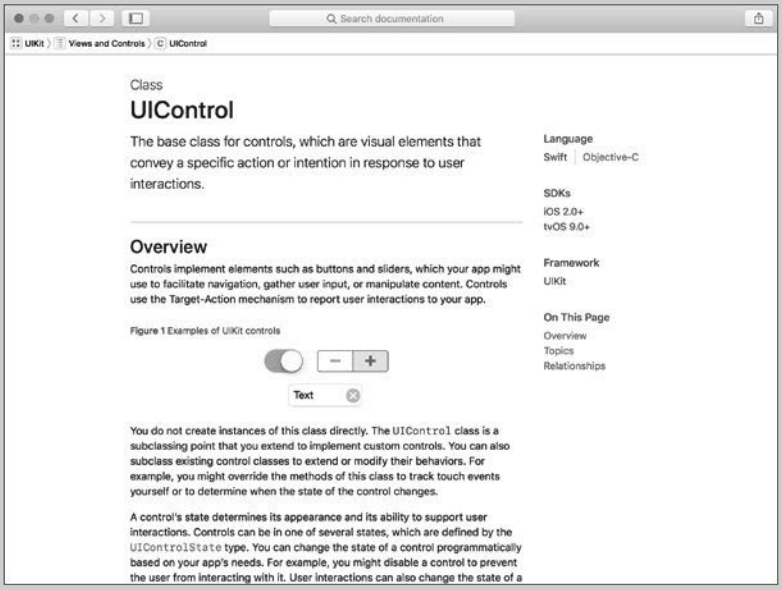


Bild 23.58 Die Klasse UIControl ist die Basisklasse für alle Views, die mit einer Action versehen werden können (wie zum Beispiel Buttons, Switches, Slider).

Mehr über `UIControl` und die verschiedenen zur Verfügung stehenden Views in der iOS-Entwicklung erfahren Sie in Abschnitt 23.6, „Oberflächen gestalten mit UIView“.

Eine Action entspricht einer Methode, die mit dem Schlüsselwort `@IBAction` deklariert ist. Eine solche Methode kann – analog zu Outlets – mit View-Elementen verbunden werden (sofern diese von `UIControl` abgeleitet sind und somit mit Actions umgehen können). Das folgende Beispiel soll die Verwendung von Actions beispielhaft demonstrieren. Erstellen Sie zu diesem Zweck zunächst ein neues iOS-Projekt auf Basis einer *Single View App* und fügen Sie dem initialen View-Controller an beliebiger Stelle ein Label und einen Button hinzu

(siehe Bild 23.59). Das Label verknüpfen Sie – so wie in Abschnitt 23.5.3.1, „Outlets“, gezeigt – als Outlet-Property mit dem Code des zugrunde liegenden View-Controllers.

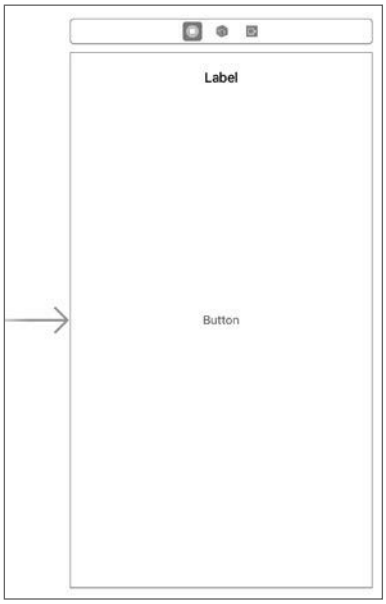


Bild 23.59
Die Beispiel-App verfügt über ein Label und einen Button. Das Label wird mithilfe eines Outlets mit dem Code verbunden.

Wann immer der Button der Beispiel-App betätigt wird, soll der Text des Labels zu „Button pressed!“ geändert werden. Um dieses Verhalten zu erreichen, müssen wir eine entsprechende Action für den Button umsetzen. Dazu gehen wir ganz ähnlich vor wie beim Erstellen eines Outlets: Zunächst öffnen wir die *Main.storyboard*-Datei und wechseln anschließend in den Assistant Editor. Im rechten Fensterbereich des Editors wird dann der Code der View-Controller-Klasse zur Anzeige ausgewählt (falls er nicht bereits automatisch angezeigt wird, siehe Bild 23.60).

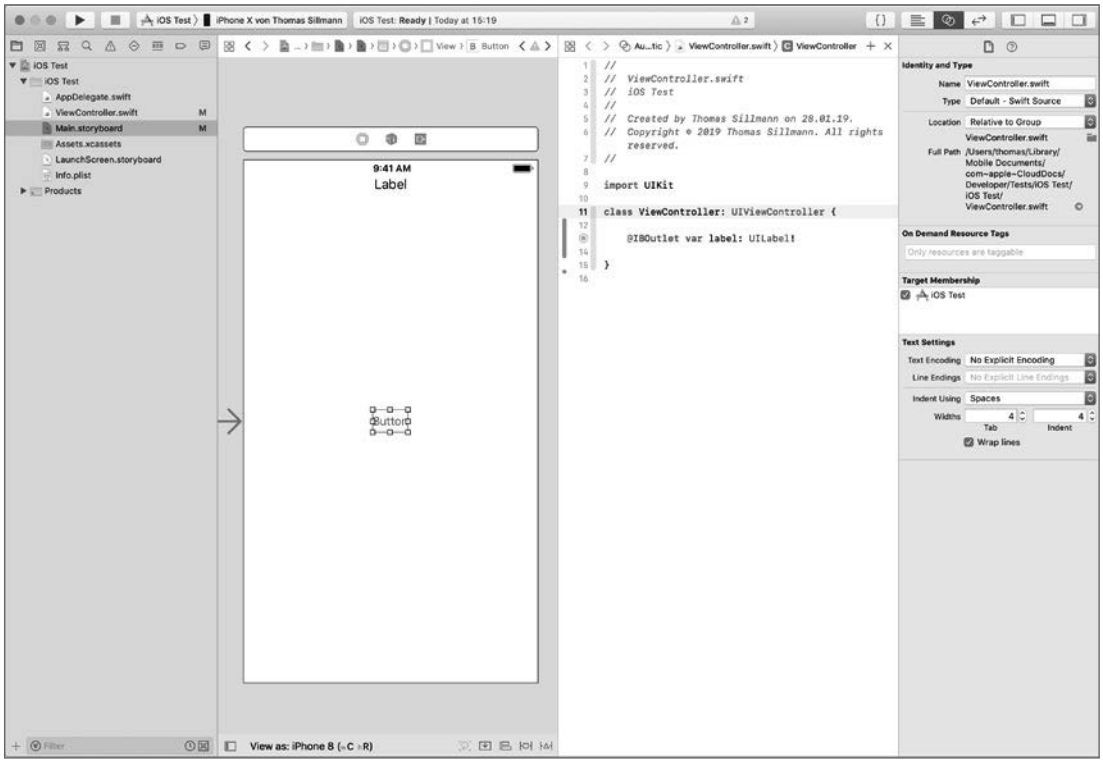


Bild 23.60 Zeigen Sie Interface und Code mithilfe des Assistant Editors nebeneinander an, um eine Action für den Button zu erstellen.

Klicken Sie dann auf den Button und ziehen Sie bei gedrückt gehaltener rechter Maustaste eine Verbindung in die Implementierung der View-Controller-Klasse (siehe Bild 23.61). Es erscheinen blaue Hilfslinien, die Ihnen zeigen, wo der Code für die Action eingefügt wird. Haben Sie sich für die passende Stelle entschieden, lassen Sie die rechte Maustaste wieder los, woraufhin das bereits von den Outlets bekannte Pop-up-Menü erscheint (siehe Bild 23.62).

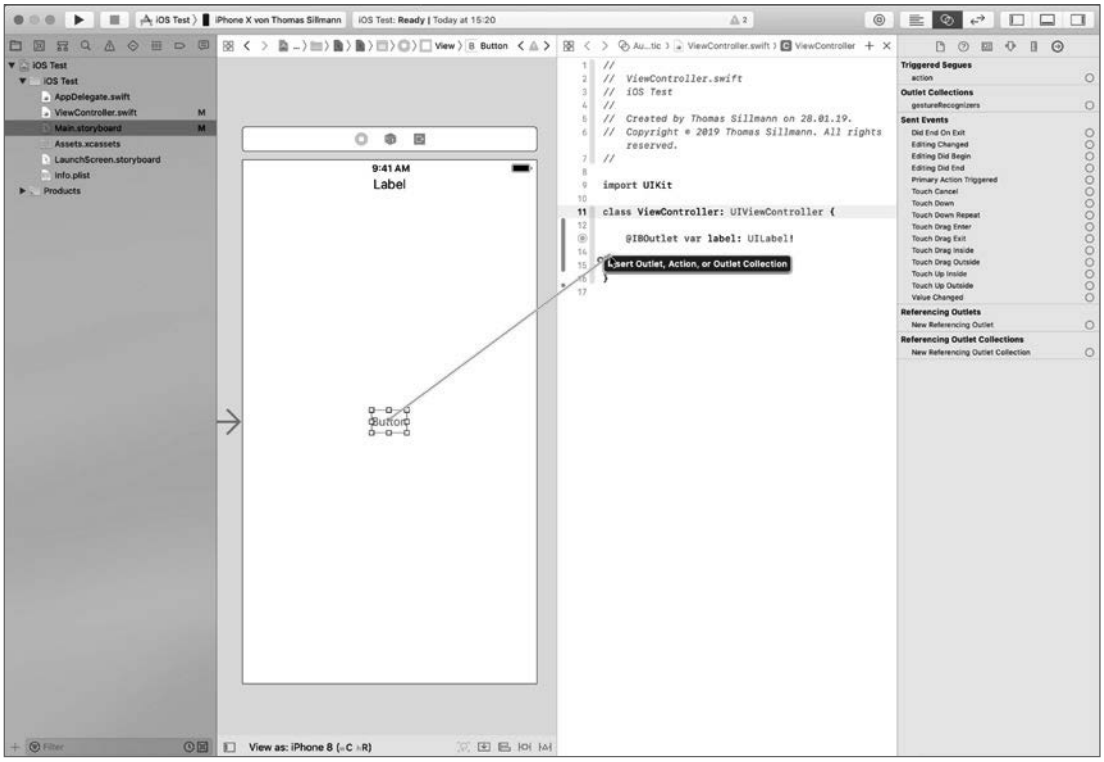


Bild 23.61 Ziehen Sie eine Verbindung vom Button in den Code, um eine Action zu erstellen.

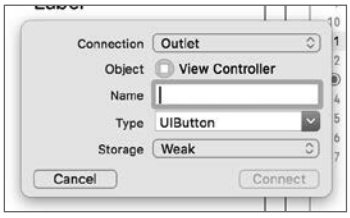


Bild 23.62
Es erscheint erneut das Pop-up-Menü, über das Sie die zu erstellende Verbindung des Buttons zwischen Interface und Code definieren.

Da für den Button kein Outlet, sondern eine Action erstellt werden soll, müssen Sie im ersten Schritt innerhalb des Pop-up-Menüs unter Connection den Punkt *Action* auswählen. Daraufhin verändert sich der Aufbau des Menüs ein wenig (siehe Bild 23.63). Die folgenden Einstellungen können Sie zur Konfiguration der zu erstellenden Action vornehmen:

- *Name:* Hier geben Sie den Namen für die zu erstellende Action-Methode ein, beispielsweise *buttonTapped*.
- *Type:* Hier definieren Sie den Typ, den die zu erstellende Action-Methode als Parameter erwartet. Standardmäßig handelt es sich bei dem Parameter um die View, die die Action auslöst – in diesem Fall also um die `UIButton`-Instanz, die wir dem Interface hinzugefügt haben. Den Parameter können Sie verwenden, um auf die `UIButton`-Instanz zuzugreifen und sie zu verändern oder weitere Informationen auszulesen. Der vorgegebene Standardwert an dieser Stelle ist `Any`, sprich bei dem Parameter kann es sich um jede beliebige

Instanz handeln. Das ist vor allen Dingen dann sinnvoll, wenn diese Action-Methode von mehreren verschiedenen Views ausgelöst wird (siehe hierzu auch den folgenden Abschnitt „Mehrere Views mit ein und derselben Action koppeln“). Wenn die zu erstellende Action hingegen ausschließlich von einer einzigen spezifischen View (wie in unserem Fall von dem Button) ausgelöst werden soll, dann können Sie für diese Option auch den passenden Typ (`UIButton`) direkt auswählen.

- *Event:* Hierüber wählen Sie das Ereignis aus, zu dem die zu erstellende Action-Methode ausgelöst werden soll. Die Klasse `UIControl` definiert für diesen Zweck verschiedene Ereignisse, beispielsweise ob der Button betätigt wurde *(Touch Up Inside),* ob ein Tippen auf den Button abgebrochen wurde *(Touch Cancel)* oder ob der Nutzer in der Fläche des Buttons mit dem Finger umherfährt *(Touch Drag Inside)*. Sie können also nicht nur auf das standardmäßige Betätigen des Buttons mit einer Action reagieren, sondern auch auf andere Ereignisse.

  Das Standard-Event zum Umgang mit Buttons ist *Touch Up Inside*. Es entspricht dem Tippen auf den Button zum Auslösen einer Aktion.

- *Arguments:* Über diese Option definieren Sie die Parameter, die für die Action-Methode von Xcode generiert werden sollen. Der Standard lautet *Sender*, was sich auf einen Parameter für den Auslöser der Action-Methode bezieht (in diesem Fall also standardmäßig die `UIButton`-Instanz aus dem Interface). Alternativ können Sie hier auch *Sender and Event* auswählen, wodurch die Methode einen zweiten Parameter vom Typ `UIEvent` erhält, der zusätzlich darüber informiert, welches Ereignis die Action-Methode ausgelöst hat (siehe den vorherigen Punkt *Event*).

  Falls für Sie die Parameter bei Durchführung der Action keine Rolle spielen, können Sie auch *None* auswählen (was ich für dieses Beispiel auch tue).

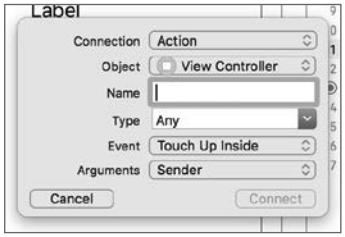


Bild 23.63
Zur Konfiguration einer zu erstellenden Action-Methode müssen Sie diverse Informationen angeben.

Haben Sie die Konfiguration der Action auf die gewünschte Art und Weise abgeschlossen, klicken Sie auf die Schaltfläche *Connect*. Daraufhin erzeugt Xcode eine Methode im Code des View-Controllers, die den von Ihnen definierten Namen trägt und die von Ihnen ausgewählten Parameter besitzt (siehe Bild 23.64).

```
9  import UIKit
10
11 class ViewController: UIViewController {
12
13     @IBOutlet weak var label: UILabel!
14
15     @IBAction func buttonTapped() {
16     }
17
18 }
19
```

Bild 23.64
Zusätzlich zum Outlet haben wir über Xcode eine Action-Methode erstellt.

Diese Methode ist genauso deklariert wie jede andere Methode in Swift auch, besitzt aber zusätzlich das Schlüsselwort `@IBAction`. Nur Methoden, die mit diesem Schlüsselwort versehen sind, können mit Views im Interface gekoppelt werden. Der ausgefüllte runde Punkt am linken Rand der Methodendeklaration zeigt darüber hinaus, dass die erstellte Action-Methode bereits mit dem Interface gekoppelt ist (nämlich mit der `UIButton`-Instanz). Diese Verbindung kann auch über den Connections Inspector im Storyboard nachvollzogen werden, nachdem man entweder den View-Controller oder den Button ausgewählt hat. Dort wird dann ebenfalls die Kopplung zwischen Interface und Code sichtbar (siehe Bild 23.65).

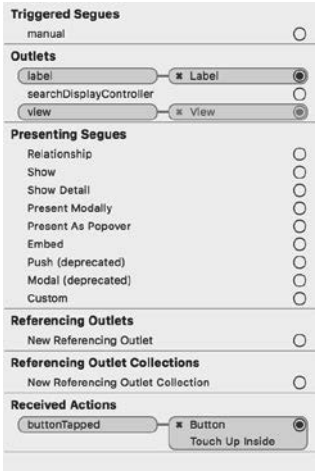


Bild 23.65
Der Connections Inspector des View-Controllers zeigt, dass das Label in Form eines Outlets und der Button in Form einer Action-Methode mit dem Code verbunden sind.

Um das Beispiel nun zu vollenden muss noch die neu erstellte Action-Methode `buttonTapped()` im Code der View-Controller-Klasse passend implementiert werden. Mithilfe der Property `text` wird darin der Text des Labels auf „Button pressed!“ geändert. Die vollständige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.11.

Listing 23.11 Implementierung der Action-Methode des Buttons

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func buttonTapped() {
        label.text = "Button pressed!"
    }

}
```

Wenn Sie nun dieses Beispielprojekt ausführen und den Button betätigen, ändert sich der Text des Labels von dem Standardwert, der im Storyboard definiert ist, zu „Button pressed!“ (siehe Bild 23.66).

Bild 23.66 Nach Betätigen des Buttons ändert sich der Text des Labels vom Standardwert aus dem Storyboard zu „Button pressed!“.

Mehrere Views mit ein und derselben Action koppeln

Es ist möglich, dass mehrere Views ein und dieselbe Action-Methode auslösen. Das kann in Fällen sinnvoll sein, in denen sich die Action verschiedener Views nur minimal voneinander unterscheidet. Statt mehrerer Action-Methoden, die im Grunde dann sehr ähnlich sind, kann man auch nur eine Action-Methode implementieren und abhängig von der Instanz, die die Action ausgelöst hat, die auszuführende Logik anpassen.

Das folgende Beispiel zeigt, wie sich mehreren Views ein und dieselbe Action-Methode zuweisen lässt. Basis ist ein neues iOS-Projekt auf Basis einer *Single View App*, dessen initialer View-Controller über ein Label und drei Buttons verfügt (siehe Bild 23.67). Die Buttons werden mit den Titeln *Button 1*, *Button 2* und *Button 3* versehen. Für das Label wird ein Outlet erstellt und für den ersten der drei Buttons eine Action-Methode mit Namen `buttonTapped(_:)`. Wichtig bei der Erstellung der Action-Methode ist, dass diese den Sender-Parameter erhält, sprich die aufrufende View bei Auslösen der Methode mit übergibt. Als Typ für diesen Parameter wird statt `Any` wieder `UIButton` ausgewählt. Diesen abgeschlossenen ersten Zwischenstand zeigt Bild 23.68.

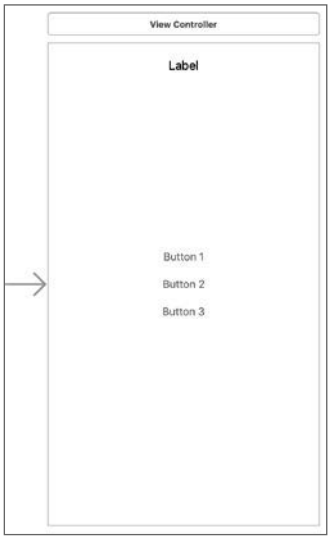


Bild 23.67
Die Beispiel-App verfügt über ein Label und drei Buttons, die mit den Titeln „Button 1“, „Button 2“ und „Button 3“ versehen sind.

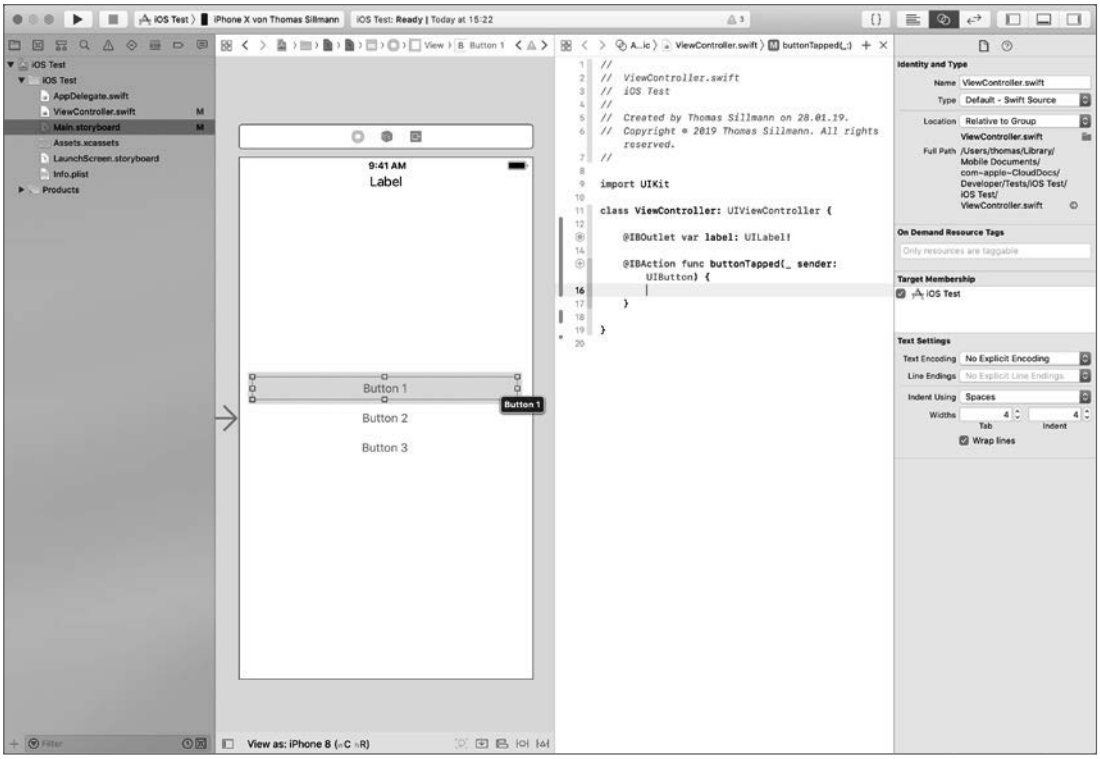


Bild 23.68 Für das Label wird ein Outlet im Code erzeugt und der erste Button wird mit einer Action-Methode namens „buttonTapped(_:)“ verknüpft, die über einen Sender-Parameter vom Typ „UIButton“ verfügt.

Die Action-Methode `buttonTapped(_:)` soll den Titel des Buttons, der die Methode ausgelöst hat, auslesen und dem Label als Text zuweisen. Damit kann das Label – abhängig vom betätigten Button – den Text *Button 1*, *Button 2* oder *Button 3* annehmen.

Den Titel eines Buttons kann man über die Property `titleLabel.text` auslesen. Zugriff auf den Button, der die Action-Methode auslöste, erhält man über den `sender`-Parameter. Listing 23.12 zeigt, wie die `ViewController`-Klasse vollständig implementiert werden muss, um die gewünschte Logik umzusetzen.

Listing 23.12 Ändern des Labels auf Basis des betätigten Buttons

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func buttonTapped(_ sender: UIButton) {
        let buttonTitle = sender.titleLabel?.text
        label.text = buttonTitle
    }

}
```

Mit diesem Zwischenstand lässt sich das Projekt bereits ausführen und bei Betätigen von *Button 1* wird auch tatsächlich der Text des Labels entsprechend geändert. Allerdings sind *Button 2* und *Button 3* noch mit keiner Action-Methode verknüpft und tun entsprechend bei Betätigung nichts. Da die Funktion, die diese Buttons ausführen sollen, identisch ist mit der bereits vorhandenen Action-Methode, macht es Sinn, ihnen dieselbe Action-Methode zuzuweisen.

Um das zu tun, ruft man zunächst die *Main.storyboard*-Datei auf und wechselt in den Assistant Editor, um parallel den Code der View-Controller-Klasse anzuzeigen. Anschließend zieht man von dem bereits gefüllten Punkt links von der Action-Methode mit gedrückt gehaltener linker Maustaste eine Verbindung zu *Button 2* und lässt anschließend los (siehe Bild 23.69). Damit ist auch der zweite Button mit der Action-Methode verknüpft. Analog geht man dann noch einmal für *Button 3* vor.

Damit rufen nun alle drei Buttons ein und dieselbe Action-Methode auf. Da in deren Implementierung der Titel des Buttons dynamisch ausgelesen und dem Label zugewiesen wird, hat man mit wenig Code eine effiziente Lösung entwickelt.

Wählt man nun im Storyboard den View-Controller aus und wechselt in den Connections Inspector, kann man darüber ebenfalls erkennen, dass die Action-Methode `buttonTapped(_:)` mit allen drei Buttons verknüpft ist (siehe Bild 23.70).

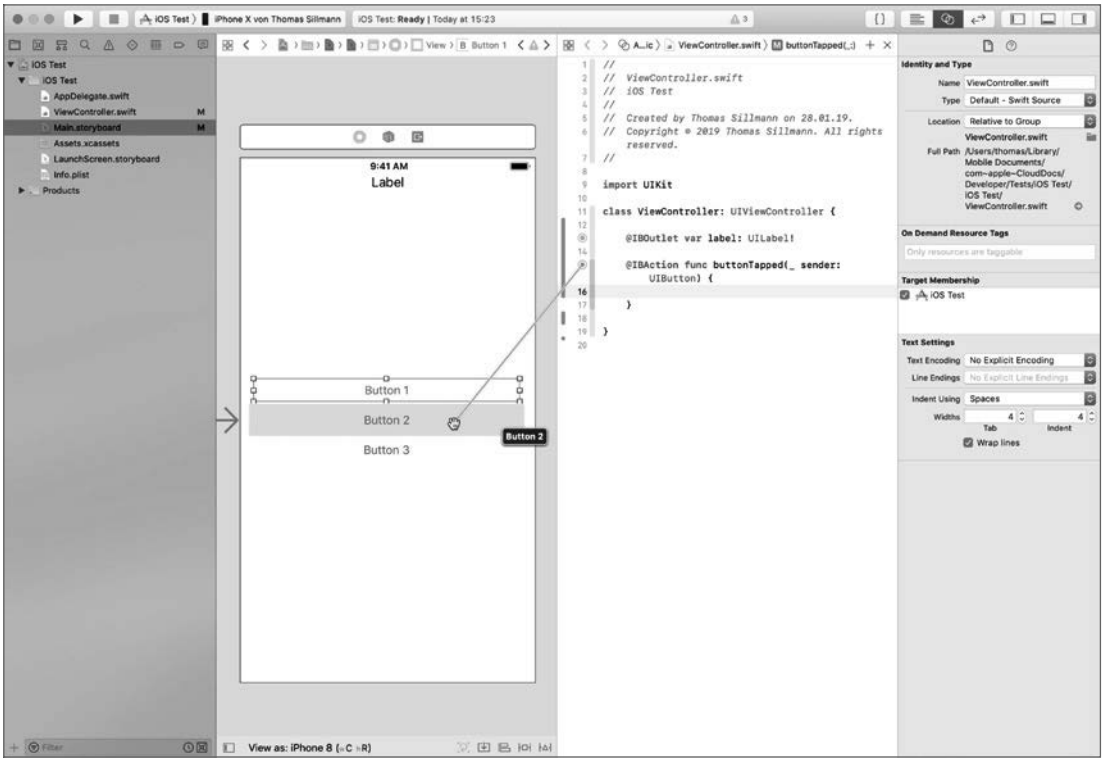


Bild 23.69 Ziehen Sie von der bereits erstellten und verknüpften Action-Methode im Code nacheinander eine Verbindung zu den beiden weiteren Buttons, die ebenfalls diese Methode aufrufen sollen.

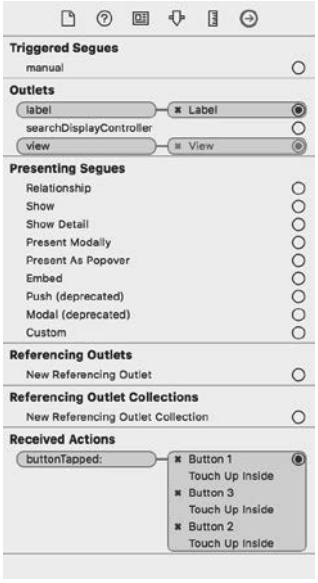


Bild 23.70
Im unteren Bereich „Received Actions“ ist zu sehen, dass alle drei Buttons mit ein und derselben Action-Methode verknüpft sind.

Wenn Sie das Projekt nun ausführen wird der Text des Labels passend zu dem Titel des Buttons geändert, den Sie auswählen.

23.5.4 Lebenszyklus eines View-Controllers

Ein View-Controller kennt vier verschiedene Zustände, die seinen sogenannten *Lebenszyklus* bestimmen:

- *Appearing:* Ein View-Controller wird gerade eingeblendet, ist aber noch nicht auf dem Display zu sehen.
- *Appeard:* Ein View-Controller wurde eingeblendet und wird angezeigt.
- *Disappearing:* Ein View-Controller wird ausgeblendet, ist aber noch auf dem Display zu sehen.
- *Disappeared:* Ein View-Controller wurde ausgeblendet und wird nicht länger angezeigt.

Für jede dieser vier Phasen bringt die Klasse `UIViewController` eine passende Methode mit, die in Subklassen überschrieben werden kann, um auf das jeweilige Event zu reagieren. Möchte man beispielsweise bei Erscheinen eines View-Controllers automatisch einen Timer starten, kann man die passende Methode für den *Appearing*-Status überschreiben und mit genau dieser Logik füllen.

Die Methoden zum Abfangen der genannten Zustände lauten wie folgt:

- *Appearing:* `viewWillAppear(_:)`
- *Appeared:* `viewDidAppear(_:)`
- *Disappearing:* `viewWillDisappear(_:)`
- *Disappered:* `viewDidDisappear(_:)`

Sie werden nacheinander aufgerufen und können – wie bereits erwähnt – überschrieben werden, um in jeder der Phasen eigenen Code für einen View-Controller auszuführen. Listing 23.13 zeigt die beispielhafte Implementierung einer View-Controller-Klasse, die über ein Label-Outlet verfügt und alle vier genannten Methoden überschreibt. Der Text des Labels wird hierbei jedes Mal entsprechend angepasst.

Listing 23.13 Implementierung der Methoden des Lebenszyklus eines View-Controllers

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        label.text = "View will appear"
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        label.text = "View did appear"
    }

    override func viewWillDisappear(_ animated: Bool) {
```

```
        super.viewWillDisappear(animated)
        label.text = "View will disappear"
    }

    override func viewDidDisappear(_ animated: Bool) {
        super.viewDidDisappear(animated)
        label.text = "View did disappear"
    }

}
```

Da die genannten Methoden allesamt von der Klasse `UIViewController` abgeleitet sind, müssen Sie in eigenen Subklassen mithilfe des `override`-Keywords überschrieben werden. Ebenso sollten Sie innerhalb Ihrer eigenen Implementierung die zugehörige Methode mittels `super` aufrufen, damit auch alle Superklassen ihre jeweils eigene Logik für diese Methoden aufrufen und ausführen können.

Methode viewDidLoad()

Die in vorherigen Beispielen verwendete Methode `viewDidLoad()` stellt einen Sonderfall dar und gehört nicht direkt zum beschriebenen Lebenszyklus eines View-Controllers. Sie wird einmalig aufgerufen, wenn ein View-Controller erstellt und dessen View zum ersten Mal geladen wird. Damit wird sie bereits vor `viewWillAppear(_:)` ausgelöst.

Solange ein View-Controller sich aber nun im Speicher befindet, wird `viewDidLoad()` nicht erneut aufgerufen; die View ist schließlich vollständig geladen und braucht nicht erneut erstellt zu werden.

Im Gegensatz dazu stehen die vorgestellten Methoden des Lebenszyklus eines View-Controllers. Diese werden immer und immer wieder aufgerufen, wenn das zugrunde liegende Ereignis eintritt.

Bei der Frage, welche der verfügbaren Methoden Sie für Ihre eigene Implementierungen nutzen, müssen Sie sich daher die Frage stellen, zu welchen Ereignissen Sie Ihre Befehle ausführen müssen. `viewDidLoad()` wird typischerweise für ergänzende Arbeiten zum vollständigen Erstellen eines View-Controllers verwendet (beispielsweise um allen Labels einen passenden Text zuzuweisen). Die Methoden des Lebenszyklus hingegen sollten sie für immer wiederkehrende Aufgaben einsetzen, die im Zusammenhang mit dem Erscheinen und Ausblenden eines View-Controllers stehen. Wenn Sie beispielsweise in einem View-Controller einen Timer laufen lassen, der nur dann aktiv sein soll, während der View-Controller angezeigt wird, können Sie `viewDidDisappear(_:)` zum Deaktivieren und `viewDidAppear(_:)` zum Aktivieren des Timers nutzen.

23.5.5 Neuen View-Controller einblenden

Eine der in fast allen iOS-Projekten anzutreffenden Aufgaben besteht darin, im Laufe einer App weitere View-Controller einzublenden. Hierfür gibt es verschiedene Vorgehensweisen, die stark von dem grundlegenden Aufbau und der Navigationsstruktur einer App abhängen.

In diesem Abschnitt stelle ich Ihnen die einfachste und grundlegendste Möglichkeit vor, um unter iOS einen neuen View-Controller einzublenden. Dabei wird ein solcher neuer View-Controller über einen anderen gelegt, wodurch dieser andere nicht länger sichtbar ist. In Kapitel 24, „iOS – App-Entwicklung“, stelle ich Ihnen dann weitere Möglichkeiten zur Strukturierung und zum Ein- und Ausblenden von View-Controllern vor.

Um in iOS einen neuen View-Controller anzuzeigen, gibt es generell zwei verschiedene Wege: Entweder nutzen Sie ein Storyboard oder Sie führen den Vorgang mithilfe passender Befehle der `UIViewController`-Klasse im Code aus. Die folgenden Abschnitte beschreiben beide Vorgehensweisen im Detail.

23.5.5.1 Über das Storyboard

Storyboards sind ein beliebtes und bequemes Mittel, um einen neuen View-Controller in iOS-Apps einzublenden. Generell muss man dazu wenigstens die folgenden Schritte durchführen:

1. Erstellen des neuen View-Controllers im Storyboard.
2. Laden des Storyboards über eine Action-View (beispielsweise einen Button) mithilfe eines Segues.

Um das Vorgehen einmal praktisch zu demonstrieren, stelle ich Ihnen im Folgenden ein kleines passendes Beispiel vor. Erstellen Sie ein neues iOS-Projekt auf Basis einer *Single View App* und rufen Sie anschließend die *Main.storyboard*-Datei auf. Wechseln Sie in die Objects Library und suchen Sie nach dem *View Controller*-Element (es wird über ein gelbes, kreisrundes Symbol mit einem weißen Viereck in der Mitte dargestellt, siehe Bild 23.71). Klicken Sie mit der linken Maustaste darauf und ziehen Sie so eine Instanz dieses Elements auf das Storyboard. Beachten Sie hierbei, dass Sie dieses Element nicht wie Views auf dem bestehenden initialen View-Controller platzieren, sondern irgendwo auf der freien Fläche des Storyboards. Der Grund hierfür ist einfach: Sie wollen ja nicht den bestehenden View-Controller anpassen, sondern einen ganz neuen View-Controller erstellen.

Bild 23.71
View-Controller werden in der Objects Library über dieses Symbol abgebildet.

Haben Sie das erledigt, werden in Ihrem Storyboard zwei View-Controller angezeigt (siehe Bild 23.72). Derjenige mit dem Pfeil an der linken Seite ist der automatisch von Xcode erzeugte initiale View-Controller, der beim Starten der App geladen und angezeigt wird. Der andere ist Ihr neu hinzugefügter View-Controller.

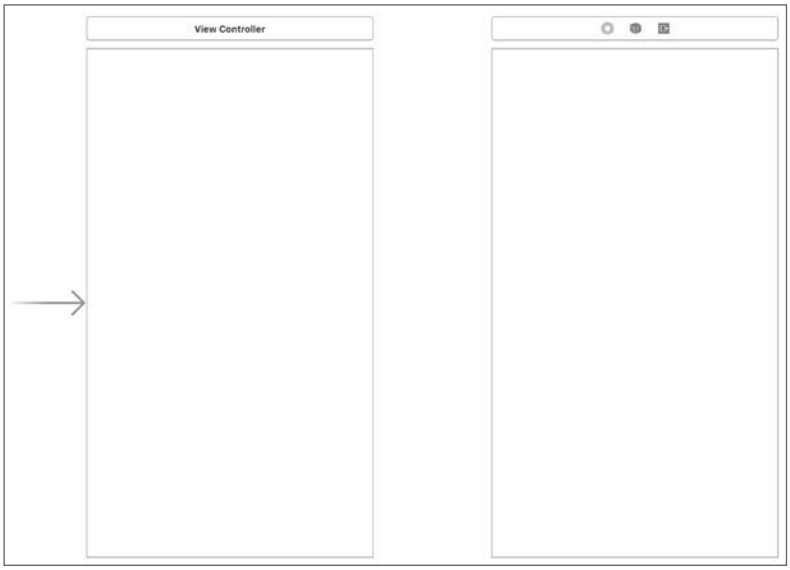


Bild 23.72 Das Storyboard verfügt nun über zwei verschiedene View-Controller (auch wenn diese noch nicht recht unterschiedlich aussehen).

Die Beispiel-App soll nun einen Button im initialen View-Controller anbieten, über den der zweite View-Controller eingeblendet werden kann. Dazu wird zunächst dem initialen View-Controller an einer beliebigen Stelle ein solcher Button mit dem Titel „Show second view controller“ hinzugefügt. Zur besseren Unterscheidung der beiden View-Controller erhält der zweite ein beliebig platziertes Label mit dem Text „Second view controller“. Das so grundlegend konfigurierte Storyboard zeigt Bild 23.73.

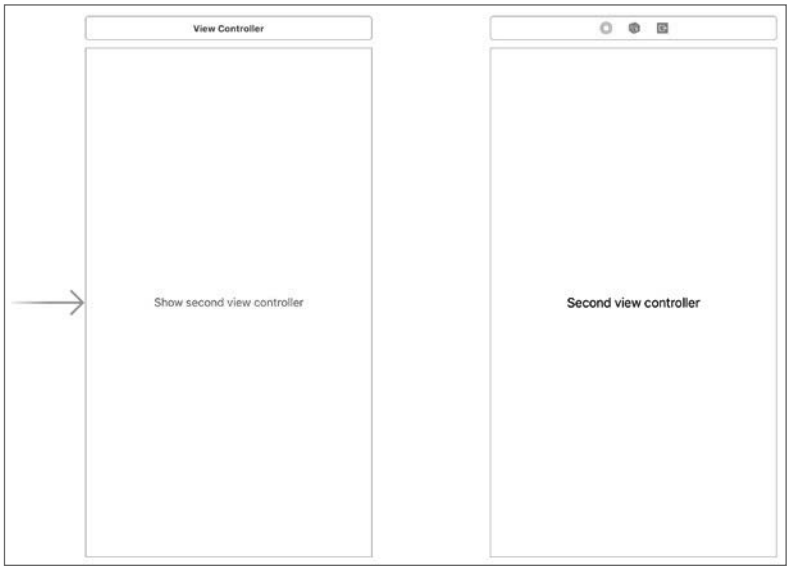


Bild 23.73 Der erste View-Controller erhält einen Button, der zweite ein Label.

Bleibt nun noch die Verbindung der beiden, um durch Betätigen des Buttons den zweiten View-Controller aufzurufen. Dazu wählen Sie den Button im initialen View-Controller aus, halten die rechte Maustaste gedrückt und ziehen eine Verbindung zu dem zweiten View-Controller, bis dieser blau hervorgehoben wird (siehe Bild 23.74). Lassen Sie anschließend die rechte Maustaste wieder los. Es erscheint daraufhin ein Pop-up-Menü, über das Sie bestimmen können, wie der Button mit dem zweiten View-Controller verbunden werden soll (siehe Bild 23.75). Um den zweiten View-Controller nach Tippen auf den Button einzublenden, wählen Sie im Abschnitt *Action Segue* den Punkt *Show* aus. Daraufhin verschwindet das Pop-up-Menü und es wird im Storyboard eine Verbindung in Form eines Pfeils zwischen den beiden View-Controllern angezeigt (siehe Bild 23.76).

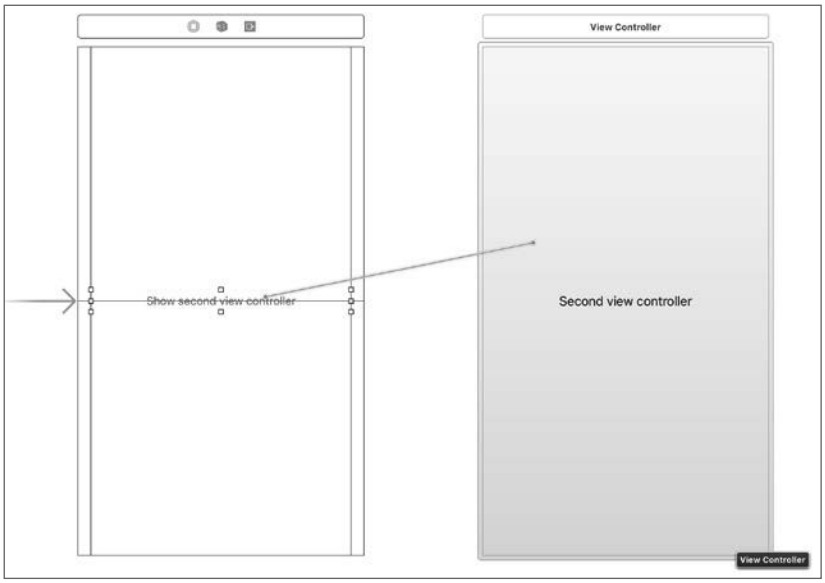


Bild 23.74 Ziehen Sie zunächst eine Verbindung von dem Button zum zweiten View-Controller.

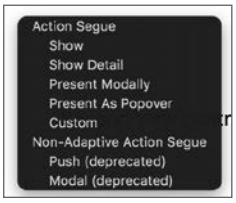


Bild 23.75
In dem erscheinenden Pop-up-Menü wählen Sie „Show“, um bei Tippen auf den Button den zweiten View-Controller einzublenden.

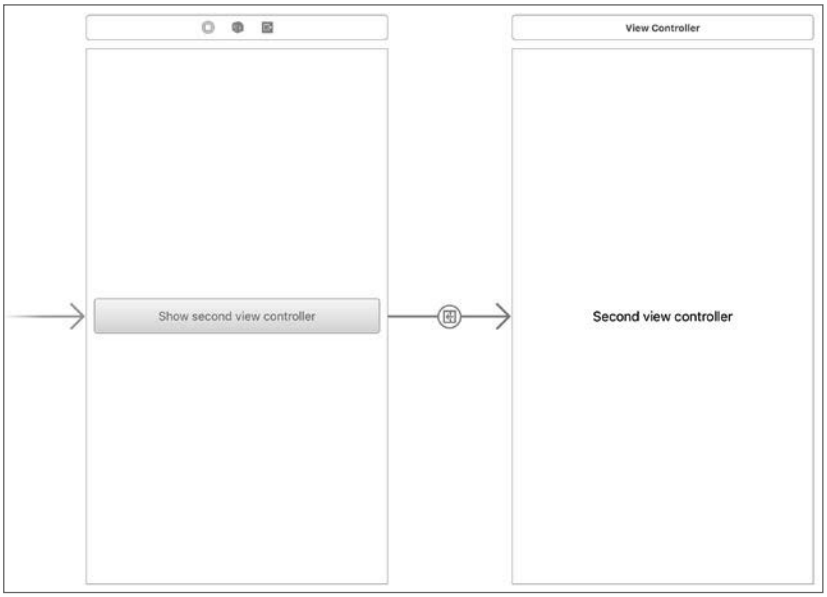


Bild 23.76 Die hergestellte Verbindung der beiden View-Controller wird im Storyboard auf Basis eines Pfeil-Symbols grafisch hervorgehoben.

Damit ist es geschafft! Ohne eine einzige Zeile Code zu schreiben blendet die App bei Tippen auf den Button den zweiten View-Controller ein. Sie können das selbst testen, indem Sie das Projekt kompilieren und ausführen (siehe Bild 23.77).

Bild 23.77 Nach Tippen auf den Button wird der zweite View-Controller eingeblendet – und das, ohne eine einzige Zeile Code schreiben zu müssen!

Das Verwenden von Storyboards zum Einblenden eines neuen View-Controllers hat zwei große Vorteile:

- Es ist einfach: Es braucht keine einzige Zeile Code, um den neu anzuzeigenden View-Controller zu erstellen und einzublenden. Um all diese Prozesse kümmert sich das Storyboard.
- Es ist übersichtlich: Ein Blick in das Storyboard verrät, wie die verschiedenen View-Controller einer App miteinander verknüpft sind. Das ist meist deutlich einfacher nachzuvollziehen und übersichtlicher, als im Code die entsprechenden Aufrufbefehle nachzuvollziehen.

Verknüpfung beliebig vieler View-Controller

Über ein Storyboard können Sie eine beliebige Anzahl von View-Controllern erstellen und miteinander verknüpfen. Beispielsweise könnte der zweite View-Controller aus dem gezeigten Beispiel wiederum einen dritten View-Controller laden und anzeigen und so weiter. ■

Segue

Die im gezeigten Beispiel hergestellte Verbindung zwischen Button und zweitem View-Controller wird als sogenannter *Segue* bezeichnet. Storyboards bieten verschiedene Arten von Segues, die aber letztlich immer der Verbindung von View-Controllern dienen.

In Kapitel 24, „iOS – App-Entwicklung“, werden Sie noch weitere Formen von Segues kennenlernen. ■

Informationen an neuen View-Controller übergeben

Beim Laden eines neuen View-Controllers möchte man in vielen Fällen zusätzliche Informationen übergeben. Nehmen wir als Beispiel eine To-do-App, die in einem View-Controller eine Liste von Aufgaben aufführt und nach Auswahl einer Aufgabe einen neuen View-Controller lädt, der Details zu eben dieser Aufgabe anzeigt. Damit der zweite View-Controller die richtigen Details anzeigen kann, muss er wissen, für welche Aufgabe er sie einblenden soll.

Zu diesem Zweck stellt die Klasse `UIViewController` eine Methode namens `prepare(for:sender:)` zur Verfügung. Sie wird aufgerufen, sobald ein View-Controller einen Segue auslöst – in dem gezeigten Beispiel also bei Tippen auf den Button. Somit lässt sich diese Methode nutzen, um auf den ausgelösten Segue zu reagieren und zusätzliche Befehle auszuführen – beispielsweise das Übergeben von Informationen an den anzuzeigenden Ziel-View-Controller.

Betrachten wir die Verwendung der Methode `prepare(for:sender:)` einmal in der Praxis. Dazu erstellen wir zunächst ein neues Projekt auf Basis einer *Single View App*, das genauso aufgebaut ist wie das aus dem vorherigen Abschnitt. Wir fügen also im ersten Schritt einen weiteren View-Controller in der *Main.storyboard*-Datei hinzu, versehen diesen mit einem Label und ergänzen einen Button im initialen View-Controller, von dem aus wir einen *Show*-Segue zum zweiten View-Controller erstellen.

Nun ergänzen wir aber zusätzlich ein Textfeld im initialen View-Controller. Es handelt sich hierbei um ein View-Element vom Typ `UITextField`. Bild 23.78 zeigt, wie das zugehörige Element in der Objects Library aussieht. Platzieren Sie ein solches Textfeld an einer beliebigen Stelle im initialen View-Controller, sodass das Interface Ihrer App in etwa so aussieht wie in Bild 23.79 gezeigt.

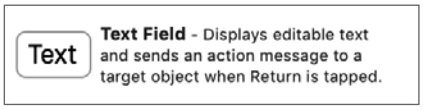


Bild 23.78
Ein Textfeld ist ein View-Element, mit dem Nutzer Text in einer App eingeben können.

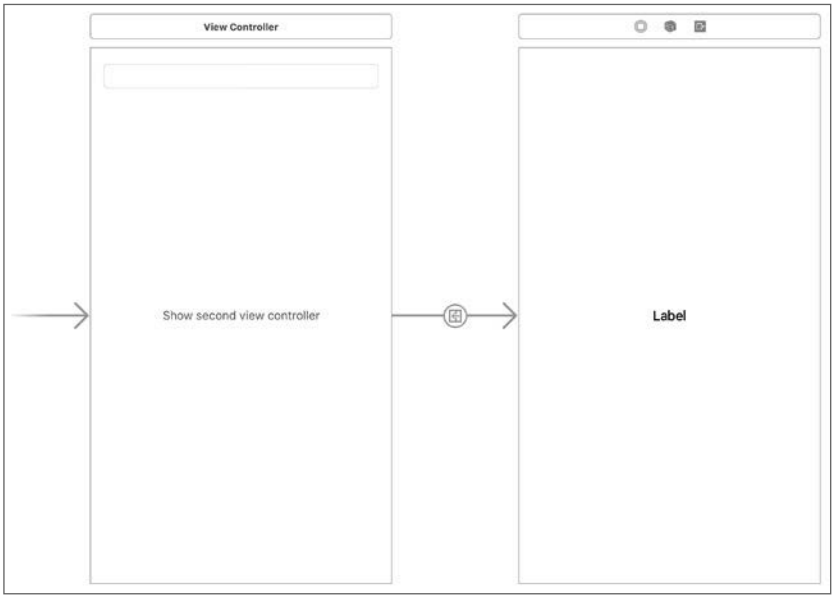


Bild 23.79 Das Interface der Beispiel-App besteht aus einem initialen View-Controller mit einem Textfeld und einem Button sowie einem zweiten View-Controller mit einem Label. Der Button ist mittels Show-Segue mit dem zweiten View-Controller verbunden.

Ziel dieser Beispiel-App soll es sein, den Text, der im Textfeld des initialen View-Controllers steht, an den zweiten View-Controller zu übergeben und dort im Label anzuzeigen. Wir übergeben also eine Information in Form eines Texts von einem View-Controller an den nächsten. Auf die gleiche Art und Weise lassen sich so beliebige weitere Informationen übertragen.

Bevor wir mit der Implementierung der Methode `prepare(for:sender:)` beginnen, müssen wir zunächst eine `UIViewController`-Subklasse für den zweiten View-Controller erstellen. Das ist notwendig, weil wir nur über eine solche Klasse die passende Logik implementieren können, um den Text des Labels dynamisch zu verändern.

Fügen wir also dem Projekt eine neue Klasse auf Basis von `UIViewController` hinzu und nennen sie `LabelViewController` (siehe Bild 23.80). Anschließend wechseln wir zurück in die *Main.storyboard*-Datei und wählen den zweiten View-Controller aus. Im Identity Inspec-

tor tragen wir sodann im Feld *Class* den Namen der eben erstellten `LabelViewController`-Klasse ein, um Interface und Code dieses View-Controllers miteinander zu koppeln (siehe Bild 23.81).

Bild 23.80
Erstellen Sie eine Klasse für den zweiten View-Controller, um dynamisch den Text des Labels verändern zu können.

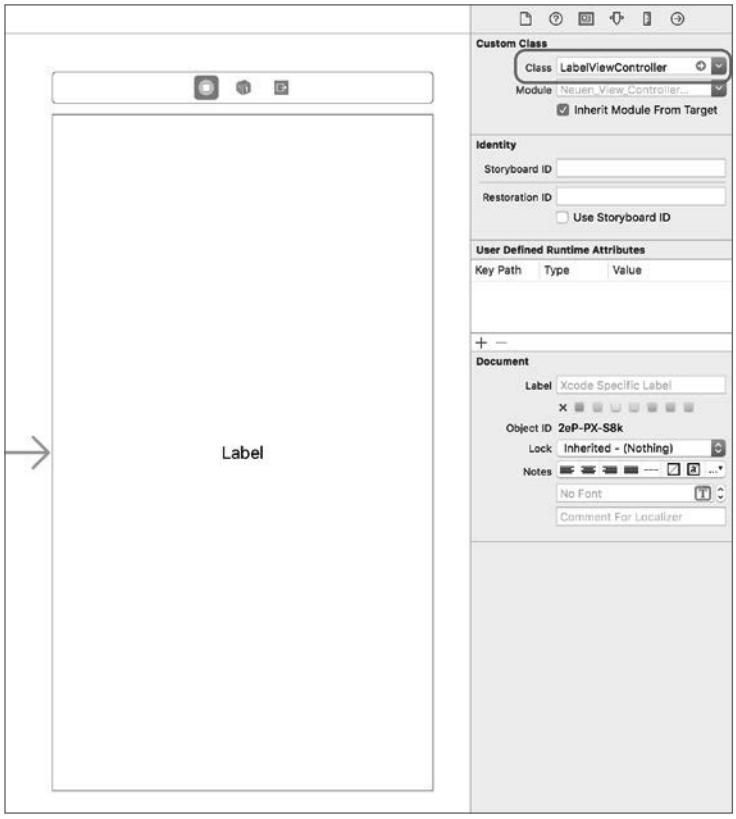


Bild 23.81 Weisen Sie dem zweiten View-Controller im Storyboard die neu erstellte UIViewController-Subklasse zu, um Interface und Code miteinander zu koppeln.

Um die Konfiguration der neuen `LabelViewController`-Klasse abzuschließen, müssen wir noch ein Outlet für das Label im Code erstellen. Wählen Sie dazu den zweiten View-Controller im Storyboard aus und öffnen Sie den Assistant Editor von Xcode. Blenden Sie im zweiten Editor-Fenster den Code der `LabelViewController`-Klasse ein (falls dieser nicht bereits automatisch angezeigt wird) und ziehen Sie eine Verbindung vom Label in den Code, um das Outlet zu erstellen. Ich nenne das Outlet in diesem Beispiel `label`.

Fügen Sie der Implementierung der `LabelViewController`-Klasse zusätzlich noch eine Property namens `text` vom Typ `String!` hinzu (beachten Sie das Ausrufezeichen, es handelt sich hierbei um ein Implicitly Unwrapped Optional). Diese Property nutzen wir, um den im initialen View-Controller eingegebenen Text an den zweiten View-Controller zu übergeben. Zusätzlich überschreiben wir in der `LabelViewController`-Klasse die Methode `viewWillAppear(_:)` und weisen darin der `label`-Property den Wert der `text`-Property zu. Die vollständige Implementierung von `LabelViewController` finden Sie in Listing 23.14.

Listing 23.14 Implementierung der `LabelViewController`-Klasse

```
class LabelViewController: UIViewController {

    var text: String!

    @IBOutlet weak var label: UILabel!

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        label.text = text
    }

}
```

Bleibt nun noch die Implementierung des initialen View-Controllers, der bereits über die von Xcode erzeugte `ViewController`-Klasse abgebildet und mit dem Interface gekoppelt ist. Im ersten Schritt benötigen wir ein Outlet für das Textfeld, um den eingegebenen Text auslesen und an den zweiten View-Controller übergeben zu können. Der zweite Schritt besteht darin, die genannte Methode `prepare(for:sender:)` zu überschreiben. Sie wird aufgerufen, sobald über den zweiten View-Controller ein Segue ausgelöst wird. Aktuell gibt es nur einen solchen Segue, über den der zweite View-Controller eingeblendet wird.

Die Methode `prepare(for:sender:)` übergibt uns zwei Informationen:

- Eine `UIStoryboardSegue`-Instanz: Der erste Parameter ist vom Typ `UIStoryboardSegue` und hilft uns dabei, Informationen über den ausgelösten Segue zu erhalten. Dazu gehört beispielsweise die Property `destination`, die auf die Instanz des Ziel-View-Controllers verweist (in unserem Fall also auf eine Instanz der `LabelViewController`-Klasse). Da `destination` vom Typ `UIViewController` ist, um alle Arten von View-Controllern darstellen zu können, ist ein Type Casting notwendig, wenn man auf spezifische Informationen des Ziel-View-Controllers zugreifen möchte (so wie in unserem Fall der Zugriff auf die `text`-Property, die exklusiv in `LabelViewController` enthalten ist).
- Den Auslöser des Segues: Der zweite Parameter ist vom Typ `Any` und verweist auf die Instanz, über die der Segue ausgelöst wurde (in unserem Beispiel also die `UIButton`-Instanz). Sie können diesen Parameter nutzen, falls Sie beim Auslösen eines Segues abhängig vom Sender zusätzliche Befehle ausführen oder weitere Informationen auslesen möchten.

In unserer Implementierung der Methode `prepare(for:sender:)` nutzen wir die `destination`-Property des `segue`-Parameters, um auf die Instanz des Ziel-View-Controllers zuzugreifen. Diese casten wir nach `LabelViewController`, da wir wissen, dass es sich bei dem Ziel-View-Controller um eine Instanz dieses Typs handelt. Anschließend weisen wir der `text`-Property des Ziel-View-Controllers den Wert zu, den das Textfeld enthält. Die Klasse `UITextField` stellt zu diesem Zweck ebenfalls eine sogenannte `text`-Property bereit. Die vollständige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.15.

Listing 23.15 Implementierung der `ViewController`-Klasse

```
class ViewController: UIViewController {

    @IBOutlet weak var textField: UITextField!

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        let labelViewController = segue.destination as! LabelViewController
        labelViewController.text = textField.text
    }

}
```

Das war's! Wann immer nun der Segue des initialen View-Controllers ausgeführt wird, wird auch automatisch die überschriebene Methode `prepare(for:sender:)` aufgerufen, die dafür sorgt, den eingegebenen Text an den Ziel-View-Controller zu übergeben (siehe Bild 23.82).

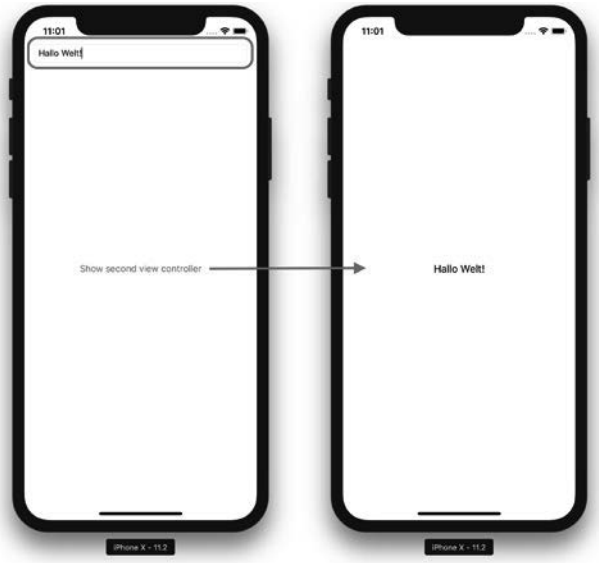


Bild 23.82 Der Text innerhalb des Textfelds wird nun bei Betätigen des Buttons für das Label des Ziel-View-Controllers eingesetzt.

Segue mittels Identifier unterscheiden

In dem gezeigten Beispiel verfügt der initiale View-Controller lediglich über einen Segue. Es gibt allerdings auch Situationen, in denen von einem View-Controller mehr als nur ein Segue ausgeht. Das wäre beispielsweise dann der Fall, wenn ein View-Controller drei unterschiedliche Buttons besitzt und jeder davon einen anderen View-Controller lädt.

Das Problem hierbei ist aber, dass, ganz gleich, *welcher* Segue konkret in einem View-Controller ausgelöst wird, immer nur die Methode `perform(for:sender:)` aufgerufen wird. Das ist insoweit ein Problem, als dass abhängig vom jeweiligen Segue ganz andere Befehle ausgeführt werden müssen und ganz unterschiedliche Ziel-View-Controller zum Einsatz kommen.

Um somit innerhalb der Methode `perform(for:sender:)` unterscheiden zu können, welcher Segue genau ausgelöst wurde, stellt der `segue`-Parameter die Property `identifier` bereit. Sie enthält einen Identifier-String, den man für jeden Segue im Storyboard definieren kann. Um einen solchen Identifier zu setzen, wählen Sie einen Segue (sprich den verbindenden Pfeil) im Storyboard aus und wechseln anschließend in den Identity Inspector. Dort finden Sie ein Feld namens *Identifier*, in dem Sie einen beliebigen String zur eindeutigen Identifikation eines Segues eintragen können (siehe Bild 23.83).

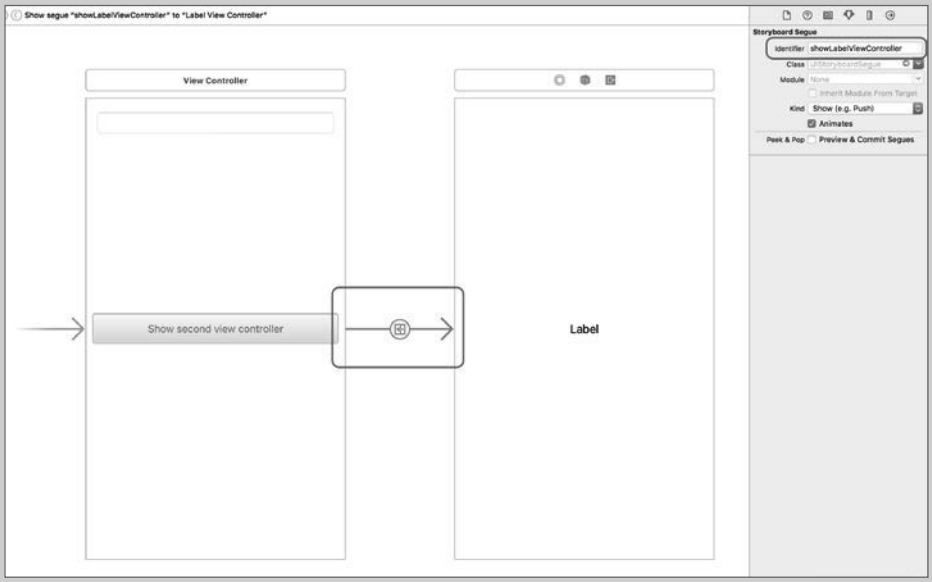


Bild 23.83 Nach Auswahl eines Segues im Storyboard können Sie im Attributes Inspector einen eigenen Identifier dafür festlegen.

Diesen Identifier können Sie nun in der Implementierung der Methode `prepare(for:sender:)` nutzen, um zwischen den verschiedenen Segues

eines View-Controllers zu unterscheiden und abhängig davon die passenden Befehle auszuführen. Listing 23.16 zeigt beispielhaft die Implementierung von `prepare(for:sender:)` aus Listing 23.15 mit zusätzlicher Prüfung des ausgelösten Segues. Dazu wurde im Storyboard für diesen Segue der Identifier *showLabelViewController* festgelegt.

Listing 23.16 Prüfen des Identifiers eines ausgelösten Segues

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showLabelViewController" {
        let labelViewController = segue.destination as!
LabelViewController
        labelViewController.text = textField.text
    }
}
```

Das Casten des Ziel-View-Controllers nach `LabelViewController` und die Übergabe des im Textfeld eingegebenen Textes erfolgen nun nur, wenn tatsächlich der dafür passende Segue im initialen View-Controller ausgelöst wurde.

Generell empfehle ich Ihnen, jeden Segue mit einem solchen eindeutigen Identifier zu versehen und mittels Abfrage beim Auslösen eines Segues zu prüfen, selbst wenn Sie nur einen Segue für einen View-Controller einsetzen. Dann sind Sie bereits optimal gerüstet, sollten langfristig bei der Weiterentwicklung einer App noch weitere Segues hinzukommen. ■

Laden des neuen View-Controller (temporär) verhindern

Mithilfe der `UIViewController`-Methode `shouldPerformSegue(withIdentifier:sender:)` können Sie definieren, ob ein Segue tatsächlich ausgelöst werden soll oder nicht. Sie erwartet einen Rückgabewert vom Typ `Bool`, wobei `true` bedeutet, dass der Segue die ihm zugewiesene Aufgabe ausführt (zum Beispiel das Laden eines neuen View-Controllers), während mit `false` die Ausführung des Segues verhindert wird.

Auch wenn standardmäßig gesetzte Segues selbstverständlich auch ausgeführt werden sollen, gibt es Situationen, in denen ein Segue zumindest temporär inaktiv sein sollte. Stellen Sie sich beispielsweise eine Login-Maske vor, die erst dann den Login-Prozess nach Tippen auf einen entsprechenden Button auslösen soll, wenn der Nutzer auch einen sinnvollen Wert in die Textfelder für Benutzername und Passwort eingetragen hat. Der Segue, der durch Tippen auf den Button gefeuert wird, soll entsprechend nur dann genutzt werden können, wenn diese Rahmenbedingungen erfüllt sind.

Demonstriert werden soll diese Technik anhand eines kleinen Beispiels. Die grundlegende Basis ist identisch mit der der vorangegangenen Beispiele: Ein neues iOS-Projekt auf Basis der *Single View App*-Vorlage erhält einen zweiten View-Controller mit einem statischen Label (Text: „Second view controller“). Der initiale View-Controller erhält einen Button mit dem Titel „Show second view controller“, der mittels eines *Show*-Segues mit dem zweiten View-Controller verbunden wird. Der Segue erhält zusätzlich den Identifier „showSecondViewController“.

Anschließend fügen Sie dem initialen View-Controller noch an einer beliebigen Stelle einen Switch hinzu. Ein Switch basiert auf der Klasse `UISwitch` und stellt einen Schalter dar, der nur zwei Zustände kennt: an oder aus. Bild 23.84 zeigt, wie das Switch-Element in der Objects Library aussieht. Und Bild 23.85 zeigt, wie das Interface dieser Beispiel-App am Ende in etwa aussehen könnte.

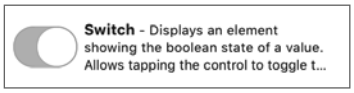


Bild 23.84
Ein Switch stellt ein View-Element in Form eines Schalters dar.

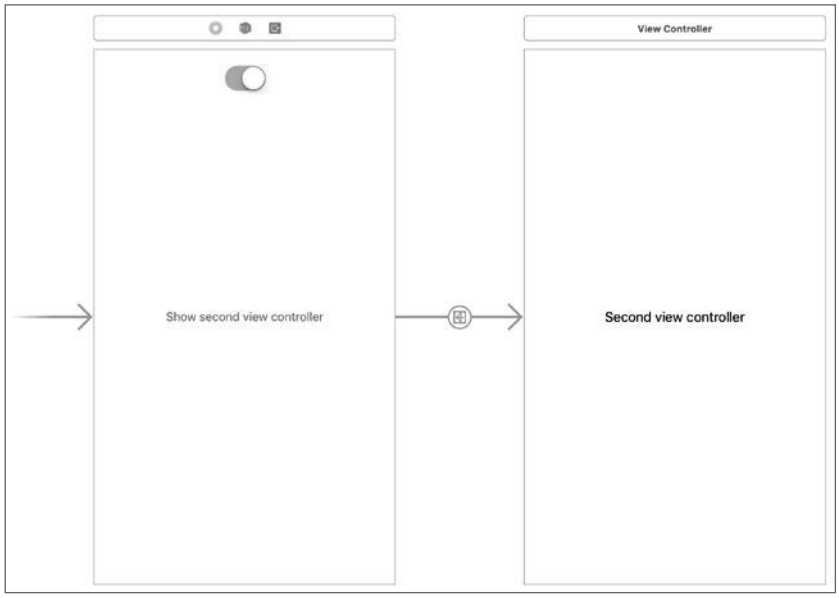


Bild 23.85 Die Beispiel-App verfügt über zwei View-Controller, die mit einem Segue miteinander verbunden sind. Ein Switch im initialen View-Controller soll steuern, ob der Segue aktiv ist oder nicht.

Die Implementierung der Klasse des initialen View-Controllers soll bei Durchführung des Segues nun prüfen, ob der Switch aktiv ist oder nicht. Nur wenn er aktiv ist, soll der Segue wie gewohnt ausgeführt werden, andernfalls nicht. Dazu muss zunächst ein Outlet für die `UISwitch`-Instanz erzeugt und anschließend die vorgestellte Methode `shouldPerformSegue(withIdentifier:sender:)` überschrieben werden. Innerhalb dieser Methode prüfen wir zunächst mithilfe des `identifier`-Parameters, ob es sich bei dem ausgelösten Segue um den mit dem von uns gesetzten Identifier „showSecondViewController“ handelt. Ist das der Fall, prüfen wir mithilfe der Property `isOn`, ob der Switch aktiv ist (`true`) oder nicht (`false`). Ist er nicht aktiv, liefern wir in der Methode `false` zurück und geben damit an, dass der Segue nicht ausgeführt werden darf. In jedem anderen Fall (auch falls ein anderer Segue ausgelöst werden sollte) liefern wir `true` zurück. Die vollständige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.17.

Listing 23.17 Prüfung, ob Segue ausgeführt werden soll oder nicht

```
class ViewController: UIViewController {

    @IBOutlet weak var segueSwitch: UISwitch!

    override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?)
-> Bool {
        if identifier == "showSecondViewController" {
            if !segueSwitch.isOn {
                return false
            }
        }
        return true
    }

}
```

Probieren Sie es selbst einmal aus! Solange der Schalter nicht aktiv ist, wird auch keine Aktion bei Betätigen des Buttons ausgeführt; es ist, als wäre der Segue gar nicht da. Erst wenn der Switch aktiviert ist, lässt sich der Segue durch Betätigen des Buttons wie gewohnt auslösen.

Segue programmatisch ausführen

Segues besitzen den großen Vorteil, dass sie einmalig über ein Storyboard konfiguriert werden und dann automatisch im passenden Kontext (zum Beispiel durch Betätigen eines Buttons wie in den vorangegangenen Beispielen) ausgelöst werden. In manchen Fällen reicht es aber nicht, die mit dem Segue verbundene Aktion auf diesen einen Kontext zu beschränken. Es kann noch weitere Ereignisse geben, die möglicherweise auch nichts mit dem Interface zu tun haben, dennoch aber dieselbe Aktion auslösen sollen.

Zu diesem Zweck stellt die `UIViewController`-Klasse die Methode `performSegue(withIdentifier:sender:)` zur Verfügung. Sie kann von einem View-Controller aufgerufen werden, um einen Segue mit einem spezifischen Identifier auszuführen. Dieser Identifier wird in Form des ersten Parameters der Methode übergeben. Zusätzlich kann man noch mithilfe des `sender`-Parameters die Instanz übergeben, die den Segue ausgelöst hat, muss das aber nicht.

Anhand eines einfachen Beispiels soll der Einsatz der Methode `performSegue(withIdentifier:sender:)` praktisch verdeutlicht werden. Die Basis ist erneut eine *Single View App*, deren *Main.storyboard*-Datei über zwei View-Controller verfügt. Der initiale View-Controller besitzt einen Button mit dem Titel „Show second view controller“, der mit einem *Show*-Segue mit dem zweiten View-Controller verbunden ist. Dieser Segue erhält den Identifier „showSecondViewController“. Der zweite View-Controller besitzt ein Label mit dem Titel „Second view controller“. Bild 23.86 zeigt diesen grundlegenden Aufbau der Beispiel-App.

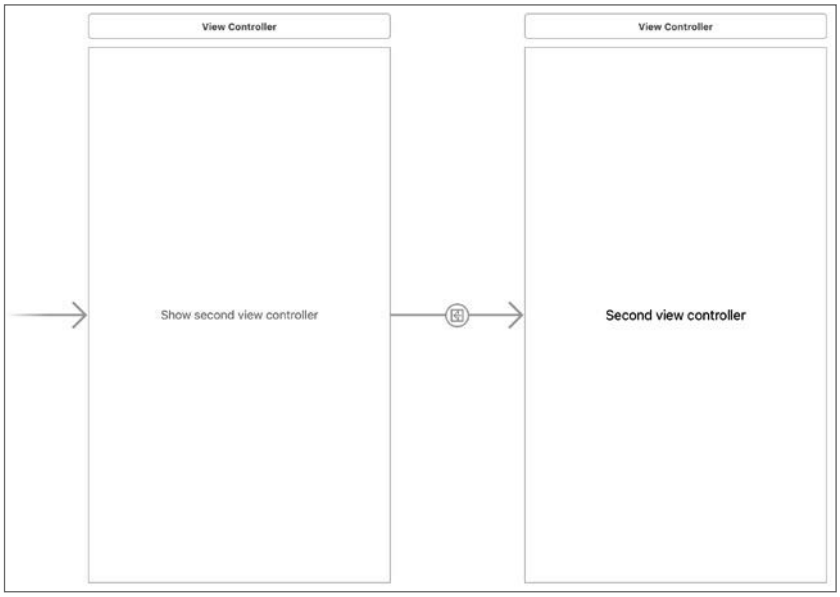


Bild 23.86 Die Beispiel-App setzt sich erneut aus zwei View-Controllern zusammen, die mithilfe eines Show-Segues mit dem Identifier „showSecondViewController“ miteinander verbunden sind.

Um die Verwendung der Methode `performSegue(withIdentifier:sender:)` zu demonstrieren, soll direkt nach Erscheinen des initialen View-Controllers der zweite View-Controller eingeblendet werden. Zu diesem Zweck überschreiben wir in der Implementierung der `ViewController`-Klasse die Methode `viewDidAppear(_:)` und rufen darin die genannte Methode auf. Als `identifier`-Parameter übergeben wir „showSecondViewController“, was dem gewünschten Segue-Identifier entspricht. Der `sender`-Parameter spielt für dieses Beispiel keine Rolle und wird daher auf `nil` gesetzt. Die vollständige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.18.

Listing 23.18 Programmatische Ausführung eines Segues

```
class ViewController: UIViewController {

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        performSegue(withIdentifier: "showSecondViewController", sender: nil)
    }

}
```

Wenn Sie dieses Projekt nun ausführen, wird direkt nach Erscheinen des initialen View-Controllers automatisch der im Storyboard definierte Segue ausgeführt und der zweite View-Controller eingeblendet.

23.5.5.2 Über den Code

Auch wenn das Einblenden neuer View-Controller über das Storyboard ein komfortabler Weg ist, ist er in manchen Situationen nicht dynamisch und flexibel genug. Manchmal

bestimmen zusätzliche Umstände, ob und welcher View-Controller angezeigt werden soll, und die statische Lösung der Storyboards passt nicht mehr.

Aus diesem Grund ist es problemlos möglich, mithilfe passender Methoden jederzeit aus dem Code heraus einen neuen View-Controller einzublenden. Dazu müssen Sie aber im ersten Schritt überhaupt eine Instanz eines solchen View-Controllers im Code besitzen. Eine solche erhalten Sie auf zwei möglichen Wegen:

- Erzeugen einer `UIViewController`-Instanz (oder einer `UIViewController`-Subklasse) im Code
- Auslesen eines in einem Storyboard erzeugten View-Controllers

Bevor wir uns damit beschäftigen, *wie* man einen View-Controller aus dem Code heraus lädt und anzeigt, stelle ich Ihnen zunächst die beiden genannten Möglichkeiten zum Erzeugen neuer View-Controller im Code vor.

Erzeugen einer UIViewController-Instanz

`UIViewController` ist die Klasse, zu der alle View-Controller in der iOS-Entwicklung gehören. Auch eigene View-Controller sollten immer von `UIViewController` abgeleitet sein.

Möchten Sie in Ihrem Projekt einen neuen View-Controller im Code erzeugen, ist die Vorgehensweise an sich sehr simpel: Sie nutzen den Default Initializer von `UIViewController` und erhalten eine voll nutzbare View-Controller-Instanz. Diese können Sie dann Ihren Wünschen entsprechend konfigurieren, beispielsweise um die Hintergrundfarbe der zugrunde liegenden View zu verändern. Listing 23.19 zeigt genau dieses Beispiel.

Listing 23.19 Erzeugen einer `UIViewController`-Instanz und verändern der Hintergrundfarbe

```
let myViewController = UIViewController()
myViewController.view.backgroundColor = .red
```

Auslesen eines View-Controllers aus dem Storyboard

Auch wenn die View-Controller eines Storyboards direkt mithilfe von Segues miteinander verbunden werden können, ist es in manchen Situationen sinnvoll, eine Instanz eines in einem Storyboard erstellten View-Controllers im Code zu erzeugen. Nehmen wir als Beispiel einen View-Controller zur Abbildung einer Anmeldemaske, um auf zusätzliche Services einer App zugreifen zu können. Womöglich führt von den Einstellungen aus direkt ein Segue zu diesem View-Controller, um sich anmelden zu können, aber derselbe View-Controller soll auch geladen und angezeigt werden, wenn der Nutzer in der App einen Service auswählt, für den eine Authentifizierung notwendig ist. In diesem Fall kann man beim Aufruf des entsprechenden Service prüfen, ob der Nutzer angemeldet ist, und falls nicht, den passenden View-Controller für die Anmeldung laden und programmatisch einblenden.

Um einen View-Controller aus einem Storyboard im Code zu laden und zu initialisieren, muss man zunächst eine Instanz der gewünschten Storyboard-Datei selbst erzeugen. Das erfolgt mithilfe der Klasse `UIStoryboard` und des Initializers `init(name:bundle:)`. Der erste Parameter des Initializers erwartet den Namen der Storyboard-Datei, die den gewünschten View-Controller enthält. Der zweite Parameter verweist auf das Bundle, zu dem das Storyboard gehört. Handelt es sich hierbei – wie das standardmäßig der Fall ist – um das Main Bundle, kann für diesen Parameter schlicht `nil` übergeben werden (mehr zum Thema Bundle erfahren Sie in Kapitel 28, „Cross-Platform“).

Auf dieser `UIStoryboard`-Instanz können nun zwei Methoden aufgerufen werden, um einen View-Controller daraus zu laden und zu initialisieren. Die erste lautet `instantiateInitialViewController()` und liefert eine neue Instanz des initialen View-Controllers zurück. Die zweite Methode hört auf den Namen `instantiateViewController(withIdentifier:)` und wird benötigt, sobald sie einen anderen als den initialen View-Controller eines Storyboards laden und erzeugen möchten. Die Methode erwartet einen `identifier`-Parameter, über den Sie definieren, *welchen* View-Controller aus dem zugrunde liegenden Storyboard Sie laden möchten. Einen solchen Identifier legen Sie selbst fest, indem Sie im Storyboard einen View-Controller auswählen und anschließend in den Identity Inspector wechseln. Dort finden Sie ein Textfeld mit dem Titel *Storyboard ID* (siehe Bild 23.87). Den Wert, den Sie in dieses Feld eintragen, nutzen Sie, um den entsprechenden View-Controller mithilfe der Methode `instantiateViewController(withIdentifier:)` im Code zu laden.

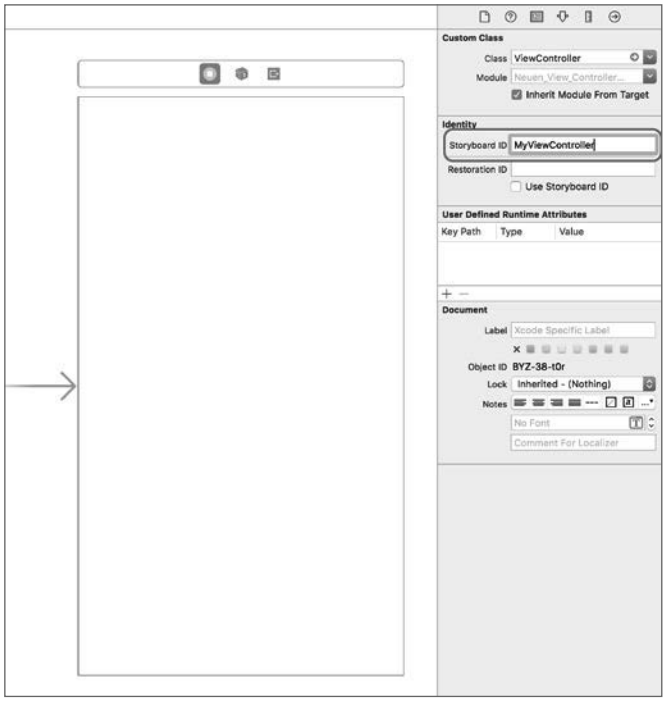


Bild 23.87 Dem hier gezeigten View-Controller wurde der Identifier MyViewController zugewiesen.

In Listing 23.20 sehen Sie ein Beispiel zur Verwendung von `UIStoryboard`. Dort wird eine Instanz für die *Main.storyboard*-Datei erzeugt und anschließend ein View-Controller mit dem Identifier `MyViewController` geladen und initialisiert.

Listing 23.20 Laden und Erzeugen eines View-Controllers aus dem Storyboard heraus

```
let mainStoryboard = UIStoryboard(name: "Main", bundle: nil)
let myViewController = mainStoryboard.instantiateViewController(withIdentifier:
"MyViewController")
```

Type Casting beachten

Bei dem gezeigten Laden und Erzeugen von View-Controllern mithilfe der Klasse `UIStoryboard` müssen Sie ein wichtiges Detail beachten: Sowohl die Methode `instantiateInitialViewController()` als auch die Methode `instantiateViewController(withIdentifier:)` liefern eine Instanz der Klasse `UIViewController` zurück. In den meisten Fällen möchte man aber eine explizite Subklasse ansprechen, die man dem entsprechenden View-Controller im Identity Inspector des Storyboards zugewiesen hat. So entspricht bei neu erstellten iOS-Projekten auf Basis einer *Single View App* der initiale View-Controller der Klasse `ViewController`, die von `UIViewController` abgeleitet ist und über zusätzliche Eigenschaften und Funktionen verfügen kann, die `UIViewController` nicht besitzt.

Um solch zusätzliche Eigenschaften und Funktionen nutzen zu können, ist es in diesen Fällen daher notwendig, die erhaltene `UIViewController`-Instanz in den gewünschten Typ zu casten. Listing 23.21 zeigt ein Beispiel dazu, in dem ein View-Controller mit dem Identifier *LabelViewController* aus dem Main-Storyboard geladen wird. Dieser View-Controller entspricht einem Typ der gleichnamigen Klasse `LabelViewController`. Um alle Eigenschaften und Funktionen dieser Klasse nutzen zu können, muss das erhaltene Ergebnis der Methode `instantiateViewController(withIdentifier:)` entsprechend gecastet werden.

Listing 23.21 Casten eines über ein Storyboard geladenen View-Controllers

```
let mainStoryboard = UIStoryboard(name: "Main", bundle: nil)
let labelViewController = mainStoryboard.instantiateViewController(with
Identifier: "LabelViewController") as! LabelViewController
```

Anzeigen eines neuen View-Controllers im Code

Nachdem Sie nun wissen, wie Sie View-Controller im Code erzeugen können, betrachten wir das Vorgehen, um einen neuen View-Controller aus dem Code heraus anzuzeigen. Basis hierfür ist die Methode `present(_:animated:completion:)` der Klasse `UIViewController`. Sie blendet einen View-Controller modal ein (sprich sie legt ihn über den aktuell sichtbaren View-Controller) und entspricht damit dem, was wir zuvor im Storyboard mithilfe des *Show*-Segues umgesetzt haben. Die Methode erwartet die folgenden drei Parameter:

- `viewControllerToPresent`: Die `UIViewController`-Instanz, die angezeigt werden soll.
- `flag`: Ein boolescher Wert, der bestimmt, ob der View-Controller animiert eingeblendet werden soll (`true`) oder nicht (`false`).
- `completion`: Ein Closure, das ausgeführt wird, sobald der Ziel-View-Controller eingeblendet wurde. Es erlaubt es Ihnen, zusätzliche Befehle auszuführen (falls gewünscht). Alternativ können Sie auch `nil` für diesen Parameter übergeben.

Betrachten wir einmal die praktische Verwendung der Methode `present(_:animated:completion:)` auf Basis eines neuen iOS-Projekts mit der *Single View App*-Vorlage. Diesem wird zunächst im initialen View-Controller an einer beliebigen Stelle ein Button mit dem

Titel „Show second view controller“ hinzugefügt und mit einer Action-Methode namens `showSecondViewController()` im Code der `ViewController`-Klasse gekoppelt (die Methode braucht in diesem Beispiel weder einen `sender`- noch einen `events`-Parameter, siehe Bild 23.88).

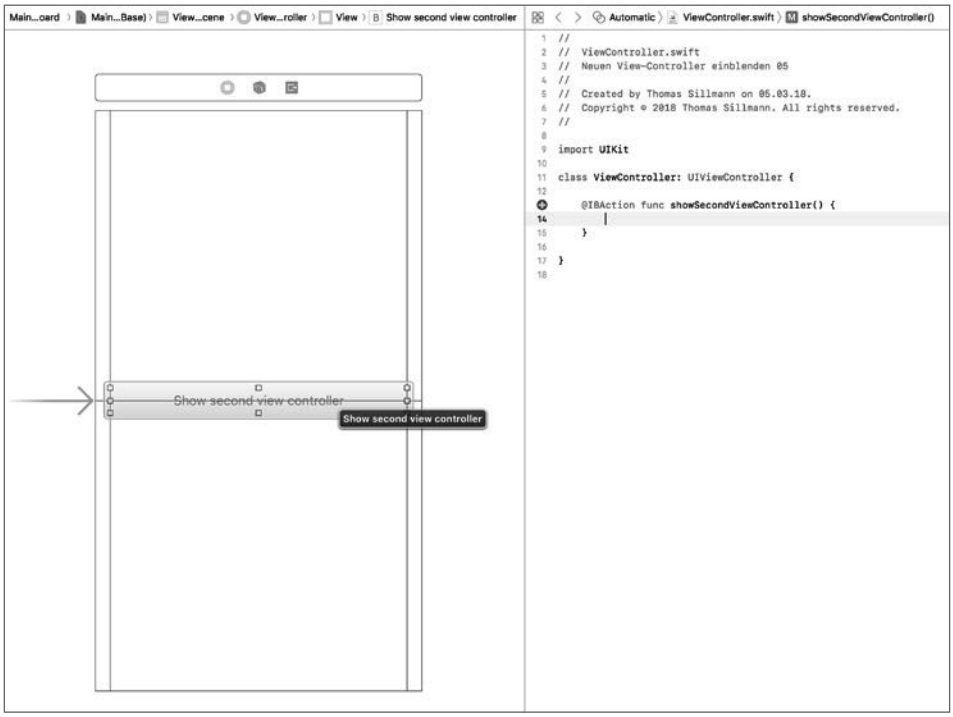


Bild 23.88 Der initiale View-Controller der Beispiel-App verfügt über einen Button, der mit einer Action-Methode namens „showSecondViewController()“ mit dem Code gekoppelt ist.

Innerhalb der Methode `showSecondViewController()` wird nun eine neue `UIViewController`-Instanz erzeugt und deren View ein blauer Hintergrund zugewiesen. Anschließend wird diese Instanz mithilfe der Methode `present(_:animated:completion:)` eingeblendet, wenn der Button betätigt wird. Die vollständige Implementierung der Methode `showSecondViewController()` finden Sie in Listing 23.22.

Listing 23.22 Einblenden eines neuen View-Controllers

```
@IBAction func showSecondViewController() {
    let blueBackgroundViewController = UIViewController()
    blueBackgroundViewController.view.backgroundColor = .blue
    present(blueBackgroundViewController, animated: true, completion: nil)
}
```

Wenn Sie das Projekt nun ausführen und den Button betätigen, wird ein neuer View-Controller mit blauem Hintergrund geladen und angezeigt.

Betrachten wir alternativ dazu noch ein Beispiel, in dem ein View-Controller aus einem Storyboard geladen und angezeigt wird. Erstellen Sie hierfür ein weiteres neues iOS-Projekt

auf Basis einer *Single View App* und gehen Sie zunächst genauso vor wie beim vorangegangenen Beispiel: Fügen Sie dem initialen View-Controller einen Button mit dem Titel „Show second view controller“ hinzu und verknüpfen Sie ihn mit der Action-Methode `showSecondViewController()` mit dem Code der `ViewController`-Klasse (verzichten Sie aber noch auf die Implementierung dieser Methode).

Fügen Sie anschließend der *Main.storyboard*-Datei einen zweiten View-Controller hinzu und versehen Sie diesen an einer beliebigen Stelle mit einem Label, das den Text „Second view controller“ anzeigt. Wählen Sie diesen neuen View-Controller anschließend aus und wechseln Sie in den Identity Inspector. Tragen Sie dort in das Feld *Storyboard ID* den Wert *MyViewController* ein (siehe Bild 23.89).

Bild 23.89 Weisen Sie dem neu hinzugefügten View-Controller einen passenden Identifier zu, um ihn so aus dem Code heraus ansprechen zu können.

Damit befinden sich in der *Main.storyboard*-Datei nun zwei View-Controller, die aber nicht durch einen Segue miteinander verbunden sind (siehe Bild 23.90).

Bild 23.90 Die Beispiel-App verfügt über zwei voneinander unabhängige und nicht miteinander verbundene View-Controller.

Mit diesen getroffenen Vorkehrungen kehren wir in den Code der `ViewController`-Klasse und zur Implementierung der `showSecondViewController()`-Methode zurück. Unsere Aufgabe besteht nun darin, den zweiten im Storyboard erstellten View-Controller mit dem Identifier *MyViewController* im Code zu laden und mithilfe der Methode `present(_:animated:completion:)` einzublenden. Dazu erstellen wir zunächst eine `UIStoryboard`-Instanz auf Basis der *Main.storyboard*-Datei und nutzen anschließend die Methode `instantiateViewController(withIdentifier:)`, um eine Instanz des zweiten View-Controllers zu erhalten und diese anzuzeigen. Die passende Implementierung der Methode `showSecondViewController()` finden Sie in Listing 23.23.

Listing 23.23 Laden und Anzeigen eines View-Controllers aus einem Storyboard

```
@IBAction func showSecondViewController() {
    let mainStoryboard = UIStoryboard(name: "Main", bundle: nil)
    let secondViewController = mainStoryboard.instantiateViewController
(withIdentifier: "MyViewController")
    present(secondViewController, animated: true, completion: nil)
}
```

Alternative: show(_:sender:)

Neben der in diesem Abschnitt vorgestellten Methode `present(_:animated:completion:)` können Sie auch die – ebenfalls in der Klasse `UIViewController` implementierte – Methode `show(_:sender:)` zum Anzeigen eines neuen View-Controllers verwenden. Als ersten Parameter übergeben Sie den anzuzeigenden Ziel-View-Controller (genau wie bei `present(_:animated:completion:)` auch) und definieren optional noch einen Sender – sprich Aufrufer – der Methode in Form des zweiten Parameters.

Der Unterschied von `show(_:sender:)` besteht darin, dass sie abhängig vom Kontext die Anzeige des Ziel-View-Controllers nicht modal, sondern auf alternative Art und Weise realisiert. Wenn Sie diese Methode beispielsweise über einen `UINavigationController` aufrufen, wird der Ziel-View-Controller auf dem Navigation Stack gepusht.

View-Controller-Klassen wie `UINavigationController` überschreiben diese Methode und implementieren die gewünschte Art, wie neue View-Controller in ihrem Kontext standardmäßig angezeigt werden sollen. Man muss sich dann bei Verwendung von `show(_:sender:)` keine Gedanken mehr darüber machen, ob ein View-Controller nun besser modal eingeblendet oder auf einem möglichen Navigation Stack gepusht werden soll; anhand des Kontexts wird diese Entscheidung durch den zugrunde liegenden View-Controller gefällt.

Wenn Sie also einen View-Controller definitiv *immer* modal einblenden möchten (unabhängig vom Kontext), müssen Sie die in diesem Abschnitt vorgestellte Methode `present(_:animated:completion:)` verwenden.

Mehr über weitere verfügbare View-Controller wie `UINavigationController` und die Arbeit mit ihnen erfahren Sie in Kapitel 24, „iOS – App-Entwicklung“. ■

23.5.5.3 View-Controller ausblenden

In den vorherigen Abschnitten haben Sie die unterschiedlichen Vorgehensweisen kennengelernt, mit denen man einen neuen View-Controller in der iOS-Entwicklung einblenden kann. Zum Abschluss möchte ich Ihnen nun noch zeigen, wie Sie einen View-Controller wieder *ausblenden* können. Denn wie Ihnen womöglich in den letzten Beispielen aufgefallen ist, konnten Sie zwar immer einen neuen View-Controller anzeigen, hatten dann aber keine Chance mehr, zum ursprünglichen View-Controller zurückzukehren, ohne die App komplett zu beenden und anschließend wieder neu zu starten.

Das Ausblenden von View-Controllern können Sie ausschließlich über den Code steuern; aus einem Storyboard heraus können Sie eine solche Aktion nicht auslösen. Basis zum Ausblenden eines View-Controllers ist die Methode `dismiss(animated:completion:)` der Klasse `UIViewController`. Sie wird auf der View-Controller-Instanz aufgerufen, die Sie ausblenden möchten. Der erste Parameter vom Typ `Bool` bestimmt hierbei, ob das Ausblenden des View-Controllers animiert erfolgen soll (`true`) oder nicht (`false`). Beim zweiten Parameter handelt es sich um ein Closure, das ausgeführt wird, sobald der View-Controller ausgeblendet wurde. Er ist optional und kann somit auch `nil` entsprechen.

Ein Beispiel soll die Verwendung von `dismiss(animated:completion:)` praktisch verdeutlichen. Ausgangspunkt hierfür ist ein neues iOS-Projekt auf Basis der *Single View App*-Vorlage. Erstellen Sie zunächst einen zweiten View-Controller in der *Main.storyboard*-Datei und fügen Sie diesem an einer beliebigen Stelle einen Button mit dem Titel „Dismiss“ hinzu. Fügen Sie anschließend noch dem initialen View-Controller ebenfalls einen Button mit dem Titel „Show“ hinzu und setzen Sie von dort aus einen *Show*-Segue auf den zweiten View-Controller, sodass per Tippen auf diesen Button der zweite View-Controller eingeblendet wird.

Die Aufgabe des „Dismiss“-Buttons besteht darin, den zweiten View-Controller wieder auszublenden. Da das – wie eingangs beschrieben – nur über den Code mithilfe der Methode `dismiss(animated:completion:)` möglich ist, erstellen wir zunächst eine neue `UIViewController`-Subklasse namens `DismissViewController`, die wir im Storyboard dem zweiten View-Controller im Identity Inspector zuweisen (siehe Bild 23.91).

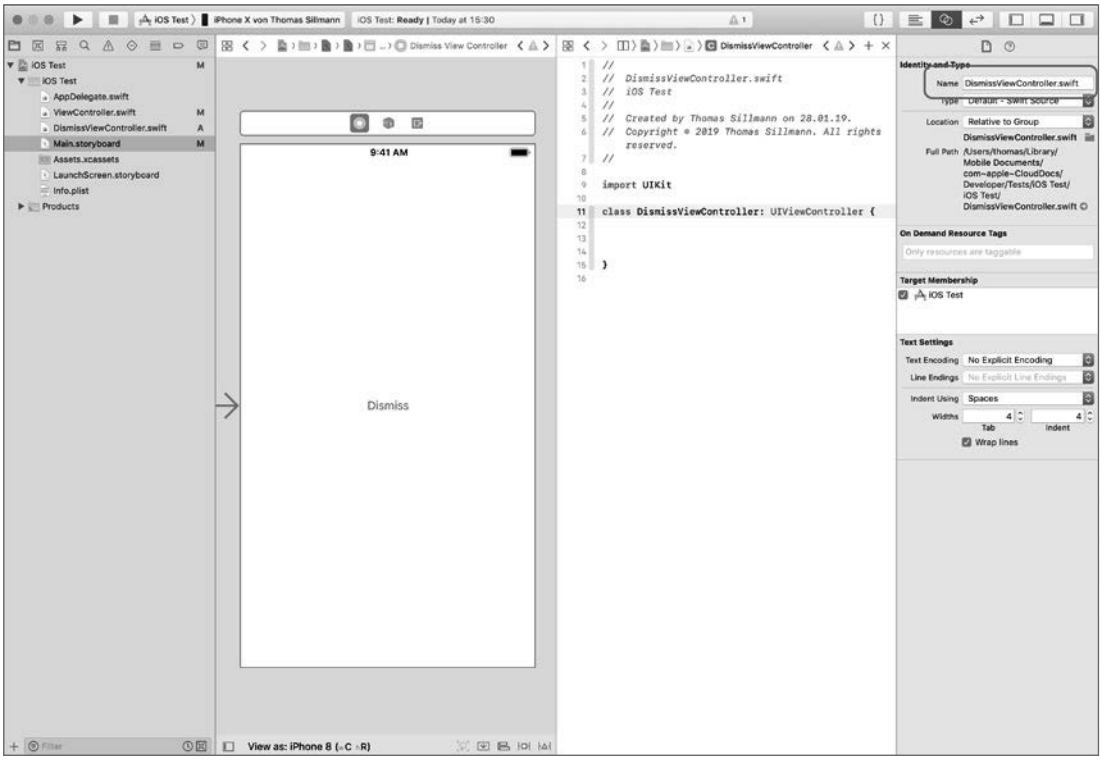


Bild 23.91 Erstellen Sie eine neue Klasse namens DismissViewController und weisen Sie sie dem zweiten View-Controller im Storyboard zu.

Wechseln Sie anschließend in den Assistant Editor und lassen Sie Storyboard und den Code der `DismissViewController`-Klasse nebeneinander anzeigen (so wie in Bild 23.91 zu sehen). Erstellen Sie dann eine Action-Methode namens `dismiss()` für den Button und rufen Sie darin `dismiss(animated:completion:)` auf. Das sorgt dafür, dass der zweite, modal eingeblendete View-Controller wieder ausgeblendet wird, sobald der „Dismiss“-

Button betätigt wird. Die vollständige Implementierung der `DismissViewController`-Klasse finden Sie in Listing 23.24.

Listing 23.24 Ausblenden eines View-Controllers mithilfe der Methode `dismiss(animated:completion:)`

```
class DismissViewController: UIViewController {

    @IBAction func dismiss() {
        dismiss(animated: true, completion: nil)
    }

}
```

Mithilfe der Methode `dismiss(animated:completion:)` können Sie somit einen modal angezeigten View-Controller wieder ausblenden. Falls Sie die Methode auf einen View-Controller aufrufen, der nicht modal angezeigt wird, passiert schlicht gar nichts.

23.6 Oberflächen gestalten mit UIView

Eine View entspricht einer Ansicht einer iOS-App, die dem Nutzer auf dem Bildschirm angezeigt wird. Sowohl der gesamte Bildschirminhalt wie auch einzelne Elemente davon entsprechen einer View.

In den vorangegangenen Abschnitten haben wir bereits verschiedene solcher Views kennengelernt: Es gibt Labels, um Textinformationen darzustellen, Buttons, um Aktionen auszulösen und Schalter, die man aktivieren oder deaktivieren kann. All diese einzelnen Elemente sind Views, und es gibt im UIKit-Framework noch viele mehr, die wir bei der Entwicklung von iOS-Apps einsetzen können.

Die Basis aller Views in der iOS-Entwicklung stellt die Klasse `UIView` dar. Sie besitzt grundlegende Eigenschaften und Funktionen, die zwingend für die Arbeit und die Verwendung von Views benötigt werden. Was die Klasse `UIViewController` für View-Controller ist, ist `UIView` für Views. Alle View-Elemente wie Labels oder Buttons sind von dieser Klasse abgeleitet.

23.6.1 Aufbau von Views

Eine View stellt eine bestimmte Ansicht dar. Eine solche Ansicht wiederum kann selbst weitere Views beinhalten – sogenannte *Subviews*. Eine solche Komposition aus einer View und ihren Subviews kann ganz individuelle und einzigartige Ansichten zutage fördern.

Ein gutes Beispiel für derartige Kompositionen sind die bereits in vorherigen Abschnitten vorgestellten View-Controller. Die Basis eines jeden View-Controllers ist eine View, die alle Inhalte und Elemente des jeweiligen View-Controllers anzeigt. Über die Property `view` – die dem Typ `UIView` entspricht – kann man bei jedem View-Controller auf dessen Ansicht zugreifen (beispielsweise um die Hintergrundfarbe dieser View zu verändern).

Bei der Arbeit mit Storyboards haben wir eine solche View eines View-Controllers bereits angepasst, indem wir zusätzliche *Subviews* hinzugefügt haben – meist Labels oder Buttons. Sie waren Teil der eigentlichen View des View-Controllers und wurden *auf ihr* platziert. Anders ausgedrückt handelte es sich in diesen Szenarios bei den Labels und Buttons um *Subviews* der View des View-Controllers. Umgekehrt handelt es sich bei der View des View-Controllers aus Sicht von Label oder Button um deren *Superview*, also um diejenige View, der sie hinzugefügt wurden. In Bild 23.92 ist dieser grundlegende Aufbau von Views zum besseren Verständnis illustriert.

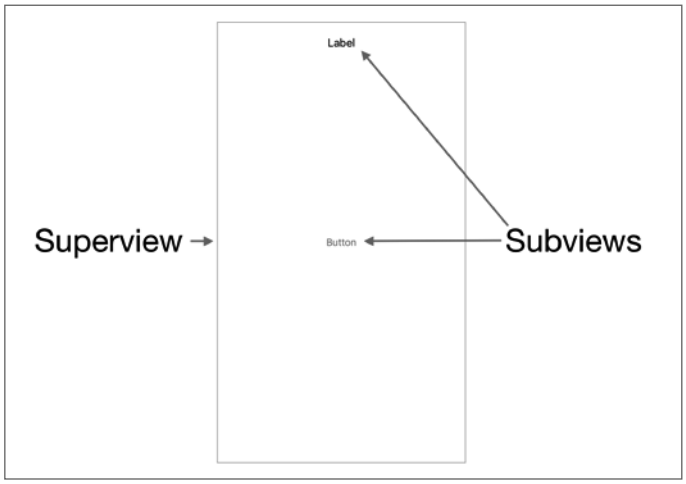


Bild 23.92 Die umgebende View ist die Superview, die in ihr platzierten Elemente (in diesem Fall ein Label und ein Button) stellen deren Subviews dar.

23.6.2 Erstellen von Views

Bisher haben wir Views immer direkt über das Storyboard erstellt, um das Aussehen und den Aufbau eines View-Controllers zu definieren. Zu diesem Zweck stellt uns jeder View-Controller in einem Storyboard die zugehörige Superview bereit, auf der wir dann nach Belieben die benötigten Subviews positionieren und anordnen können.

Alternativ dazu können Views aber auch jederzeit dynamisch im Code erzeugt werden. Dazu erstellt man eine Instanz der gewünschten View und fügt sie an der Stelle ein, wo man sie haben möchte. Das kann entweder eine andere View oder die Superview eines View-Controllers sein.

Wichtig beim Erstellen einer jeden View ist der sogenannte *Frame*. Er definiert zwei essenzielle Angaben für jede View:

- X- und Y-Koordinate: Die Koordinaten bestimmten die Position der View im Verhältnis zu ihrer Superview.
- Breite und Höhe: Sie bestimmen die Größe der View.

Ein solcher Frame wird in der iOS-Entwicklung mithilfe des Typs `CGRect` abgebildet. Beim Erstellen einer Instanz dieses Typs gibt man die Informationen zu X- und Y-Koordinate

sowie Höhe und Breite an. Bei der Initialisierung der eigentlichen View nutzt man dann diese Information, um Größe und Position der View festzulegen. Die Klasse `UIView` bringt bereits einen Initializer namens `init(frame:)` mit, der einen solchen Frame in Form einer `CGRect`-Instanz als Parameter erwartet.

In Listing 23.25 sehen Sie ein Beispiel zur Erstellung eines Frames und einer View. Der View wird hierbei eine Größe von 200 × 200 Punkten und eine X- und Y-Koordinate mit dem Wert 50 zugewiesen.

Listing 23.25 Erstellen einer View im Code

```
// Erstellen des Frames für die zu erzeugende View.
let myViewFrame = CGRect(x: 50, y: 50, width: 200, height: 200)

// Erstellen einer View.
let myView = UIView(frame: myViewFrame)
```

Punkte vs. Pixel

Die Größen- und Koordinatenangaben in Form eines `CGRect` werden intern in Form von Punkten abgebildet. Je nach zugrunde liegendem Endgerät und Display kann ein Punkt ein oder mehrere Pixel enthalten. Würde man sich demzufolge bei der Berechnung von Größen auf Pixel verlassen, würden Views auf einem höher auflösenden Gerät verhältnismäßig kleiner erscheinen als auf einem Device mit niedrigerer Bildschirmauflösung. Damit Ansichten von iOS-Apps aber auf allen aktuellen und kommenden Geräten möglichst identisch aussehen, setzt `CGRect` auf Punkte, die den für den Nutzer sichtbaren Bildschirminhalt widerspiegeln. ■

Möchte man eine so erzeugte View einer anderen View als Subview hinzufügen, kommt hierfür die Methode `addSubview(_:)` der `UIView`-Klasse zum Einsatz. Sie wird auf der View aufgerufen, der eine Subview mit Position und Größe des definierten Frames hinzugefügt werden soll. Diese Subview wird als Parameter übergeben.

Ein kleines Beispiel soll das Erstellen und Hinzufügen von Views im Code einmal praktisch verdeutlichen. Ausgangspunkt ist ein iOS-Projekt auf Basis einer *Single View App*. Wechseln Sie darin in die standardmäßig erzeugte `ViewController`-Klasse und überschreiben Sie in dieser die Methode `viewDidLoad()`. Innerhalb der Methode erstellen wir nun eine neue View namens `redCube`, die eine Größe von 200 x 200 Punkten besitzt und deren X- und Y-Koordinate 50 entspricht. Der so erzeugten View wird eine rote Hintergrundfarbe zugewiesen. Abschließend wird die neu erstellte `redCube`-View der zugrunde liegenden View des View-Controllers als Subview hinzugefügt.

Die vollständige Implementierung der ViewController-Klasse finden Sie in Listing 23.26.

Listing 23.26 Erstellen und Hinzufügen einer View im Code

```
class ViewController: UIViewController {

    override func viewDidLoad() {
```

```
        super.viewDidLoad()

        // Erstellen des Frames für die neue Subview
        let redCubeFrame = CGRect(x: 50, y: 50, width: 200, height: 200)

        // Erstellen der View auf Basis des zuvor erzeugten Frames.
        let redCube = UIView(frame: redCubeFrame)

        // Setzen einer roten Hintergrundfarbe für die neue View
        redCube.backgroundColor = .red

        // Hinzufügen der neuen View zur zugrunde liegenden View des View-Controllers
        view.addSubview(redCube)
    }

}
```

Wenn Sie das fertige Projekt ausführen, wird beim Laden des initialen View-Controllers der Code innerhalb der Methode `viewDidLoad()` ausgeführt. Dann wird somit die neue View (`redCube`) erzeugt und der View des View-Controllers als Subview zugewiesen. Das Ergebnis im iPhone X-Simulator sehen Sie in Bild 23.93.

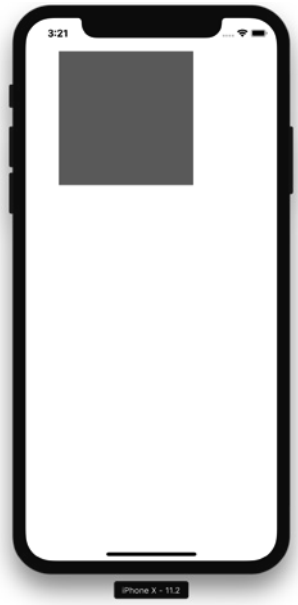


Bild 23.93
Im Code wurde eine neue View erstellt und dem View-Controller programmatisch hinzugefügt.

Views entfernen

Views können nicht nur auf die gezeigte Art und Weise hinzugefügt, sondern auch wieder entfernt werden. Hierfür kommt die Methode `removeFromSuperview()` der `UIView`-Klasse zum Einsatz. Die View, auf der sie aufgerufen wird, wird entfernt und damit ausgeblendet.

Um dieses Verhalten zu demonstrieren, erweitern wir das zuvor erstellte Beispiel um einen Button mit dem Titel „Hide red cube“, der im Storyboard am unteren Rand des initialen View-Controllers platziert wird (siehe Bild 23.94). Bei Tippen auf diesen Button soll die

View des roten Würfels wieder vom View-Controller entfernt werden. Dazu erstellen wir im ersten Schritt eine Action-Methode für diesen Button im View-Controller, die den Namen `hideRedCube()` trägt.

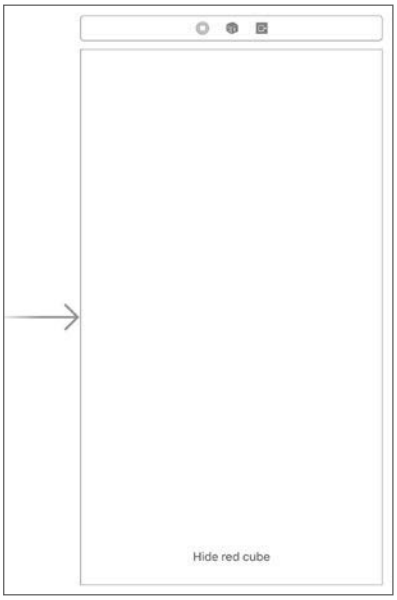


Bild 23.94
Das Beispielprojekt erhält einen Button am unteren Bildschirmrand, über den die im Code hinzugefügte View des roten Würfels wieder entfernt wird.

Um das gewünschte Verhalten nun im Code umzusetzen, muss zunächst die bestehende Implementierung der `ViewController`-Klasse ein wenig angepasst werden. Schließlich hat die neu erstellte Action-Methode `hideRedCube()` keine Referenz auf die View des roten Würfels, die sie ausblenden soll. Aus diesem Grund wird die View in Form einer Property im View-Controller erzeugt und `viewDidLoad()` nur noch mithilfe der Methode `addSubview(_:)` hinzugefügt. Dann kann innerhalb der Methode `hideRedCube()` auf diese Property die Methode `removeFromSuperview()` aufgerufen werden, um die View wieder zu entfernen. Die so aktualisierte und vollständige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.27.

Listing 23.27 Hinzufügen und Entfernen von Views

```
class ViewController: UIViewController {

    var redCube: UIView = {
        let redCubeFrame = CGRect(x: 50, y: 50, width: 200, height: 200)
        let redCube = UIView(frame: redCubeFrame)
        redCube.backgroundColor = .red
        return redCube
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(redCube)
    }
```

```
    @IBAction func hideRedCube() {
        redCube.removeFromSuperview()
    }

}
```

Führt man dieses aktualisierte Projekt nun aus, verschwindet die in `viewDidLoad()` hinzugefügte View, sobald der Button „Hide red cube“ betätigt wird.

Hinzufügen über Superview, Entfernen über Subview

Wie Sie am letzten Beispiel gesehen haben, verfolgen das Hinzufügen einer Subview und das Entfernen derselbigen zwei verschiedene Ansätze.

Neue Subviews werden immer über die Superview hinzugefügt, zu der sie gehören sollen (in dem gezeigten Beispiel war das die View des View-Controllers). Auf dieser Superview wird `addSubview(_:)` aufgerufen und ihr die neue Subview als Parameter übergeben.

Möchte man hingegen eine Subview wieder entfernen, verwendet man dazu nicht ebenfalls die Superview, sondern die Subview selbst, die verschwinden soll, und ruft direkt auf ihr die Methode `removeFromSuperview()` auf.

23.6.3 Grundlegende Eigenschaften aller Views

In den vorangegangenen Abschnitten haben wir eine Eigenschaft von Views bereits einige Male verwendet: die Hintergrundfarbe in Form der Property `backgroundColor`. Es gibt aber noch eine Vielzahl mehr davon, und in diesem Abschnitt möchte ich Ihnen einige der in meinen Augen wichtigsten und interessantesten vorstellen. All diese Eigenschaften sind innerhalb der Klasse `UIView` deklariert, was bedeutet, dass sie auch für alle anderen Arten von Views, wie sie zum Teil in Abschnitt 23.6.4, „Verfügbare UIView-Subklassen“, aufgeführt sind, zur Verfügung stehen. Eine Übersicht *aller* Eigenschaften und Funktionen der Klasse `UIView` finden Sie in der Dokumentation von Xcode (siehe Bild 23.95).

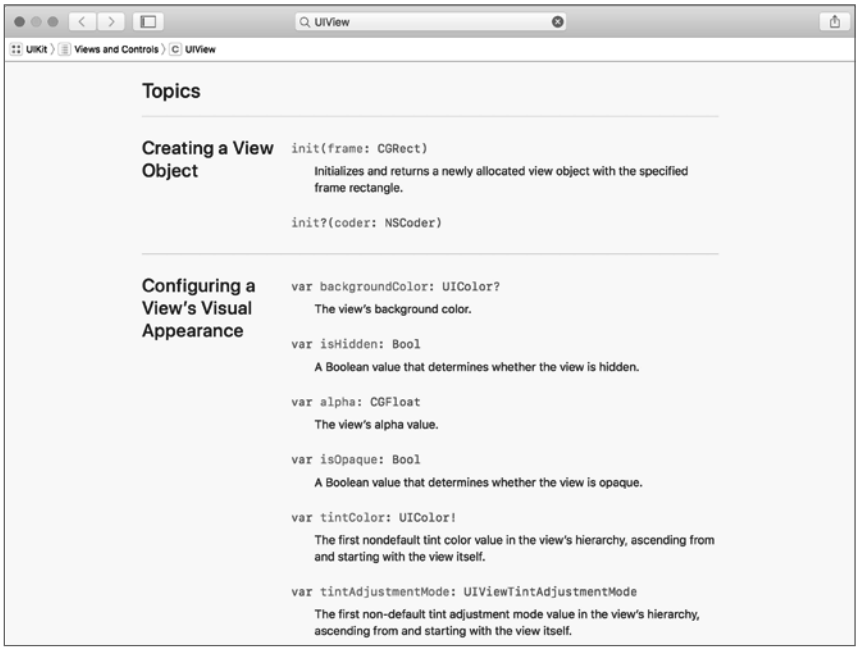


Bild 23.95 In der Xcode-Dokumentation zu UIView finden sie alle Eigenschaften und Funktionen, die diese Klasse zur Verfügung stellt.

23.6.3.1 Optische Anpassungen

Die im Folgenden vorgestellten Eigenschaften der Klasse `UIView` dienen allesamt dazu, das Aussehen einer View zu verändern.

backgroundColor

Mithilfe der Property `backgroundColor` bestimmen sie die Hintergrundfarbe einer View. Sie ist vom Typ `UIColor`. `UIColor` bringt für eine Vielzahl von Farben bereits passende Klassenvariablen mit, über die Sie schnell und einfach eine bestimmte Farbe im Code erzeugen können, beispielsweise `black`, `red` oder `blue`. Darüber hinaus können Sie mithilfe passender Methoden und Initializer auch ganz individuelle Farben erzeugen. Beispielsweise können Sie den Initializer `init(red:green:blue:alpha:)` dazu verwenden, um eine Farbe auf Basis von RGB-Werten zu erstellen. Pro RGB-Wert geben Sie eine Zahl zwischen 0 (keine Verwendung der Farbe) und 1 (volle Verwendung der Farbe) an. Der letzte Parameter – `alpha` – bestimmt die Transparenz der Farbe (0 bedeutet volle Transparenz, 1 keine Transparenz).

Listing 23.28 zeigt einige Beispiele zur Verwendung der Klasse `UIColor`. Alle darin erzeugten Instanzen können direkt einer View als Hintergrundfarbe mithilfe der Property `backgroundColor` zugewiesen werden.

Listing 23.28 Erstellen von Farben mithilfe von `UIColor`

```
// Rot
let redColor = UIColor.red

// Grün
let greenColor = UIColor.green

// Blau
let blueColor = UIColor.blue

// Weiß
let whiteColor = UIColor.white

// Schwarz
let blackColor = UIColor.black

// Braun
let brownColor = UIColor.brown

// Orange
let orangeColor = UIColor.orange

// Lila
let purpleColor = UIColor.purple

// Hellblau
let lightBlueColor = UIColor(red: 0.8, green: 0.9, blue: 1, alpha: 1)
```

Ein Beispiel zur praktischen Veränderung der Hintergrundfarbe finden Sie in Listing 23.27 in Abschnitt 23.6.2, „Erstellen von Views“.

isHidden

Mithilfe der booleschen Property `isHidden` regeln Sie, ob eine View sichtbar ist (`true`) oder nicht (`false`). Im Gegensatz zu der Methode `removeFromSuperview()`, die eine View vollständig aus einer Ansicht entfernt, ändert `isHidden` nur die Sichtbarkeit einer View. Sie können diese Property nutzen, um bestimmte Elemente nur unter gewissen Voraussetzungen anzuzeigen.

Ein kleines Beispiel zur Verwendung der Property `isHidden` zeigt das folgende Projekt. Es basiert auf einer *Single View App*, der initiale View-Controller verfügt über einen Schalter und ein Label (siehe Bild 23.96). Das Label ist mithilfe eines Outlets mit dem Code der zugrunde liegenden `ViewController`-Klasse verknüpft, der Switch ist einer Action-Methode namens `updateLabelVisibility(_:)` zugewiesen.

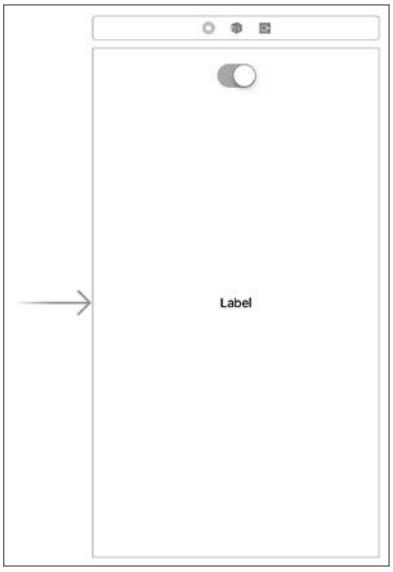


Bild 23.96
Wann immer der Switch des Beispielprojekts inaktiv ist, soll das Label ausgeblendet werden.

Innerhalb der Action-Methode wird der Status des Schalters mithilfe der zugehörigen Property `isOn` überprüft. Ist er inaktiv, wird das Label ausgeblendet, indem die Property `isHidden` des Labels auf `false` gesetzt wird. Umgekehrt wird `isHidden` der Wert `true` zugewiesen, sollte der Schalter wieder aktiviert werden. Die zugehörige Implementierung der `ViewController`-Klasse finden Sie in Listing 23.29.

Listing 23.29 Ein- und Ausblenden eines Labels auf Basis eines Switches

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func updateLabelVisibility(_ sender: UISwitch) {
        label.isHidden = !sender.isOn
    }

}
```

Mehr zur Klasse `UISwitch` und weiteren View-Elementen erfahren Sie in Abschnitt 23.6.4, „Verfügbare UIView-Subklassen“.

alpha

Mithilfe der `alpha`-Property steuern Sie die Transparenz einer View. Ein Wert von 1 bedeutet volle Sichtbarkeit (dies ist der Standard), ein Wert von 0 bedeutet volle Transparenz.

Das folgende Beispiel verdeutlicht die Funktionsweise der `alpha`-Property. Es basiert auf einer *Single View App*, der initiale View-Controller verfügt über einen Slider (Klasse `UISlider`) und ein Label (siehe Bild 23.97). Der Slider verfügt über einen Wertebereich von 0 bis 1 und soll den Alpha-Wert des Labels darstellen. Wird der Wert des Sliders geändert, soll entsprechend auch der Alpha-Wert des Labels geändert werden.

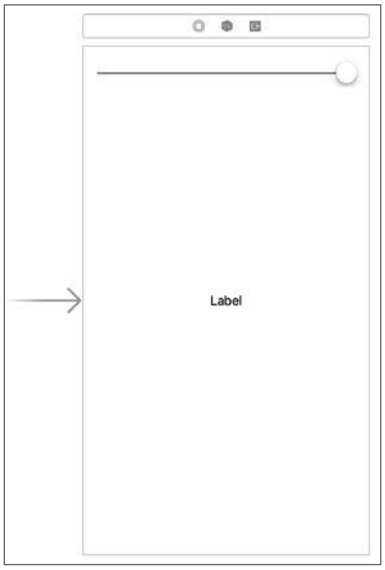


Bild 23.97
In der Beispiel-App steuert ein Slider den Alpha-Wert des Labels.

Zu diesem Zweck wird das Label mithilfe eines Outlets mit dem Code verbunden. Der Slider wird mit einer Action-Methode namens `updateLabelTransparency(_:)` mit der zugrunde liegenden `ViewController`-Klasse verknüpft. Diese Action-Methode wird immer dann aufgerufen, wenn sich der Wert des Sliders ändert. Über die Property `value` der Klasse `UISlider` kann der aktuelle Wert ausgelesen werden. In der Implementierung von `updateLabelTransparency(_:)` wird er direkt der `alpha`-Property des Labels zugewiesen. Die entsprechende Implementierung der `ViewController`-Klasse finden Sie in Listing 23.30.

Listing 23.30 Verändern des Alpha-Werts eines Labels auf Basis eines Sliders

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func updateLabelTransparency(_ sender: UISlider) {
        label.alpha = CGFloat(sender.value)
    }

}
```

Je nachdem, an welcher Position sich der Slider befindet (und welchem Wert er somit entspricht), verändert sich auch der Alpha-Wert des Labels (siehe Bild 23.98).

Bild 23.98 Mithilfe des Sliders wird der Alpha-Wert des Labels verändert.

CGFloat

Der Typ der `alpha`-Property entspricht `CGFloat`. Es handelt sich um eine Structure aus dem *Core Graphics*-Framework, die zur Abbildung von Fließkommazahlen verwendet wird. Bei der Programmierung mit Swift wird `CGFloat` intern automatisch als `Float` genutzt, sollte eine App auf einem 32-Bit-Prozessor ausgeführt werden. Bei Geräten mit 64-Bit-Prozessor entspricht eine `CGFloat`-Instanz hingegen automatisch einem `Double`.

In vielen Fällen ist dieser Umstand für uns nicht weiter relevant. Einer `CGFloat`-Property kann problemlos ein Wert wie `19.99` zugewiesen werden; dieser wird dann automatisch in ein `CGFloat` umgewandelt.

Anders liegt der Fall allerdings im gezeigten Beispiel aus Listing 23.30. Die `value`-Property der Klasse `UISlider` liefert eine Instanz vom Typ `Float` zurück, und diese kann nicht einfach einem `CGFloat` zugewiesen werden. Hier ist ein explizites Type Casting von `Float` nach `CGFloat` notwendig.

Ein solches Casting ist erfreulicherweise sehr einfach: Die Structure `CGFloat` bringt für verschiedenste Zahlen-Typen wie `Int`, `Float` und `Double` einen passenden Initializer mit, der den übergebenen Wert in einen `CGFloat` umwandelt. Genau das ist auch in Listing 23.30 geschehen. ■

23.6.3.2 Verändern der Größe und Position

Was die Größe und die Position einer View betrifft, so wird in der iOS-Entwicklung zwischen *Frame* und *Bounds* unterschieden. Beide sind vom Typ `CGRect` und fassen somit die X- und Y-Koordinaten einer View sowie deren Höhe und Breite zusammen. Doch worin liegt der Unterschied zwischen den beiden?

Der *Frame* einer View bezieht sich auf die Position und Größe der View im Verhältnis zu ihrer *Superview*. Die *Bounds* hingegen beziehen sich auf die Position und Größe einer View innerhalb ihres *eigenen Koordinatensystems*.

Am verständlichsten erläutern lässt sich der Unterschied zwischen Frame und Bounds anhand eines Beispiels. Betrachten Sie dazu einmal die Grafik in Bild 23.99. Sie stellt eine Superview dar, der ein blaues Rechteck als Subview hinzugefügt wurde.

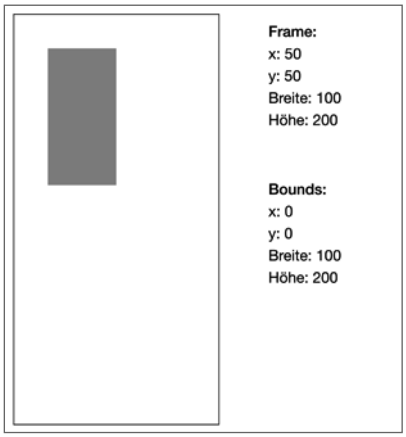


Bild 23.99
Frame und Bounds des Rechtecks innerhalb der Superview unterscheiden sich voneinander.

Der Frame bezieht sich auf die Position des blauen Rechtecks aus Sicht der *Superview*, in der sie eingebettet ist. Daher unterscheiden sich die Werte für die X- und Y-Koordinaten im Vergleich zu Bounds, die sich auf das Koordinatensystem *innerhalb* des blauen Rechtecks beziehen.

Noch deutlicher wird der Unterschied, wenn die View des blauen Rechtecks *rotiert* wird, so wie in Bild 23.100 zu sehen. Auch in diesem Fall bleiben die Bounds unverändert, da sich *innerhalb* des Rechtecks erneut nichts geändert hat. Der Frame hingegen, der die View aus Sicht der Superview betrachtet, hat sich sehr wohl geändert, was mithilfe der roten Linie symbolisiert wird. Der Frame beschreibt die Fläche, in der sich das blaue Rechteck befindet, und durch die Rotation nahmen sowohl Höhe als auch Breite in dieser Hinsicht zu.

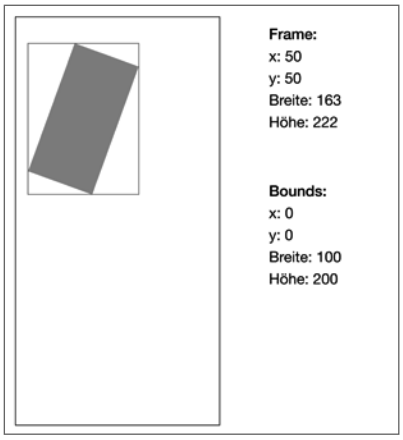


Bild 23.100
Durch die Rotation des blauen Rechtecks haben sich Breite und Höhe des Frames stark verändert (symbolisiert durch den roten Kasten).

Frame und Bounds sind beide über entsprechende Properties der Klasse `UIView` abrufbar und veränderbar: `frame` liefert den Frame, `bounds` liefert die Bounds. Ihr jeweils zugrunde liegender Typ ist `CGRect` und stammt – genau wie `CGFloat` – aus dem *Core Graphics*-Framework von Apple. Eine `CGRect`-Instanz besteht aus zwei Bestandteilen:

- `origin`: Bei `origin` handelt es sich um eine Instanz vom Typ `CGPoint`. Sie dient dazu, Koordinaten abzubilden. In Verbindung mit Frame und Bounds einer View wird sie verwendet, um die X- und Y-Koordinate der View zu definieren.
- `size`: Die Property `size` spiegelt die Größe einer View auf Basis von Höhe und Breite wider. Sie wird mithilfe des Typs `CGSize` abgebildet.

In Listing 23.31 sehen Sie ein Beispiel zur Erstellung einer `CGRect`-Instanz auf Basis eines `CGPoint` und einer `CGSize`.

Listing 23.31 Erstellen einer `CGRect`-Instanz

```
let point = CGPoint(x: 50, y: 50)
let size = CGSize(width: 100, height: 200)
let frame = CGRect(origin: point, size: size)
```

Frame vs. Bounds: Wann nimmt man was?

Sie haben in diesem Abschnitt erfahren, was Frame und Bounds sind und dass sie dem Typ `CGRect` entsprechen. Über die Properties `frame` und `bounds` einer `UIView` können Sie darauf zugreifen und entweder ihren aktuellen Wert auslesen oder einen neuen Wert setzen.

Doch wann sollte man Frame und wann Bounds einsetzen?

Die Antwort auf diese Frage hängt letzten Endes immer vom zugrunde liegenden Kontext ab. Wenn Sie einer Superview eine Subview hinzufügen möchten, konfigurieren Sie in der Regel den *Frame* dieser Subview. Der Grund hierfür ist, dass die Subview sich in Sachen Position und Größe in diesem Szenario an ihrer Superview orientieren soll; genau das leistet der Frame.

Bounds hingegen sollten Sie verwenden, wenn Sie die zugrunde liegende View selbst verändern (zum Beispiel deren Größe anpassen) oder sich darin orientieren möchten, um eine passende Position für mögliche Subviews zu ermitteln. ■

23.6.4 Verfügbare UIView-Subklassen

Die Klasse `UIView` stellt *die Mutter aller Views* in der iOS-Entwicklung dar. Sie enthält alle essenziellen Eigenschaften und Funktionen, die jede View ausmachen und verfügt über die Techniken, um Views zu verschachteln und mithilfe von Sub- und Superviews komplexe Ansichten umzusetzen.

Innerhalb des *UIKit*-Frameworks finden sich neben der Klasse `UIView` aber eine Vielzahl weiterer Klassen, die von `UIView` abgeleitet sind. Diese `UIView`-Subklassen stellen essenzielle Elemente dar, mit denen wir unseren Apps Leben einhauchen und unser User Interface gestalten können.

Einige dieser Subklassen haben wir bereits in den vorangegangenen Abschnitten dieses Kapitels kennengelernt: Labels, Buttons, Switches und Slider kamen bereits zum Einsatz. All diese Elemente sind Subklassen von `UIView` und verfügen daher über all die Eigenschaften und Funktionen, die zuvor vorgestellt wurden.

In diesem Abschnitt möchte ich Ihnen eine Auswahl verschiedener Elemente vorstellen, mit denen Sie Ihr User Interface in der iOS-Entwicklung gestalten können. Dabei gehe ich sowohl auf die jeweilige Funktionsweise der View ein als auch darauf, wie Sie sie in eigenen Projekten einsetzen und welche Eigenschaften sie besitzen.

Betrachtung „einfacher" View-Elemente

Dieser Abschnitt widmet sich den „einfacheren" `UIView`-Subklassen. „Einfacher" meint hierbei, dass die Funktionsweise dieser Elemente auf genau eine Aufgabe beschränkt ist und keine komplexen Konfigurationen möglich sind. Solch komplexere Views wie beispielsweise Tabellen oder Picker werden ausführlich in Kapitel 24, „iOS – App-Entwicklung" behandelt.

23.6.4.1 Label

Ein Label dient zur Darstellung eines einfachen Texts (siehe Bild 23.101). Es wird im Code mithilfe der Klasse `UILabel` abgebildet. Innerhalb des Interface Builders stehen Ihnen unter anderem die folgenden Konfigurationsmöglichkeiten zur Verfügung (siehe Bild 23.102):

- *Text:* Der Text, den das Label anzeigt.
- *Color:* Die Farbe des Labels.
- *Font:* Die Schriftart und -größe des Labels.
- *Alignment:* Die Ausrichtung (linksbündig, zentriert, rechtsbündig) des Labels.
- *Lines:* Die Anzahl der Zeilen, die das Label maximal besitzen kann. Ist der verwendete Text zu lang für das Label, wird er mittels dreier Punkte am Ende („…") abgeschnitten. Wenn Sie hier als Wert 0 einsetzen, erhält das Label so viele Zeilen wie es braucht, um den gegebenen Text darzustellen.

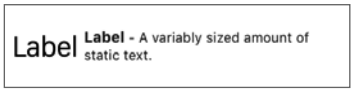


Bild 23.101
Mithilfe von Labels stellen Sie einfache Texte in iOS-Apps dar.

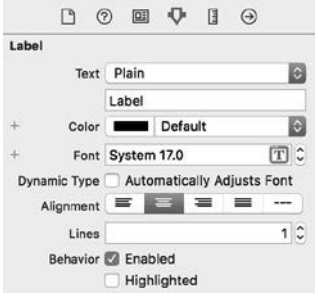


Bild 23.102
Innerhalb des Interface Builders können Sie verschiedene Einstellungen eines Labels wie den anzuzeigenden Text oder die Farbe verändern.

Konfiguration im Code

Zur Anpassung oder Erstellung eines Labels im Code stellt Ihnen die Klasse `UILabel` ein Set an passenden Eigenschaften und Funktionen zur Verfügung.

So ändern Sie beispielsweise mithilfe der Property `text` den anzuzeigenden Text des Labels, mit `textColor` die Textfarbe.

Schriftart und -größe werden mithilfe des Typs `UIFont` innerhalb der Property `font` abgebildet. Die Ausrichtung eines Labels (linksbündig, zentriert etc.) ist in der Property `textAlignment` vom Typ `NSTextAlignment` definiert. Es handelt sich bei diesem Typ um eine Enumeration, die passende Werte wie `left`, `right` und `center` zur Konfiguration zur Verfügung stellt.

Die Anzahl der Zeilen eines Labels wird über die Property `numberOfLines` abgebildet.

23.6.4.2 Button

Mithilfe eines Buttons (siehe Bild 23.103) können Aktionen von Ihrem User Interface im Code ausgelöst werden. Dazu wird ein Button mit einer passenden Action-Methode im zugrunde liegenden View-Controller gekoppelt. Im Code werden Buttons mithilfe der Klasse `UIButton` abgebildet. Über den Interface Builder können Sie Eigenschaften wie den anzuzeigenden Text, die Textfarbe oder alternativ eine Grafik für einen Button festlegen (siehe Bild 23.104).

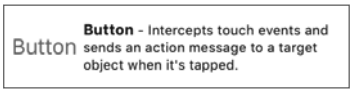


Bild 23.103
Mithilfe von Buttons können Sie eine einfache Aktion in Ihrer App auslösen.

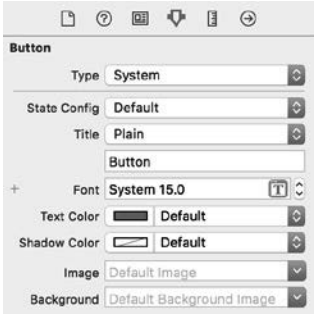


Bild 23.104
Sie können über den Interface Builder unter anderem die Farbe und den Text sowie eine optionale Grafik für einen Button festlegen.

Konfiguration im Code

Wenn Sie eine neue `UIButton`-Instanz im Code erzeugen, sollten Sie dazu den spezialisierten Initializer `init(type:)` einsetzen. Er erwartet als Parameter eine Instanz vom Typ `UIButtonType`, die den grundlegenden Style einer Schaltfläche definiert. Diese Enumeration verfügt über die folgenden Werte:

- `custom`: Es wird kein spezifischer Style festgelegt und Sie kümmern sich vollständig um die passende Konfiguration des Buttons.
- `system`: Hierbei handelt es sich um den Standard für Schaltflächen in iOS. Auf diese Art und Weise sind beispielsweise Buttons konfiguriert, die innerhalb von Navigation Bars

oder Toolbars platziert sind. Die Textfarbe wird hier automatisch angepasst, um einen Button nicht möglicherweise mit einem statischen Label ohne Aktion zu verwechseln.

- `infoLight`: Der Button wird so konfiguriert, dass er ein kleines „i“, umgeben von einem runden Kreis, anzeigt. Diese Konfiguration dient dazu, komfortabel einen Informations-Button zu erstellen (siehe Bild 23.105).
- `contactAdd`: Bei diesem Style wird der Button als Plus-Schaltfläche konfiguriert, um zu signalisieren, dass darüber ein Element (egal welcher Art) hinzugefügt werden kann (siehe Bild 23.105).

Bild 23.105
Die Styles infoLight und contactAdd verändern umgehend das optische Erscheinungsbild eines Buttons und sind ausschließlich für spezifische Einsatzzwecke interessant.

Listing 23.32 zeigt beispielsweise die Erstellung einer neuen `UIButton`-Instanz im Code auf Basis des `infoLight`-Styles. Möchten Sie einen „frischen“ `UIButton` ohne jedwede vorausgehende optische Konfiguration erstellen, müssen Sie den Style `custom` einsetzen.

Listing 23.32 Erstellen einer `UIButton`-Instanz auf Basis eines Styles

```
let infoButton = UIButton(type: .infoLight)
```

Was Größe und Position eines im Code erzeugten Buttons betrifft, müssen Sie diese nach der Erstellung der Instanz mithilfe der `frame`-Property festlegen, indem Sie dieser einen passenden Wert zuweisen.

Was die sonstige Konfiguration eines Buttons im Code betrifft, so kommen hierfür primär verschiedene Methoden zum Einsatz. Den Titel eines Buttons beispielsweise ändern Sie mithilfe der Methode `setTitle(_:for:)`. Neben dem neuen Titel erwartet diese Methode auch eine Information darüber, für welchen *Zustand* (*State*) dieser Titel gilt. Hintergrund hierbei ist, dass Sie für unterschiedliche Zustände eines Buttons auch verschiedene Titel vergeben können.

Die Zustände werden in Form der Structure `UIControlState` abgebildet. Sie verfügt unter anderem über die folgenden Werte:

- `normal`: Der „Normalzustand“ eines Buttons. Verwenden Sie diesen State auch immer dann, wenn Sie nur einen Titel (unabhängig vom Zustand) für einen Button setzen möchten. Wenn Sie dann für keinen der anderen Zustände einen separaten Titel übergeben, wird immer der von Ihnen für den `normal`-Zustand gesetzte Titel für den Button angezeigt.
- `highlighted`: Dieser Zustand wird erreicht, wenn der Button ausgewählt wird, sprich in iOS mit dem Finger berührt wird.
- `disabled`: Sollte der Button deaktiviert sein, wird der Titel angezeigt, den Sie für diesen Zustand gesetzt haben. Ein Button wird vom System dann als inaktiv angesehen, wenn dessen `isEnabled`-Property `false` entspricht.

Wie Sie so beispielsweise den Titel eines Buttons für die beschriebenen Zustände verändern, sehen Sie in Listing 23.33.

Listing 23.33 Ändern des Titels eines Buttons für verschiedene Zustände

```
let myButton = UIButton(type: .custom)
myButton.setTitle("Normal", for: .normal)
myButton.setTitle("Highlighted", for: .highlighted)
myButton.setTitle("Disabled", for: .disabled)
```

Neben dem Titel können Sie mithilfe der Methode `setTitleColor(_:for:)` auch die Textfarbe eines Buttons verändern. Ähnlich verhält es sich bei einer (Hintergrund-)Grafik; hierfür stehen Ihnen die Methoden `setImage(_:for:)` und `setBackgroundImage(_:for:)` zur Verfügung (mehr über den Umgang mit Grafiken erfahren Sie in Abschnitt 23.6.4.6, „Image View“).

23.6.4.3 Segmented Control

Ein Segmented Control basiert auf der Klasse `UISegmentedControl` und ist in seiner Funktionsweise dem eines Buttons sehr ähnlich (siehe Bild 23.106). Es verfügt über mehrere sogenannte *Segmente* (mindestens zwei), zwischen denen der Nutzer auswählen kann. Genau wie bei einem Button löst eine solche Auswahl eines Segments eine eigens definierte Action-Methode im Code des zugrunde liegenden View-Controllers aus. Innerhalb dieser Action-Methode kann man dann überprüfen, welches Segment ausgewählt wurde, und entsprechende Befehle ausführen. Mehr als ein Segment kann nie in einem Segmented Control aktiv sein.

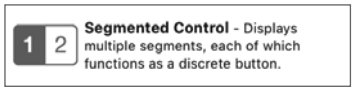


Bild 23.106 Ein Segmented Control stellt eine Leiste mit zwei oder mehr Schaltflächen dar, zwischen denen der Nutzer wählen kann. Es ist aber immer nur maximal eine Schaltfläche aktiv.

Es gibt verschiedene Einsatzzwecke für ein solches Segmented Control. Ein Beispiel wäre eine Filterfunktion von Aufgaben. Ein Segmented Control kann hierzu Buttons wie „Titel“, „Deadline“ oder „Erstellungsdatum“ anbieten, die eine Liste von Aufgaben entsprechend sortieren. Ein anderes Beispiel ist die Auswahl einer Priorität für eine Aufgabe, für die ein Segmented Control eine Auswahl zwischen „Normal“, „Wichtig“ und „Dringend“ anbieten kann.

Innerhalb des Interface Builders haben Sie verschiedene Konfigurationsmöglichkeiten für ein Segmented Control (siehe Bild 23.107). Die folgende Auflistung stellt Ihnen einige davon im Detail vor:

- *Segments:* Hier geben Sie die Anzahl der Segmente ein, über die das Segmented Control verfügen soll (das Minimum ist zwei).
- *Segment:* In diesem Dropdown-Menü können Sie auf jedes Segment des Segmented Controls zugreifen. Wie viele Elemente hier angezeigt werden, hängt davon ab, welche Anzahl an Segmenten Sie unter *Segments* definiert haben.

  Die Auswahl dient zur Konfiguration des jeweiligen Elements mithilfe der unter *Segment* aufgeführten Optionen. Um also ein Segment zu konfigurieren, wählen Sie es an dieser Stelle aus und passen anschließend die darunter aufgeführten Optionen an.

- *Title:* Hier legen Sie den Titel für das unter *Segment* ausgewählte Segment fest.
- *Image:* Hier können Sie optional ein Hintergrundbild für das unter *Segment* ausgewählte Segment setzen.
- *Selected:* Ist diese Checkbox aktiv, ist das unter *Segment* ausgewählte Segment standardmäßig aktiv und ausgewählt. Es kann immer nur ein Segment eines Segmented Controls aktiv sein. Wenn Sie diese Checkbox bei einem Segment aktivieren, wird eine mögliche Aktivierung eines anderen Segments umgehend aufgehoben.

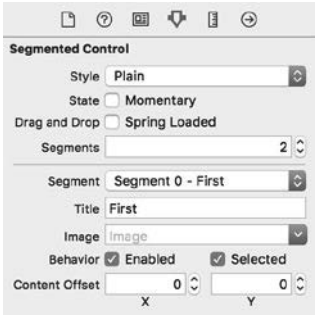


Bild 23.107
Mithilfe des Interface Builders können Sie die Segmente eines Segmented Controls konfigurieren.

Konfiguration im Code

Wenn Sie ein `UISegmentedControl` im Code erzeugen, kommt hierfür der Initializer `init(items:)` zum Einsatz. Er erwartet ein Array, in dem Sie entweder den Titel für die gewünschten Segmente des Segmented Controls oder die Hintergrundbilder (in Form von `UIImage`-Instanzen) übergeben (mehr zum Umgang mit Bildern erfahren Sie in Abschnitt 23.6.4.6, „Image View“). Ein Element dieses Arrays entspricht hierbei einem Segment des zu erstellenden Segmented Controls. Listing 23.34 zeigt hierzu ein Beispiel, in dem ein neues Segmented Control mit drei Segmenten erzeugt wird.

Listing 23.34 Erstellen eines Segmented Controls im Code

```
let mySegmentedControl = UISegmentedControl(items: ["First", "Second", "Third"])
```

Um die Hintergrundgrafik oder den Titel eines Segments zu ändern, stehen Ihnen die Methoden `setImage(_:forSegmentAt:)` und `setTitle(_:forSegmentAt:)` zur Verfügung. Neben dem jeweiligen Element (Grafik oder Titel) erwarten die Methoden zusätzlich noch den Index des Segments, das Sie ändern möchten. Das erste Segment beginnt mit dem Index 0, das zweite besitzt entsprechend den Index 1, das dritte 2 und so weiter.

Sie können einem Segment auch neue Segmente mithilfe der Methoden `insertSegment(withTitle:at:animated:)` (für Segmente auf Basis eines Titels) und `insertSegment(with:at:animated:)` (für Segmente auf Basis einer Grafik) hinzufügen. Neben dem jeweiligen Element (Titel oder Grafik) übergeben Sie noch die gewünschte Index-Position, an der das Segment im bestehenden Segmented Control eingefügt werden soll, und definieren im letzten Parameter, ob das Einfügen animiert erfolgen soll (`true`) oder nicht (`false`).

Sie können auch jederzeit Segmente aus einem Segmented Control wieder entfernen. Nutzen Sie hierfür entweder die Methode `removeSegment(at:animated:)` (um ein einzelnes Segment zu entfernen) oder `removeAllSegments()` (um alle Segmente zu entfernen). In

letzterem Fall sollten Sie anschließend passende neue Segmente erstellen oder das Segmented Control ausblenden beziehungsweise entfernen (da es ja nicht länger benötigt wird, wenn es keine Segmente mehr besitzt). Beim Entfernen eines einzelnen Segments geben Sie den Index des Segments an, das Sie entfernen möchten, und eine Information, ob das Entfernen animiert erfolgen soll (`true`) oder nicht (`false`).

Die praktische Verwendung der Methoden zum Hinzufügen neuer und Entfernen bestehender Segmente sehen Sie beispielhaft in Listing 23.35.

Listing 23.35 Hinzufügen neuer und Entfernen bestehender Elemente in einem Segmented Control

```
let mySegmentedControl = UISegmentedControl(items: ["First", "Second", "Third"])
mySegmentedControl.insertSegment(withTitle: "Fourth", at: 3, animated: false)
mySegmentedControl.removeSegment(at: 0, animated: true)
```

Abschließend möchte ich Ihnen noch die sehr wichtige Property `selectedSegmentIndex` vorstellen. Sie liefert den Index des Segments zurück, das aktuell ausgewählt ist. Darüber hinaus können Sie die Property dazu verwenden, ein anderes Segment als aktiv auszuwählen, indem Sie ihr den Index des zugehörigen Segments zuweisen.

Diese Property ist beispielsweise wichtig, um bei der einem Segmented Control zugeordneten Action-Methode zu ermitteln, welches Segment gerade aktiv ist und somit ausgewählt wurde. Praktisch demonstriert wird dieses Prinzip in Listing 23.36. Darin finden Sie die Implementierung einer `ViewController`-Klasse, die über ein Label und ein Segmented Control verfügt. Das Label ist mithilfe eines Outlets mit dem View-Controller verknüpft. Dem Segmented Control ist eine Action-Methode namens `changeSegment(_:)` zugewiesen, die als `sender`-Parameter die Referenz auf das Segmented Control liefert.

Wann immer nun ein Segment innerhalb des Segmented Controls ausgewählt wird, wird dem Label der Text dieses Segmented Controls zugewiesen (siehe Bild 23.108). Zu diesem Zweck kommt die Methode `titleForSegment(at:)` zum Einsatz, die den Titel für das Segment des übergebenen Index zurückliefert (oder `nil`, falls das Segment nicht existiert beziehungsweise keinen Titel besitzt).

Listing 23.36 Reaktion auf die Auswahl eines Segments innerhalb eines Segmented Controls

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func changeSegment(_ sender: UISegmentedControl) {
        let selectedSegment = sender.selectedSegmentIndex
        let selectedSegmentTitle = sender.titleForSegment(at: selectedSegment)!
        label.text = selectedSegmentTitle
    }

}
```

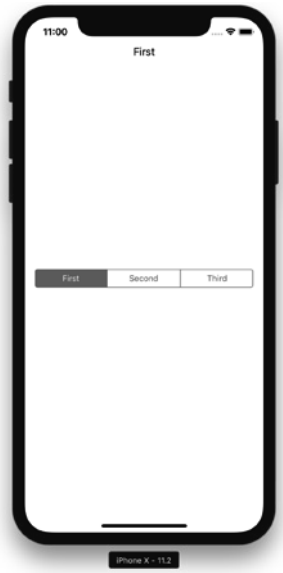


Bild 23.108
Der Titel des Labels entspricht dem des ausgewählten Segments des Segmented Controls.

23.6.4.4 Switch

Switches basieren auf der Klasse `UISwitch` und kennen nur zwei Zustände: an oder aus (siehe Bild 23.109). Sie eignen sich ideal für bestimmte Einstellungen, bei denen der Nutzer lediglich entscheiden kann, ob er ein bestimmtes Feature (beispielsweise den Erhalt von Notifications oder die Verwendung von iCloud) nutzen möchte oder nicht. Um über Änderungen beim Status eines Switches informiert zu werden, verbinden Sie diesen mit einer passenden Action-Methode im zugrunde liegenden View-Controller.

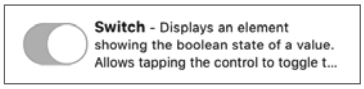


Bild 23.109 Mithilfe eines Switches können Sie Schalter in Ihre iOS-Apps einbauen, die nur zwei Zustände kennen: an oder aus.

Zur Konfiguration eines Switches stehen Ihnen im Interface Builder die folgenden Optionen zur Verfügung (siehe Bild 23.110):

- *State:* Der Ausgangszustand des Switches (*On* oder *Off*).
- *On Tint:* Die Farbe des Switches, wenn er aktiv ist. Standardmäßig wird hierfür ein in Bild 23.110 gezeigtes Grün verwendet, das Sie über diese Option ändern können.
- *Thumb Tint:* Hierüber legen Sie die Farbe für den Knopf des Schalters fest.

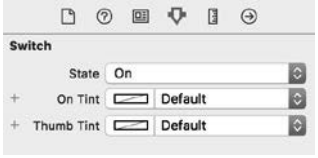


Bild 23.110
Im Interface Builder können Sie den Status und das optische Aussehen eines Switches verändern.

Konfiguration im Code

Die wichtigste Property einer `UISwitch`-Instanz ist `isOn`. Sie liefert `true` zurück, wenn der Schalter aktiv ist, und andernfalls `false`. Sie kann auch dazu verwendet werden, den Zustand des Schalters programmatisch zu ändern, indem man ihr den gewünschten Wert zuweist.

Alternativ zum Ändern des Status eines Switches können Sie auch die Methode `setOn(_:animated:)` aufrufen, die einen zusätzlichen `animated`-Parameter erwartet. Ist dieser `true`, wird das Ändern des Status animiert dargestellt, andernfalls nicht. Das direkte Ändern der `isOn`-Property entspricht übrigens dem Aufruf der Methode `setOn(_:animated:)` mit deaktivierter Animation.

In Listing 23.37 finden Sie ein Beispiel zur Verwendung der `isOn`-Property sowie der `setOn(_:animated:)`-Methode.

Listing 23.37 Programmatisches Ändern des Status eines Switches

```
// mySwitch verweist auf eine UISwitch-Instanz.
mySwitch.isOn = false
mySwitch.setOn(true, animated: true)
```

Um darüber informiert zu werden, wenn ein Nutzer einen Switch betätigt und somit den Status von *On* zu *Off* (und umgekehrt) ändert, müssen Sie den Switch im zugrunde liegenden View-Controller mit einer Action-Methode verbinden. Ein Beispiel hierfür finden Sie in Listing 23.38. Darin wird eine View-Controller-Klasse implementiert, die über ein Label und einen Switch verfügt. Wann immer der Schalter betätigt wird, wird eine Action-Methode namens `switchChanged(_:)` ausgelöst, die die zugrunde liegende `UISwitch`-Instanz als Referenz übergibt. Diese wird dazu verwendet, den Status des Schalters mithilfe der `isOn`-Property zu prüfen und dem Label einen entsprechend passenden Text zuzuweisen.

Listing 23.38 Reaktion auf Betätigung eines Switches

```
class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBAction func switchChanged(_ sender: UISwitch) {
        if sender.isOn {
            label.text = "Is on"
        } else {
            label.text = "Is off"
        }
    }

}
```

23.6.4.5 Activity Indicator View

Eine Activity Indicator View stellt eine animierte Ladeansicht dar, die sich ideal in Situationen einblenden lässt, in denen länger andauernde Aktionen ausgeführt oder beispielsweise neue Inhalte einer News-App aus dem Internet geladen werden (siehe Bild 23.111). Die View basiert auf der Klasse `UIActivityIndicatorView`.

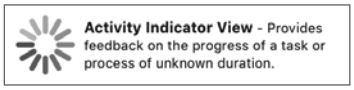


Bild 23.111
Eine Activity Indicator View stellt eine animierte Ladeansicht dar.

Im Interface Builder können Sie die folgenden Konfigurationen für eine Activity Indicator View vornehmen (siehe Bild 23.112).

- *Style:* Hierüber bestimmen Sie das Aussehen der Activity Indicator View. Sie können zwischen verschiedenen vorgegebenen Farben und Größen wählen. Unabhängig von der hier getroffenen Auswahl können Sie die Farbe einer Activity Indicator View mithilfe des folgenden Punkts *Color* beliebig anpassen.
- *Color:* Die Farbe, die Sie für die Activity Indicator View verwenden möchten.
- *Animating:* Ist diese Checkbox aktiviert, wird die Activity Indicator View animiert dargestellt und simuliert eine Art Drehbewegung. Andernfalls wird die View statisch und ohne Bewegung angezeigt.
- *Hides When Stopped:* Ist diese Checkbox aktiv, wird die Activity Indicator View nur dann eingeblendet und angezeigt, wenn sie eine Animation ausführt (siehe den vorherigen Punkt *Animating*).

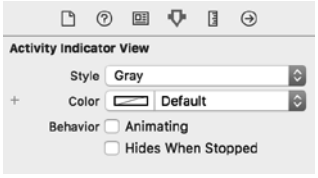


Bild 23.112
Im Interface Builder legen Sie das Aussehen und das Verhalten einer Activity Indicator View fest.

Konfiguration im Code

Im Code erstellen Sie neue Instanzen der Klasse `UIActivityIndicatorView` mithilfe des Initializers `init(activityIndicatorStyle:)`. Als Parameter erwartet dieser eine Instanz vom Typ `UIActivityIndicatorViewStyle`, bei dem es sich um eine Enumeration handelt, die dieselben Styles für eine Activity Indicator View abbildet, die auch im Interface Builder zur Auswahl stehen:

- `whiteLarge`: Eine große Activity Indicator View in weißer Farbe.
- `white`: Eine normalgroße Activity Indicator View in weißer Farbe.
- `gray`: Eine normalgroße Activity Indicator View in grauer Farbe.

Listing 23.39 zeigt beispielhaft die Erstellung einer `UIActivityIndicatorView`-Instanz auf Basis des `whiteLarge`-Styles.

Listing 23.39 Erzeugen einer `UIActivityIndicatorView`-Instanz im Code

```
let myActivityIndicatorView = UIActivityIndicatorView(activityIndicatorStyle:
.whiteLarge)
```

Mithilfe der Methoden `startAnimating()` beziehungsweise `stopAnimating()` können Sie den Animationsvorgang einer Activity Indicator View starten beziehungsweise stoppen. Der boolesche Parameter `isAnimating` liefert `true` zurück, wenn eine Activity Indicator View gerade animiert wird, andernfalls `false`. Und die Property `hidesWhenStopped` ist das

Gegenstück zur Checkbox *Hides When Stopped* aus dem Interface Builder; entspricht sie `true`, wird die Activity Indicator View nur angezeigt, wenn sie auch gerade animiert wird.

Die Farbe einer Activity Indicator View können Sie jederzeit mithilfe der `color`-Property auslesen und verändern.

Listing 23.40 zeigt ein paar Beispiele zur Verwendung der vorgestellten Properties und Methoden.

Listing 23.40 Beispielhafte Arbeit mit einer `UIActivityIndicatorView`-Instanz

```
// myActivityIndicatorView entspricht einer UIActivityIndicatorView-Instanz.
myActivityIndicatorView.color = .blue
myActivityIndicatorView.hidesWhenStopped = true
if myActivityIndicatorView.isAnimating {
    myActivityIndicatorView.stopAnimating()
}
```

23.6.4.6 Image View

Mithilfe von Image Views können Sie Grafiken und Bilder in iOS-App anzeigen (siehe Bild 23.113). Die Klasse, die zu diesem Zweck zum Einsatz kommt, lautet `UIImageView`. Sie kennt zwei Zustände: *Default* und *Highlighted*. Im Code können Sie zwischen den beiden wechseln und pro Zustand eine andere Grafik festlegen, die die Image View anzeigen soll. Sie können somit einer Image View mehrere Bilder zuweisen und dynamisch durch Ändern des Zustands zwischen ihnen wechseln. Im Interface Builder können Sie unter anderem die folgenden Einstellungen für dieses Element anpassen (siehe Bild 23.114):

- *Image:* Hier tragen Sie den Namen der Grafik ein, die Sie innerhalb der Image View anzeigen möchten. Die Grafik muss Teil des Main Bundles und damit des zugrunde liegenden iOS-Projekts sein, um sie hierüber auswählen zu können. Dabei spielt es keine Rolle, ob die Grafik direkt dem Xcode-Projekt als Datei hinzugefügt wurde oder Teil eines Asset Catalogs ist.
- *Highlighted:* Sie können optional einer Image View ein zweites Bild als *Highlighted Image* zuweisen. Abhängig davon, in welchem Zustand sich eine Image View befindet (Highlighted oder nicht), zeigt sie die jeweils zugewiesene Grafik an. Den Zustand einer Image View können Sie jederzeit im Code dynamisch verändern (siehe hierzu den folgenden Abschnitt „Konfiguration im Code“).
- *State:* Hier steht Ihnen die Checkbox *Highlighted* zur Verfügung. Sie dient innerhalb des Interface Builders zum Testen, ob die Image View – abhängig von ihrem Zustand – die korrekte Grafik anzeigt. Ist sie aktiviert, wird der *Highlighted*-Zustand simuliert und entsprechend die Highlighted-Grafik angezeigt (siehe den vorherigen Punkt *Highlighted*). Ist sie hingegen deaktiviert, wird das unter *Image* zugewiesene Bild angezeigt.

  Die Checkbox hat keine Auswirkungen auf den *tatsächlichen* Zustand einer Image View. Dieser kann *ausschließlich* im Code geändert werden (siehe den folgenden Abschnitt „Konfiguration im Code“). Sie dient einzig und allein zu Testzwecken. Entsprechend blendet Xcode eine Warnung ein, sobald die *Highlighted*-Checkbox aktiviert ist, um Sie über genau diesen Umstand zu informieren.

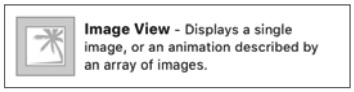


Bild 23.113 Mithilfe einer Image View können Sie Grafiken und Bilder in Ihrer iOS-App darstellen.

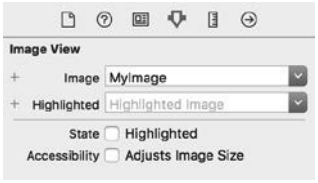


Bild 23.114
Im Interface Builder weisen Sie einer Image View eine Grafik zu (optional auch für den Highlighted-Zustand).

Eine weitere wichtige Einstellung in Bezug auf Image Views betrifft den sogenannten *Content Mode*. Dieser bestimmt, wie die Grafik innerhalb der Image View positioniert und in welcher Größe sie eingefügt wird (siehe Bild 23.115).

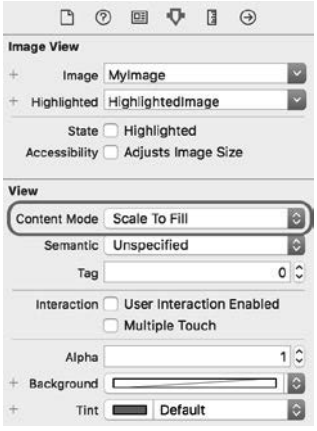


Bild 23.115
Über den Content Mode definieren Sie, wie ein Bild innerhalb einer Image View positioniert und angezeigt wird.

Standardmäßig kommt hierbei der Content Mode *Scale To Fill* zum Einsatz Mit ihm wird das Bild exakt auf die Ausmaße und die Größe der Image View skaliert. Entsprechen Bild und Image View nicht demselben Seitenverhältnis, kann diese Einstellung zu Verzerrungen des Bildes führen.

Anders verhält es sich mit dem Content Mode *Aspect Fit*. Dieser fügt das Bild mit seinem Original-Seitenverhältnis in die Image View ein, wodurch es zu keinen Verzerrungen des Bildes kommt. Stattdessen bleibt die verbleibende freie Fläche der Image View leer.

Den Mittelweg der beiden vorgestellten Lösungen geht die Option *Aspect Fill*. Das Bild wird darüber – genau wie bei *Aspect Fit* – ebenfalls nicht verzerrt, dennoch aber die volle Fläche der Image View ausgefüllt, indem soweit wie dafür nötig in das Bild hineingezoomt wird.

Bild 23.116 zeigt zum besseren Verständnis anhand eines Beispiels, wie sich die verschiedenen Content Modi in der Praxis auswirken.

Bild 23.116 Abhängig vom Content Mode wird das Bild einer Image View unterschiedlich positioniert und skaliert.

Konfiguration im Code

Eine `UIImageView` nutzt Instanzen der Klasse `UIImage`, um Grafiken und Bilder anzuzeigen. Jedes Bild, das man somit innerhalb einer `UIImageView` in einer App anzeigen möchte, muss zuvor in ein `UIImage` gepackt werden.

Es gibt verschiedene Möglichkeiten, um eine neue `UIImage`-Instanz zu erstellen. Einer der einfachsten Wege besteht in der Verwendung des Initializers `init(named:)`. Dieser erwartet den Namen einer innerhalb des Main Bundles gespeicherten Grafik und generiert daraus die passende `UIImage`-Instanz. Main Bundle bedeutet hierbei, dass sich die Grafik innerhalb des Xcode-Projekts der App befindet (entweder als direkt hinzugefügte Datei oder als Teil eines Asset Catalogs).

Dateiendung nicht vergessen

Wenn Sie bei der Arbeit mit `UIImage` auf eine PNG-Grafik zurückgreifen, können Sie sich bei der Angabe des Dateinamens die Dateiendung sparen. Bei allen anderen Formaten müssen Sie zwingend den vollen Dateinamen (inklusive Dateiendung) mit angeben.

In Listing 23.41 sehen Sie ein Beispiel zur Erstellung einer `UIImage`-Instanz auf Basis einer PNG-Grafik namens *MyImage*, die Teil des Main Bundles der zugrunde liegenden App ist. Beachten Sie hierbei, dass es sich bei `init(named:)` um einen Failable Initializer handelt, der also ein Optional oder `nil` zurückliefert. Da in dem Beispiel in Listing 23.41 sichergestellt ist, dass die gewünschte Grafik tatsächlich existiert, wird das vom Initializer zurückgelieferte Optional direkt entpackt.

Listing 23.41 Erstellen einer `UIImage`-Instanz

```
let myImage = UIImage(named: "MyImage")!
```

Auf Basis einer so erzeugten `UIImage`-Instanz können Sie nun wiederum auch eine `UIImageView` im Code mithilfe des Initializers `init(image:)` erzeugen. Ein Beispiel dazu finden Sie in Listing 23.42.

Listing 23.42 Erzeugen einer `UIImageView`-Instanz im Code

```
let myImage = UIImage(named: "MyImage")!
let myImageView = UIImageView(image: myImage)
```

Alternativ steht Ihnen auch der Initializer `init(image:highlightedImage:)` zur Verfügung, mit dem Sie zusätzlich eine zweite `UIImage`-Instanz für den Highlighted-Zustand der Image View übergeben können.

Apropos Highlighted: Wie zuvor bereits bei der Konfiguration einer Image View im Interface Builder erwähnt, erfolgt der Wechsel zwischen Default- und Highlighted-Zustand einer Image View ausschließlich im Code, und zwar mithilfe der booleschen Property `isHighlighted`. Darüber können Sie festlegen, ob sich die Image View im Default- (`false`) oder Highlighted-Zustand (`true`) befindet, indem Sie einfach den passenden Wert zuweisen.

Die einer Image View zugewiesene Default-Grafik kann jederzeit über die `image`-Property ausgelesen und geändert werden. Für den Zugriff auf die Highlighted-Grafik kommt die `highlightedImage`-Property zum Einsatz.

23.6.4.7 Weitere Views

Die iOS-Entwicklung hat noch weitere View-Elemente als diejenigen zu bieten, die in den vorangegangenen Abschnitten im Detail vorgestellt wurden. Im Folgenden stelle ich Ihnen eine Auswahl weiterer nützlicher `UIView`-Subklassen vor und erkläre Ihnen, welche Aufgabe sie erfüllen und wie Sie sie in eigenen iOS-Projekten nutzen können. Weitere View-Klassen werden dann ausführlich in Kapitel 24, „iOS – App-Entwicklung“, vorgestellt.

23.6.4.7.1 Slider

Mithilfe eines Sliders (basierend auf der Klasse `UISlider`) setzen Sie einen Schieberegler in Ihren iOS-Apps um (siehe Bild 23.117). Hierfür legen Sie mithilfe eines Minimal- und eines Maximalwerts den Wertebereich fest, den der Slider abbildet. Wird der Slider vom Nutzer bewegt, wird eine Action-Methode ausgelöst, in der Sie den aktuellen Wert des Sliders mithilfe der Property `value` auslesen können. Sie können die Property auch dazu verwenden, programmatisch einen neuen Wert für den Slider festzulegen.

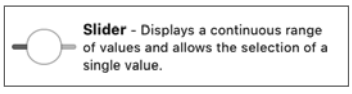


Bild 23.117
Mit einem Slider setzen Sie einen Schiebregler in Ihrer iOS-App um.

23.6.4.7.2 Stepper

Ein Stepper hat eine ähnliche Aufgabe wie der in Abschnitt 23.6.4.7.1 vorgestellte Slider (siehe Bild 23.118). Er basiert auf der Klasse `UIStepper` und erlaubt es, mithilfe einer Plus- und einer Minus-Schaltfläche einen zugrunde liegenden Wert zu erhöhen beziehungsweise zu verringern. Dazu legen Sie einen Minimal- und einen Maximalwert fest, in dem sich der Stepper bewegt und den er nicht unter- beziehungsweise überschreiten darf. Den aktuellen Wert des Steppers können Sie mithilfe der Property `value` auslesen oder auch selbst neu setzen.

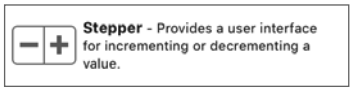


Bild 23.118 Mithilfe eines Steppers können Sie einen zugrunde liegenden Wert dynamisch durch den Nutzer vergrößern beziehungsweise verkleinern lassen.

Wie viele „Schritte" der Stepper bei Tippen auf die Plus- beziehungsweise Minus-Schaltfläche springt, können Sie ebenfalls sowohl im Interface Builder als auch im Code über die Property `stepValue` selbst definieren. Wann immer eine der beiden Schaltflächen betätigt wird, wird auch eine möglicherweise zugeordnete Action-Methode des Steppers ausgelöst.

23.6.4.7.3 Progress View

Mithilfe einer Progress View (basierend auf der Klasse `UIProgressView`) können Sie in iOS-Apps einen Fortschrittsbalken umsetzen (siehe Bild 23.119). Den Fortschritt regeln Sie dabei mit der `value`-Property. Ein Wert von 0 bedeutet, dass der Fortschrittsbalken ganz am Anfang steht, ein Wert von 1 füllt ihn vollständig aus. Um also einen Fortschritt (beispielsweise während eines Dateidownloads) in Ihrer App darzustellen, müssen Sie diesen auf einen passenden Wert zwischen 0 und 1 herunterrechnen und der `value`-Property zuweisen.

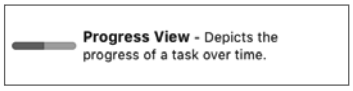


Bild 23.119 Fortschrittsbalken können Sie komfortabel mithilfe einer Progress View in Ihren iOS-Apps einbinden.

23.6.4.7.4 Stack Views

Mithilfe von Stack Views können Sie beliebige View-Elemente in einer Stapelansicht zusammenfassen (siehe Bild 23.120). Die Views innerhalb einer Stack View können entweder horizontal oder vertikal angeordnet werden, die Stack View kümmert sich anschließend selbsttätig um die korrekte Ausrichtung und Positionierung der Elemente. Auch feste Abstände zwischen den Views innerhalb einer Stack View können definiert werden.

Bild 23.120
Mithilfe von Stack Views können Sie beliebige View-Elemente direkt horizontal oder vertikal nebeneinander anordnen.

Eine Stack View hat in diesem Sinne also keine besondere Funktion für den Nutzer, sondern soll uns App-Entwicklern das Leben ein wenig erleichtern. Intern werden die einer Stack View zugewiesenen View-Elemente automatisch mittels Auto Layout arrangiert. Ihr großer Vorteil besteht darin, dass man sich selbst bei der Verwendung von Stack Views keinerlei Gedanken über Constraints und die korrekte Positionierung der Views machen muss; all das übernimmt die Stack View für uns.

Möchte man also eine Ansicht erstellen, in der einfach verschiedene Views entweder neben- oder untereinander platziert werden, erspart einem die Verwendung einer passenden Stack View die aufwendige Konfiguration mittels Auto Layout.

23.6.4.8 Übersicht über Einsatz und Verwendungszweck von Views: UIKit Catalog

In den vorherigen Abschnitten wurde eine Vielzahl von verschiedenen View-Elementen mitsamt ihrer jeweiligen Aufgabe und Funktionsweise im Detail vorgestellt. Wer sich darüber hinaus einmal ein Bild darüber machen möchte, welche `UIView`-Subklassen so alle in der iOS-Entwicklung zur Verfügung stehen und wie sie sich verwenden und konfigurieren lassen, dem empfehle ich einen Blick auf ein Beispielprojekt von Apple: *UIKit Catalog*.

UIKit Catalog ist eine iOS-App, die zu einer Vielzahl von verfügbaren UI-Elementen eine Vorschau mitsamt unterschiedlicher Konfigurationen enthält. Mit ihrer Hilfe erhält man schnell einen Überblick, welche Arten von Views man von Haus aus in eigenen iOS-Apps umsetzen kann und welche Konfigurationsmöglichkeiten sie mit sich bringen (siehe Bild 23.121).

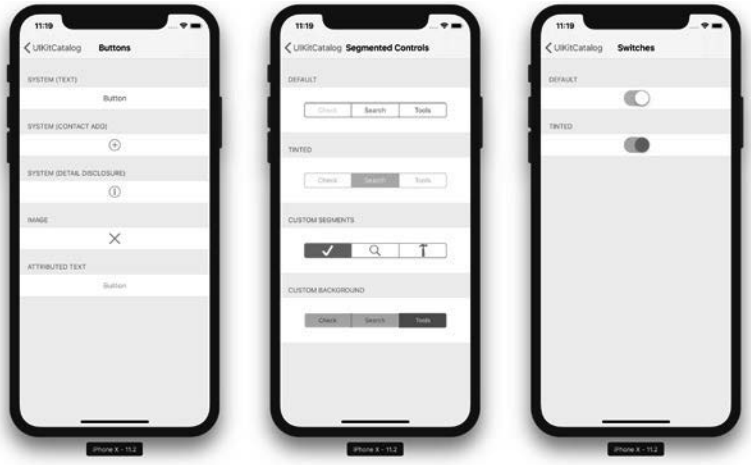


Bild 23.121 Die Beispiel-App „UIKit Catalog“ gibt einen Überblick über die verfügbaren View-Elemente in der iOS-Entwicklung und ihre individuellen Konfigurationsmöglichkeiten.

Mit das Beste an UIKit Catalog ist aber nicht nur die Übersicht verfügbarer View-Elemente, sondern auch die Kopplung mit dem zugrunde liegenden Code. Innerhalb der App finden Sie Verweise auf die Klassen des Projekts, die den jeweiligen Code einer Ansicht beinhalten. So können Sie direkt nachprüfen, wie ein bestimmtes View-Element erstellt und konfiguriert wurde (siehe Bild 23.122).

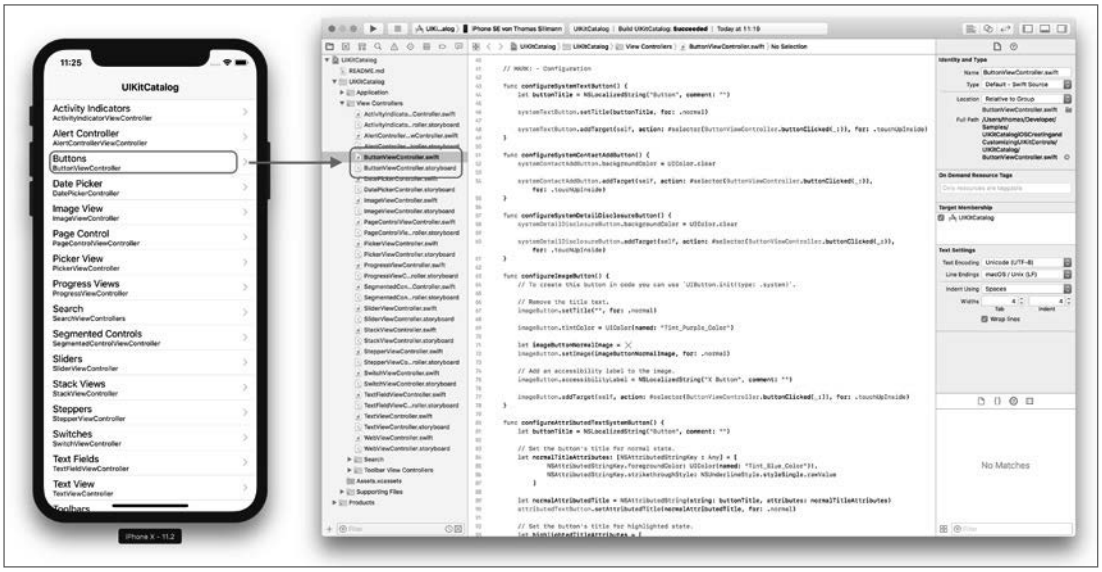


Bild 23.122 Wie wurden die Buttons in UIKit Catalog konfiguriert? Die App gibt Aufschluss darüber und verweist auf die passende Klasse im Code.

UIKit Catalog können Sie über die folgende Website herunterladen: *https://developer.apple.com/library/content/samplecode/UICatalog/Introduction/Intro.html*

23.6.5 Views mit Actions verbinden

Wenn Sie eine von `UIControl` abgeleitete View – beispielsweise einen `UIButton`, einen `UISwitch` oder einen `UISlider` – im Code und nicht über den Interface Builder erzeugen, haben Sie keine Möglichkeit, aus dem Interface Builder heraus eine Action-Methode für die View zu erzeugen (so wie es in Abschnitt 23.5.3.2, „Actions“, beschrieben wurde). Stattdessen müssen Sie in diesem Fall der jeweiligen View-Instanz im Code eine passende Action-Methode zuweisen.

Zu diesem Zweck kommt die Methode `addTarget(_:action:for:)` der Klasse `UIControl` zum Einsatz. Mit ihr führen Sie exakt dieselbe Aufgabe im Code durch, die Sie ansonsten durch Verbinden von Interface und View-Controller ausgeführt haben. Sie rufen diese Methode auf den Views auf, die Sie mit einer Action-Methode koppeln möchten, und übergeben dabei die folgenden Parameter:

- `target`: Hierbei handelt es sich um die Instanz, auf der die Action-Methode ausgelöst werden soll. In vielen Fällen wird es sich dabei um die zugrunde liegende View-Controller-Klasse handeln, in der die jeweilige View erstellt und hinzugefügt wurde.
- `action`: Dieser Parameter verweist in Form einer Selector-Instanz auf die Action-Methode, die im Zusammenspiel mit der View aufgerufen werden soll (bei einem Button beispielsweise durch dessen Betätigung). Die hier übergebene Action-Methode muss innerhalb des zuvor übergebenen Targets implementiert sein.

- `controlEvents`: Hierbei handelt es sich um ein Option Set vom Typ `UIControlEvents`. Es enthält verschiedene Optionen, die die zur Verfügung stehenden Ereignisse definieren, unter denen eine View die ihr zugewiesene Action auslösen kann. Die Option `touchUpInside` beispielsweise entspricht dem Standardwert zur Reaktion auf das Tippen auf einen Button. Die Option `valueChanged` kommt vor allen Dingen für View-Elemente wie Switches, Segmented Controls oder Slider zum Einsatz, die intern einen Wert speichern. Das Ereignis `valueChanged` wird immer dann ausgelöst, wenn sich eben dieser Wert ändert (zum Beispiel indem ein Switch aktiviert beziehungsweise deaktiviert oder ein anderes Segment in einem Segmented Control ausgewählt wird).

  Entsprechend übergeben Sie für diesen Parameter alle Events, die zur Ausführung der Action-Methode führen sollen. Eine komplette Aufstellung aller weiteren Events inklusive Beschreibung finden Sie in der Dokumentation zu `UIControlEvents` (siehe Bild 23.123).

Bild 23.123 In der Dokumentation zu UIControlEvents finden Sie eine Auflistung aller verfügbaren Ereignisse, die eine Action-Methode auslösen können.

Anhand eines kleinen Beispiels möchte ich die Verwendung der beschriebenen Methode `addTarget(_:action:for:)` einmal praktisch demonstrieren. Ziel ist das Erstellen einer kleinen App, die über ein Label und einen Button verfügt. Beide werden dieses Mal direkt im Code und nicht im Storyboard erzeugt. Der Button wird mit einer Action-Methode verknüpft, die den Standardtext des Labels („Label“) in „Update“ ändert. Die vollständige Implementierung der View-Controller-Klasse für dieses Beispiel finden Sie in Listing 23.43.

Listing 23.43 Umsetzung einer Action-Methode für einen Button im Code

```
class ViewController: UIViewController {

    private var label: UILabel = {
        let label = UILabel(frame: CGRect(x: 44, y: 44, width: 300, height: 44))
        label.text = "Label"
        return label
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        view.addSubview(label)
        createActionButton()
    }

    private func createActionButton() {
        let button = UIButton(type: .system)
        button.frame = CGRect(x: 44, y: 88, width: 300, height: 44)
        button.setTitle("Update label", for: .normal)
        button.addTarget(self, action: #selector(updateLabel), for: .touchUpInside)
        view.addSubview(button)
    }

    @objc private func updateLabel() {
        label.text = "Update"
    }

}
```

An dieser Stelle noch ein paar Anmerkungen zur Umsetzung:

Zu Beginn wird für das Label des View-Controllers eine Property erstellt. Darin werden Position und Größe sowie der Standardtext („Label“) festgelegt. Innerhalb der Methode `viewDidLoad()` wird dieses Label dann als Subview der View des View-Controllers hinzugefügt (andernfalls wäre sie nicht auf dem Bildschirm sichtbar).

Ebenfalls in `viewDidLoad()` wird eine eigens implementierte Methode namens `createActionButton()` aufgerufen. Diese Methode stellt das Herzstück dieses Beispiels dar und kümmert sich um die Initialisierung einer `UIButton`-Instanz, der ein passender Frame sowie ein Titel zugewiesen werden. Im Anschluss wird darauf die vorgestellte Methode `addTarget(_:action:for:)` aufgerufen und dabei die folgenden Informationen übergeben:

- `target`: Als Ziel für die Action des Buttons wird der zugrunde liegende View-Controller selbst verwendet, weshalb hierfür `self` als Parameter übergeben wird.
- `action`: Als aufzurufende Methode bei Verwendung des Buttons soll `updateLabel()` verwendet werden. Der Verweis auf diese Methode wird mithilfe der `#selector`-Syntax umgesetzt.
- `controlEvents`: Die Methode `updateLabel()` soll immer dann aufgerufen werden, wenn der Button betätigt wird. Dieses Ereignis entspricht dem Control Event `touchUpInside`.

Durch die Angabe, dass der Button als Action-Methode `updateLabel()` aufrufen soll, muss abschließend natürlich noch eine entsprechende Methode implementiert werden. Damit diese mittels `#selector`-Syntax angesprochen werden kann, muss sie mit dem `@objc`-Schlüsselwort deklariert werden. Die Methode selbst führt schließlich die gewünschte Aufgabe durch und ändert den Text des Labels zu „Update“ (siehe Bild 23.124).

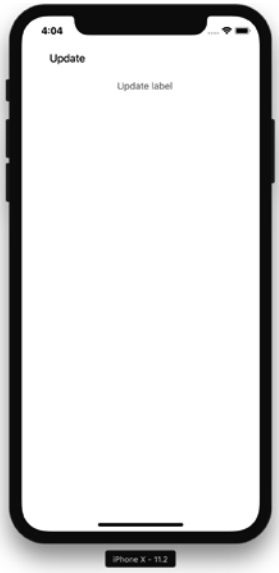


Bild 23.124
Das komplett im Code erzeugte Interface nutzt eine ebenfalls im Code zugewiesene Action-Methode im Zusammenspiel mit dem Button.

23.7 Arbeit mit dem Simulator

Bei der Entwicklung von Apps für iOS spielt der Simulator eine essenzielle Rolle. Er erlaubt es, die App auf unterschiedlichen Geräten mit unterschiedlichen Displaygrößen und -auflösungen zu testen. Gerade zu Beginn der App-Entwicklung ist das der deutlich schnellere und komfortablere Weg, als bei jeder kleinen Änderung eine App auf einem „realen" Endgerät zu installieren und auszuprobieren.

Dennoch hat auch der Simulator seine Tücken und ersetzt – wenigstens in letzter Instanz – niemals die Tests einer App auf echten Endgeräten. Warum das so ist, welche besonderen Funktionen der Simulator sonst noch mit sich bringt und mit welchen Einschränkungen man leben muss, erfahren Sie in den folgenden Abschnitten.

23.7.1 Ausführen von Apps im Simulator

Um eine iOS-App in einem passenden Simulator auszuführen, wählen Sie diesen aus der Liste der verfügbaren Geräte und Simulatoren, die Sie per Klick auf die Schaltfläche rechts neben der Scheme-Auswahl im oberen linken Bereich von Xcode erreichen (siehe Bild 23.125).

Bild 23.125
Sie können aus einer Liste verfügbarer Simulatoren wählen, um Ihre iOS-App auf einem entsprechenden Gerät auszuführen.

Die verfügbaren Simulatoren finden Sie im Abschnitt *iOS Simulators*. Welche Simulatoren Ihnen hier angeboten werden, hängt von verschiedenen Faktoren ab, darunter:

- Welche Geräte unterstützt Ihre App? Ist sie iPhone-only? Dann werden Ihnen auch nur iPhone-Simulatoren angeboten. Das Gleiche gilt umgekehrt für iPad-only-Apps.
- Welche iOS-Versionen unterstützt Ihre App (und welche Simulatoren für ältere iOS-Versionen haben Sie heruntergeladen)? Unterstützt Ihre App beispielsweise iOS 11 und 12, und Sie haben für beide iOS-Versionen Simulatoren installiert und eingerichtet, werden Ihnen in dieser Liste deutlich mehr Simulatoren (pro Gerät und iOS-Version) angeboten (mehr über die Verwaltung von Simulatoren erfahren Sie in Abschnitt 23.7.3, „Verwalten der Simulatoren“).

Download zusätzlicher Simulatoren

Innerhalb der Einstellungen von Xcode (die Sie entweder über das Tastaturkürzel **cmd+**, oder das Menü *Xcode → Preferences…* erreichen) haben Sie im Reiter *Components* die Möglichkeit, zusätzliche Simulatoren für verschiedene iOS-Versionen herunterzuladen (siehe Bild 23.126). Standardmäßig unterstützt Xcode nach der Installation über den App Store nur die zu diesem Zeitpunkt aktuellste und jüngste iOS-Version. Möchten Sie eine App auch unter älteren iOS-Versionen testen (sofern Ihre App solche überhaupt unterstützt), müssen Sie dafür zuvor die gewünschten Simulatoren über diesen Reiter herunterladen.

Sobald eine solche ältere iOS-Version erfolgreich heruntergeladen wurde, richtet Xcode automatisch auch ein Set neuer Simulatoren ein, die auf dieser Version basieren; Sie können entsprechend direkt mit dem Testen loslegen.

Mehr über die Verwaltung von iOS-Simulatoren erfahren Sie in Abschnitt 23.7.3, „Verwalten der Simulatoren“.

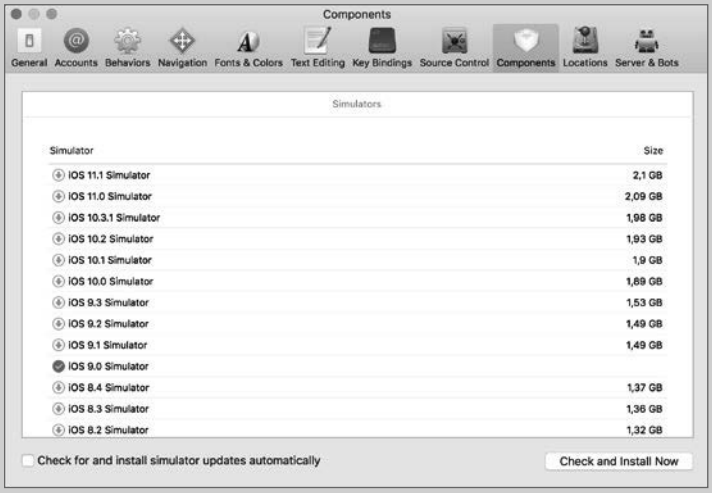


Bild 23.126 In den Einstellungen von Xcode haben Sie die Möglichkeit, zusätzliche Simulatoren zum Testen von iOS-Apps für ältere iOS-Versionen herunterzuladen.

Aus der Liste der verfügbaren Simulatoren können Sie nun das gewünschte Zielgerät auswählen und Ihre App anschließend per Klick auf die Run-Schaltfläche ausführen. Xcode startet daraufhin automatisch den entsprechenden Simulator (sofern noch nicht zuvor geschehen) und führt Ihre App darin aus (siehe Bild 23.127).

Bild 23.127
Eine iOS-App wird im ausgewählten Simulator (hier einem iPhone X) unter der zugehörigen iOS-Version (hier 11.2) ausgeführt.

23.7.2 Arbeiten mit dem Simulator

Auch wenn die verschiedenen iOS-Simulatoren optisch sehr viel Ähnlichkeit mit den entsprechenden Endgeräten besitzen, so ist die Bedienung eine gänzlich andere. Da der Mac nicht über einen Touchscreen verfügt, müssen Sie Apps im Simulator mithilfe von Maus und Tastatur bedienen. Das funktioniert insgesamt auch durchaus gut, jedoch gibt es manche Dinge zu beachten.

Um ein einfaches Tippen auszuführen, klicken Sie schlicht mit der linken Maustaste. Um zu scrollen, klicken Sie an den gewünschten Startpunkt der Geste im Simulator, halten die linke Maustaste gedrückt, und ziehen anschließend die Maus so, wie Sie auch den Finger für diese Geste auf einem Endgerät bewegen würden.

Das Simulieren dieser beiden Gesten erscheint noch relativ eingängig, auch wenn es zu Beginn gewöhnungsbedürftig sein kann, dass man sich nicht durch das herkömmliche Scrollen, wie man es vom Mac gewohnt ist, durch die Ansicht im iOS-Simulator bewegen kann.

Schwieriger wird es schon, wenn man versucht, die sogenannte *Pinch-to-Zoom*-Geste auszuführen. Diese Geste wird beispielsweise verwendet, um Fotos oder einen Kartenausschnitt zu vergrößern beziehungsweise zu verkleinern und es bedarf zu ihrer Anwendung zweier Finger.

Um diese Geste im iOS-Simulator nachzustellen, müssen Sie die **Option**-Taste gedrückt halten. Wenn Sie dann mit der Maus über das Fenster des Simulators fahren, werden Sie feststellen, dass zwei graue Punkte auf dem Display erscheinen (siehe Bild 23.128). Einer befindet sich an der Position des Mauszeigers, der andere an der gegenüberliegenden Position. Wenn Sie anschließend die linke Maustaste gedrückt halten und bewegen, stellen die beiden grauen Punkte zwei Finger dar, über die Sie die Pinch-to-Zoom-Geste nachstellen können.

Bild 23.128
Durch Gedrückthalten der **Option**-Taste können Sie im iOS-Simulator die Pinch-to-Zoom-Geste nachstellen.

Neben diesem allgemeinen Umgang mit dem Simulator gibt es noch eine Vielzahl weiterer Aktionen, die Sie ausführen können. Was ist zum Beispiel, wenn Sie eine App im iPhone X-Simulator mittels Face ID entsperren möchten? Oder wenn Sie Ihre App im Landscape-Modus testen möchten?

Bild 23.129 Über das Hardware-Menü steuern Sie verschiedene Funktionen des Simulators, die Ihnen ansonsten nur auf einem echten Endgerät zur Verfügung stehen würden.

In all diesen Aufgaben kommt das *Hardware*-Menü des Simulators zum Einsatz (siehe Bild 23.129). Darüber haben Sie die Möglichkeit, verschiedene Aktionen auszulösen, die ansonsten direkt an die Hardware eines iOS-Geräts gekoppelt sind. Im Folgenden finden Sie eine Auswahl der verfügbaren Optionen:

- *Rotate Left/Rotate Right:* Damit drehen Sie das iOS-Geräte in die entsprechende Richtung und wechseln so die Orientierung (Portrait oder Landscape).
- *Shake Gesture:* Da Sie Ihren Mac nicht schütteln können (und *sollten* ☺), können Sie mithilfe dieser Schaltfläche eine Schüttelgeste des iOS-Geräts nachstellen. Falls Ihre App auf ein entsprechendes Ereignis reagiert, können Sie so die korrekte Funktionsweise auch im Simulator überprüfen und testen.
- *Home:* Damit kehren Sie auf den Home-Bildschirm des iOS-Simulators zurück. Alternativ können Sie auch den Home-Button des Simulators betätigen oder – bei iPhone X und neuer – den Home Indicator mit gedrückt gehaltener linker Maustaste nach oben ziehen.
- *Lock:* Damit sperren Sie den iOS-Simulator.
- *Siri:* Mithilfe dieser Schaltfläche starten Sie Siri im Simulator. Das ist sehr hilfreich, um eine Integration von Siri in eigenen Apps zu testen.

- *Touch ID/Face ID:* Unterstützt der aktive Simulator die Entsperrung mittels Touch ID oder Face ID, finden Sie einen entsprechenden Punkt im *Hardware*-Menü. Darüber können Sie zunächst mithilfe von *Enrolled* festlegen, ob der Simulator die jeweilige Technik auch tatsächlich unterstützt. Ist *Enrolled* nicht aktiv, wird der Simulator sich so verhalten, als wäre Touch ID beziehungsweise Face ID nicht eingerichtet. Ist *Enrolled* hingegen aktiv und Ihre App fragt an einer bestimmten Stelle nach der Authentifizierung mittels Touch ID/Face ID, können Sie mithilfe der beiden Punkte *Matching Touch* (bei Touch ID) beziehungsweise *Matching Face* (bei Face ID) und *Non-matching Touch* (bei Touch ID) beziehungsweise *Non-matching Face* (bei Face ID) simulieren, ob die Authentifizierung erfolgreich war oder nicht.
- *Authorize Apple Pay:* Darüber können Sie eine Apple Pay-Autorisierung innerhalb Ihrer App durchführen.
- *Keyboard:* Darüber können Sie festlegen, ob Shortcuts bei aktivem Simulator sich auf macOS beziehen oder auf die Steuerung des Simulators *(Send Menu Keyboard Shortcuts to Device)*. Außerdem können Sie mithilfe des Punkts *Toggle Software Keyboard* steuern, ob die virtuelle Bildschirmtastatur bei Texteingaben angezeigt werden soll oder nicht.
- *Increase Volume/Decrease Volume:* Hierüber erhöhen beziehungsweise verringern Sie die Lautstärke des iOS-Simulators.

Neben diesen Optionen zur Steuerung der simulierten Hardware stehen Ihnen auch noch folgende Optionen zur Verfügung:

- *Device:* Hierüber können Sie aus einer Auswahl der verfügbaren Simulatoren und iOS-Versionen wählen und diese zusätzlich zum bestehenden Simulator starten. Das kann nützlich sein, wenn Sie eine App parallel auf unterschiedlichen Geräten testen möchten (siehe Bild 23.130).
- *Restart:* Über diesen Punkt starten Sie den gewählten iOS-Simulator neu.
- *Erase All Content and Settings…:* Hierüber können Sie den gewählten Simulator komplett zurücksetzen. Das kann insbesondere dann sinnvoll sein, wenn Sie viele verschiedene Tests durchgeführt haben und den Simulator bereinigen wollen, um keine Altlasten mit sich herumzutragen, die sich möglicherweise fälschlich auf das Verhalten Ihrer App auswirken.

Bild 23.130 Sie können parallel mehrere Simulatoren auf einmal anzeigen und verwenden.

Debugging im Simulator

Über das *Debug*-Menü stehen Ihnen zusätzlich verschiedene Funktionen zum Debuggen von iOS-Apps im Simulator zur Verfügung (siehe Bild 23.131). Mithilfe des Punkts *Slow Animations* können Sie beispielsweise alle Animationen verlangsamt darstellen. Dadurch sehen Sie genau, was Schritt für Schritt während einer Animation geschieht und ob diese korrekt ausgeführt wird. Mittels *Trigger iCloud Sync* können Sie die Synchronisation mit der iCloud anstoßen und über den Punkt *Location* den Ort ändern, den der Simulator als Ihren Standpunkt verwendet (was sehr nützlich ist, wenn Sie mit der Position des Nutzers in eigenen Apps arbeiten). Auch können Sie bestimmte Bereiche Ihrer App einfärben und das System Log aufrufen.

Bild 23.131 Der Simulator bringt verschiedene Optionen für das Debugging einer iOS-App mit.

Neben all diesen Optionen zum Steuern des iOS-Simulators können Sie diesen auch frei bewegen und in seiner Größe verändern. Klicken Sie zum Bewegen des Simulator-Fensters an den Rand oder klicken Sie auf die Bezeichnung, die unterhalb des Simulators steht. Darüber können Sie das Fenster dann mittels gedrückt gehaltener linker Maustaste bewegen. Die Größe verändern Sie, indem Sie die Maus an den oberen linken oder rechten beziehungsweise unteren linken oder rechten Rand bewegen, bis der Mauszeiger einem Symbol mit zwei Pfeilspitzen an jedem Ende entspricht. Halten Sie anschließend die linke Maustaste gedrückt und ziehen Sie das Simulator-Fenster größer oder kleiner.

Übrigens können Sie den Rahmen, der das Fenster des Simulators umgibt und standardmäßig das iOS-Gerät darstellt, dem der Simulator entspricht, auf Wunsch auch ausblenden.

Deaktivieren Sie dazu im *Window*-Menü den Punkt *Show Device Bezels*. Anschließend sehen Sie nur noch das eigentliche Inhaltsfenster des Simulators ohne die Geräteumrisse (siehe Bild 23.132).

Bild 23.132
Auf Wunsch können Sie sich auch nur das Simulator-Fenster ohne die Umrisse des zugrunde liegenden Geräts anzeigen lassen.

23.7.3 Verwalten der Simulatoren

Alle in Xcode eingerichteten Simulatoren können Sie sich über das *Devices and Simulators*-Fensters anzeigen lassen und Sie können die Simulatoren dort auch verwalten. Um dieses Fenster einzublenden, klicken Sie entweder im Menü auf *Window → Devices and Simulators* oder Sie führen das Tastaturkürzel **cmd+Umschalt+2** aus (siehe Bild 23.133). Das Fenster unterteilt sich in die zwei Reiter *Devices* und *Simulators*, zwischen denen Sie über eine jeweils passende Schaltfläche am oberen Rand wechseln können. Die in Xcode zur Verfügung stehenden Simulatoren finden Sie wenig überraschend im Reiter *Simulators*.

Jeder Simulator setzt sich aus insgesamt drei Bestandteilen zusammen:

- einem frei wählbaren *Namen*,
- dem *Gerätetyp* (iPhone X, iPad Pro etc.),
- und der *iOS-Version*.

In der Liste der verfügbaren Simulatoren am linken Fensterrand werden die Namen angezeigt, ein Klick auf einen der Simulatoren zeigt eine Detailansicht dazu im Hauptfenster an. Dort finden sich auch die Informationen zum *Gerätetyp* und zur *iOS-Version* des Simulators (siehe Bild 23.134).

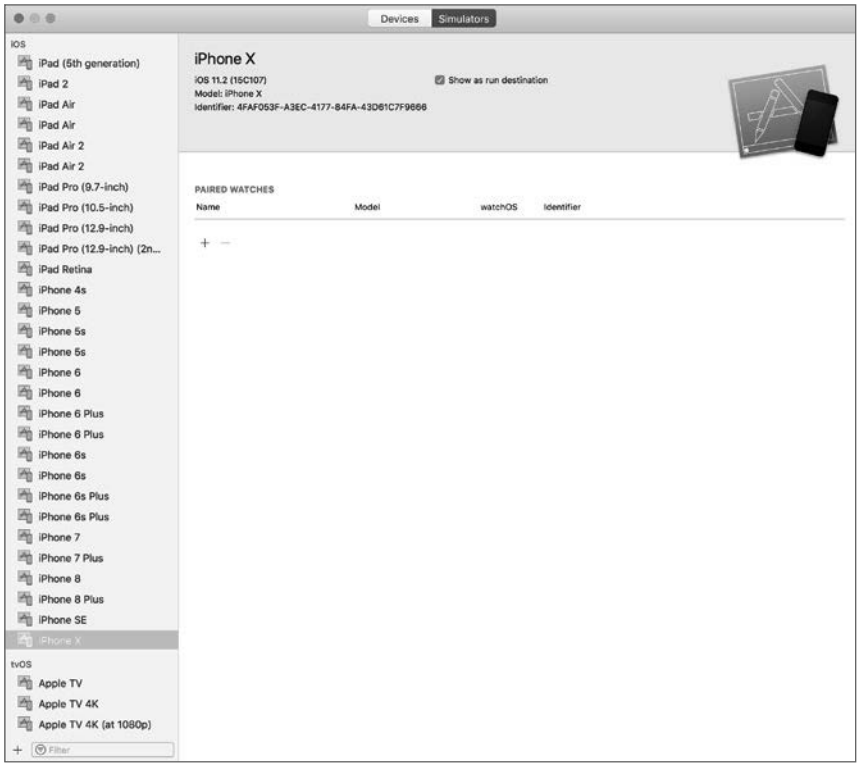


Bild 23.133 Die in Xcode zur Verfügung stehenden Simulatoren finden Sie im Reiter „Simulators“.

Bild 23.134 Jeder Simulator verfügt über einen Namen, einen Gerätetyp und eine iOS-Version.

Bestehende Simulatoren können Sie per Rechtsklick auf Ihren Namen in der Liste links bearbeiten, zur Auswahl stehen die Optionen *Rename* (zum Ändern des Namens) oder *Delete* (zum Löschen des Simulators).

Neue Simulatoren können Sie mithilfe der Plus-Schaltfläche am unteren linken Fensterrand erstellen. Daraufhin erscheint ein Pop-up, in dem Sie den gewünschten Namen, den Gerätetyp und die iOS-Version des neuen Simulators angeben (siehe Bild 23.135). Per anschließendem Klick auf *Create* wird der neue Simulator erstellt und er steht zum Ausführen Ihrer iOS-Projekte zur Verfügung.

Bedenken Sie aber, dass Sie beim Erstellen neuer Simulatoren nur die iOS-Versionen auswählen können, die Sie zuvor über die *Components*-Einstellung von Xcode heruntergeladen und installiert haben (siehe hierzu auch den Kasten „Download zusätzlicher Simulatoren“ in Abschnitt 23.7.1, „Ausführen von Apps im Simulator“).

Bild 23.135
Sie können beliebige weitere Simulatoren in Xcode erstellen.

23.7.4 Einschränkungen des Simulators

So komfortabel und angenehm die Arbeit mit dem Simulator auch ist, so ersetzt er in letzter Instanz niemals abschließende Tests auf echten Endgeräten. Hierfür gibt es mehrere Gründe. Der erste liegt in Sachen Performance begründet, denn der Simulator nutzt die vollständige Hardware des zugrunde liegenden Mac. Wenn Sie also beispielsweise einen iMac Pro mit Acht-Kern-Prozessor und 32 GByte Arbeitsspeicher einsetzen, hat auch der iOS-Simulator Zugriff auf diese Ressourcen (siehe Bild 23.136).

„Toll!“, werden Sie jetzt womöglich denken und haben dahingehend Recht, dass der Simulator auf modernen Macs eine ordentliche Leistung erzielt, die selbst die des eigentlichen iOS-Geräts bei weitem übertreffen kann. Aber genau darin liegt das Problem: Eine App kann innerhalb des Simulators aufgrund dieser weit mächtigeren Ressourcen ruckel- und problemfrei laufen, bereitet auf einem schwachbrüstigeren Endgerät dann aber möglicherweise unerwartet Probleme, weil die benötigte Leistung nicht abgerufen werden kann. Schließlich haben Sie nichts davon, wenn eine App im Simulator absolut rundläuft, wenn sie auf dem eigentlichen Zielgerät Schwierigkeiten bei der Ausführung hat.

Während der Entwicklung ist es durchaus begrüßenswert und sehr angenehm, die vollen Ressourcen des zugrunde liegenden Mac zu nutzen, aber das spiegelt nun einmal nicht die Realität wider.

Darüber hinaus verfügt der Simulator nicht über alle Funktionen, die ein echtes Endgerät mit sich bringt. Ein Beispiel ist der Zugriff auf die Kamera: Im Simulator steht diese Funktion nicht zur Verfügung, Foto- und Videoaufnahmen können Sie also nur auf einem echten iOS-Gerät testen. Das Gleiche gilt entsprechend für komplexe Augmented Reality-Apps, die ebenfalls auf den Kamerazugriff angewiesen sind.

Zu guter Letzt kann nur die Nutzung einer App auf einem tatsächlichen Endgerät zeigen, ob die Bedienung gelungen ist und die App die an sie gestellten Anforderungen erfüllt.

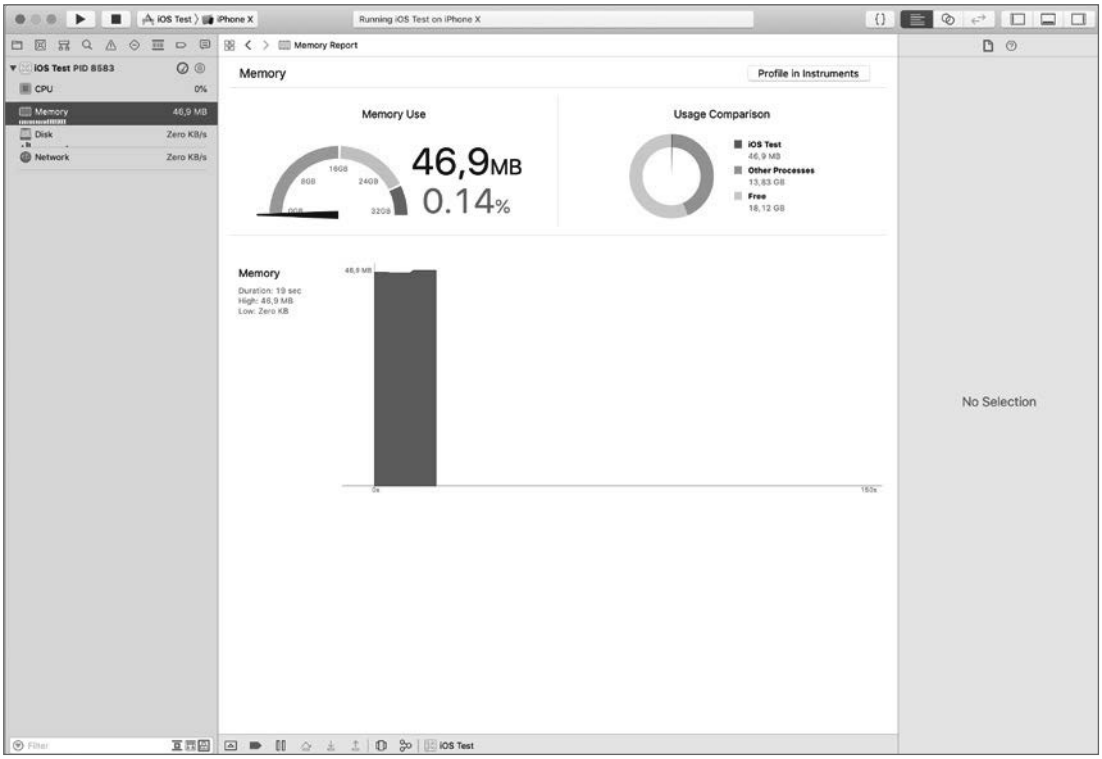


Bild 23.136 Ein iOS-Gerät mit Acht-Kern-Prozessor und 32 GByte Arbeitsspeicher? Im Simulator kein Problem!

23.8 App-Icon

Das App-Icon stellt das optische Aushängeschild Ihrer Anwendung dar. Auch wenn dieses Buch nicht darauf eingeht, *wie* Sie eine Grafik für ein App-Icon erstellen, erhalten Sie im Folgenden aber alle Informationen zu den Rahmenbedingungen, die ein App-Icon erfüllen muss.

Die Mindestgröße, in der Sie ein App-Icon für Ihre App zur Verfügung stellen müssen, beträgt 1024 × 1024 Pixel. Diese Größe wird für den App Store verwendet. Darüber hinaus sollten Sie für die unterschiedlichen Gerätetypen wie iPad Pro und iPhone ebenfalls passend aufgelöste Varianten Ihres App-Icons zur Verfügung stellen (dazu gleich mehr).

Eingebunden wird das App-Icon über den standardmäßig mit einem neuen iOS-Projekt erzeugten Asset Catalog. Dieser bringt bereits von Haus aus ein Image Set zur Unterbringung des App-Icons mit (siehe Bild 23.137).

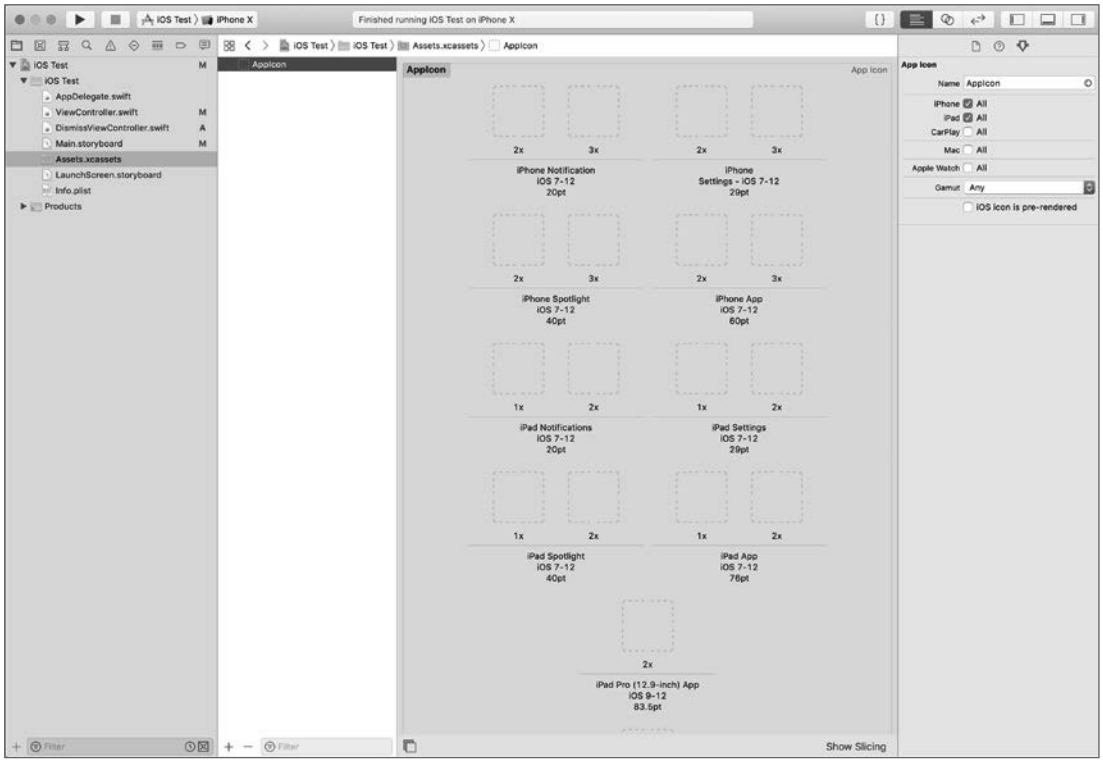


Bild 23.137 Im Asset Catalog Ihres iOS-Projekts bringen Sie das App-Icon in verschiedenen Größen und Auflösungen unter.

Essenziell für den Asset Catalog sind die folgenden Formate und Größen des App-Icons:

- *iPhone App, iOS 7 – 11, 60pt, 3x:* Das App-Icon für das iPhone. Sie müssen es mindestens in dreifacher Pixelgröße bereitstellen, also 180 × 180 Pixel.
- *iPad App, iOS 7 – 11, 76pt, 2x:* Das App-Icon für das iPad (ohne Pro). Sie müssen es mindestens in doppelter Pixelgröße bereitstellen, also 152 × 152 Pixel.
- *iPad Pro App, iOS 9 – 11, 83.5pt, 2x:* Das App-Icon für das iPad Pro. Sie müssen es in doppelter Pixelgröße bereitstellen, also 167 × 167 Pixel.
- *App Store, iOS, 1024pt:* Das App-Icon für den App Store. Sie müssen es in einer Größe von 1024 × 1024 Pixel bereitstellen.

Neben diesen zwingenden Größen und Formaten, in denen Sie ein App-Icon für Ihre iOS-App anbieten müssen, gibt es noch weitere Optionen, ein angepasstes App-Icon bereitzustellen, beispielsweise für Notifications oder die Einstellungen. Auch variieren diese Optionen bisweilen abhängig von der iOS-Version, für die sie angeboten werden.

Wann man verschiedene App-Icons anbieten sollte

Möglicherweise fragen Sie sich, warum Sie ein App-Icon für eine solche Vielzahl an Größen erstellen sollten. Die einfache Antwort lautet: Sie müssen es nicht! Solange Sie wenigstens die vier genannten Varianten anbieten, ist bereits alles gut. Doch es gibt Situationen, in denen es sinnvoll sein kann, gerade für die kleineren Auflösungen ein angepasstes (und nicht einfach nur herunterskaliertes) App-Icon anzubieten.

Das Beispiel in Bild 23.138 soll eine typische Problematik einmal praktisch demonstrieren. Dort ist ein App-Icon in zwei verschiedenen Größen zu sehen: links die am höchsten aufgelöste Variante für den App Store, rechts eine Variante für Notifications.

Wie unschwer zu erkennen ist, ist bei der zweiten Variante kaum noch zu erkennen, was das App-Icon darstellen soll; dafür befinden sich darin zu viele Details. Beim App-Icon für den App Store stört das nicht, da die dafür geforderte Auflösung derartige Details erlaubt.

Bild 23.138 Während man im App-Icon für den App Store durchaus viele Details unterbringen kann, gehen diese bei den kleineren Varianten verloren und es wird möglicherweise nicht ersichtlich, um welche App es sich handelt.

Daher bietet sich an dieser Stelle aufgrund des detaillierten App-Icons an, eine abgespeckte Variante für die niedrigeren Auflösungen bereitzustellen (beispielsweise eine mit einer einzigen Ziffer, so wie im Vergleich in Bild 23.139 zu sehen). Damit schafft man einen effektiven Mehrwert für den Nutzer.

Bild 23.139 Bei sehr detaillierten App-Icons ist es sinnvoll, angepasste Versionen für geringere Auflösungen bereitzustellen.

23.9 Target-Einstellungen

Das Herzstück eines jeden iOS-Projekts ist das zugehörige iOS-Target. In diesem Target werden alle Dateien und Klassen gebündelt, die Sie als Bestandteil Ihrer App ausliefern möchten.

Wenn Sie ein neues Xcode-Projekt auf Basis einer der verschiedenen iOS-Vorlagen erstellen, erhalten Sie automatisch ein solches iOS-Target. Wenn Sie ein bereits bestehendes Projekt für eine andere Plattform (zum Beispiel macOS) oder ein komplett leeres Projekt um eine iOS-App erweitern möchten, müssen Sie selbst ein entsprechendes iOS-Target erstellen. Wählen Sie dazu in Xcode das eigentliche Projekt aus und klicken Sie in der Target-Übersicht auf den unteren Plus-Button (siehe Bild 23.140). Anschließend öffnet sich die Template-Auswahl, in der Sie im Reiter *iOS* im Abschnitt *Application* die passenden Target-Vorlagen für neue iOS-Apps finden und darüber dem Projekt hinzufügen können (siehe Bild 23.141).

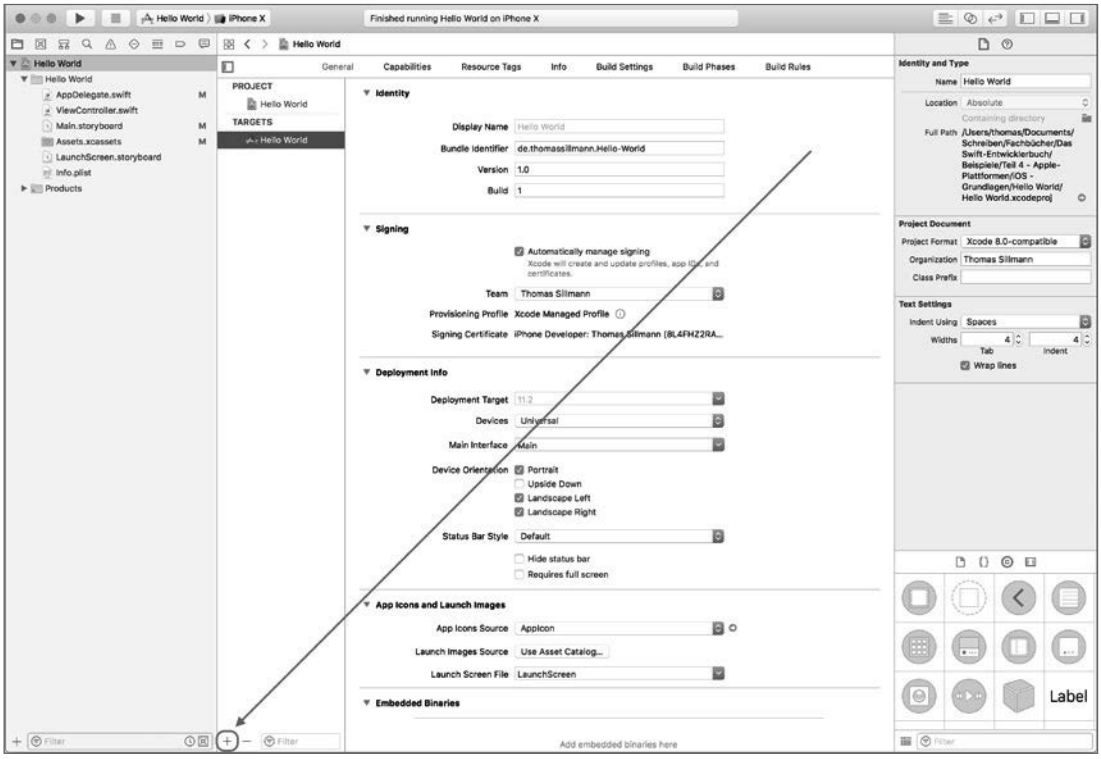


Bild 23.140 In den Projekteinstellungen können Sie ein Xcode-Projekt um ein neues iOS-Target erweitern.

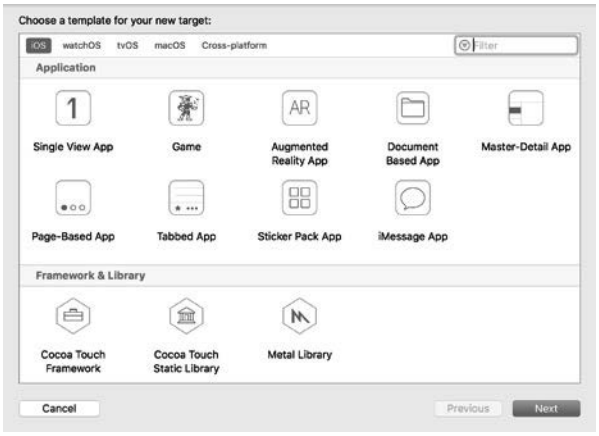


Bild 23.141
Wählen Sie für das neue iOS-Target die passende Vorlage und schließen Sie die Konfiguration ab, um es einem Xcode-Projekt hinzuzufügen.

Über das eigentliche Target selbst können Sie verschiedene grundlegende Einstellungen für Ihre iOS-App vornehmen. Wählen Sie dazu das passende iOS-Target in der Projektübersicht aus (siehe Bild 23.142).

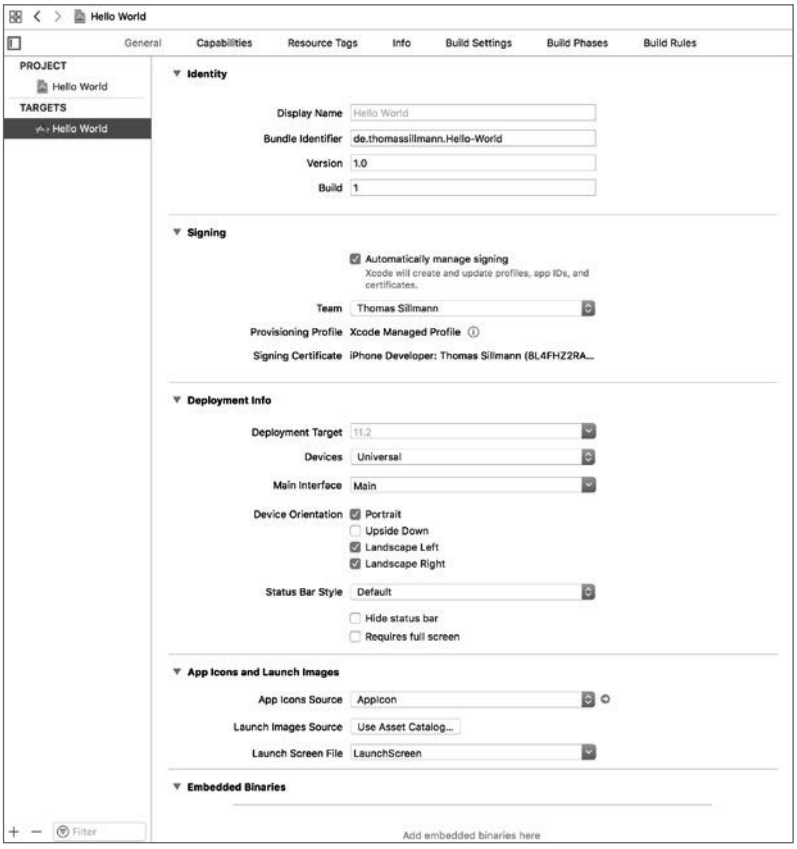


Bild 23.142 Einige grundlegende Einstellungen Ihrer iOS-App werden über das Target festgelegt.

Viele der Einstellungen, die Sie innerhalb des Targets vornehmen, werden intern auf die *Info.plist*-Datei übertragen. Die Target-Einstellungen dienen in diesem Fall lediglich als grafische Oberfläche zum Verändern der *Info.plist*. Das gilt beispielsweise für den *Display Name* (den Namen Ihrer App), den *Bundle Identifier* sowie die *Version*- und *Build*-Nummer, die Sie im Target festlegen können. Diese Werte finden Sie im *General*-Tab im Abschnitt *Identity*. Ebenfalls im *General*-Tab untergebracht sind Informationen zum Code Signing (Abschnitt *Signing*); mehr dazu erfahren Sie in Kapitel 34, „Veröffentlichung im App Store".

Im Abschnitt *Deployment Info* geben Sie einige grundlegende Informationen zur iOS-App an, darunter:

- *Deployment Target:* Die kleinstmögliche iOS-Version, unter der die App noch lauffähig ist.
- *Devices:* Hier wählen Sie, ob es sich bei Ihrer App um eine iPhone-only, iPad-only oder um eine Universal-App (für beide Plattformen) handelt.
- *Main Interface:* Hier hinterlegen Sie den Namen der Storyboard-Datei, dessen initialer View-Controller beim Start der App geladen und angezeigt werden soll.
- *Device Orientation:* Mithilfe der Checkboxen geben Sie an, welche Geräteorientierungen Ihre App unterstützt.
- *Requires full screen:* Ist diese Checkbox aktiv, kann Ihre App auf dem iPad nur im Vollbildmodus und nicht parallel mit anderen Apps ausgeführt werden.

Im Abschnitt *App Icons and Launch Images* ist der Verweis auf das Image Set innerhalb des Asset Catalogs hinterlegt, in dem das App-Icon für die App untergebracht ist. Ebenfalls findet sich dort ein Verweis auf ein sogenanntes *Launch Image*. Dabei kann es sich entweder – genau wie beim App-Icon – um eine statische Grafik aus dem Asset Catalog oder ein Interface aus einer separaten Storyboard-Datei handeln; letzteres ist der Standard. Das Launch Image wird immer beim Starten einer App angezeigt, bis der initiale View-Controller vollständig geladen ist.

Ebenfalls relevant für einige spezielle Funktionen einer iOS-App ist der Reiter *Capabilities* (siehe Bild 23.143). Dort können Sie spezifische Services aktivieren, die Sie für Ihre App verwenden möchten, beispielsweise der Zugriff auf iCloud oder das Anbieten von In-App-Käufen. Weitere Details zu den verschiedenen Elementen erfahren Sie in den entsprechenden Kapiteln dieses Buches.

Weitere Einstellungen

Bei den weiteren Tabs, die Ihnen bei der Konfiguration eines iOS-Targets zur Verfügung stehen, handelt es sich um allgemeine Tabs, deren Einstellungen auch auf den anderen App-Plattformen wie macOS oder tvOS zur Verfügung stehen. Mehr über deren Aufgabe und Nutzung lesen Sie in Kapitel 16, „Grundlagen, Aufbau und Einstellungen von Xcode".

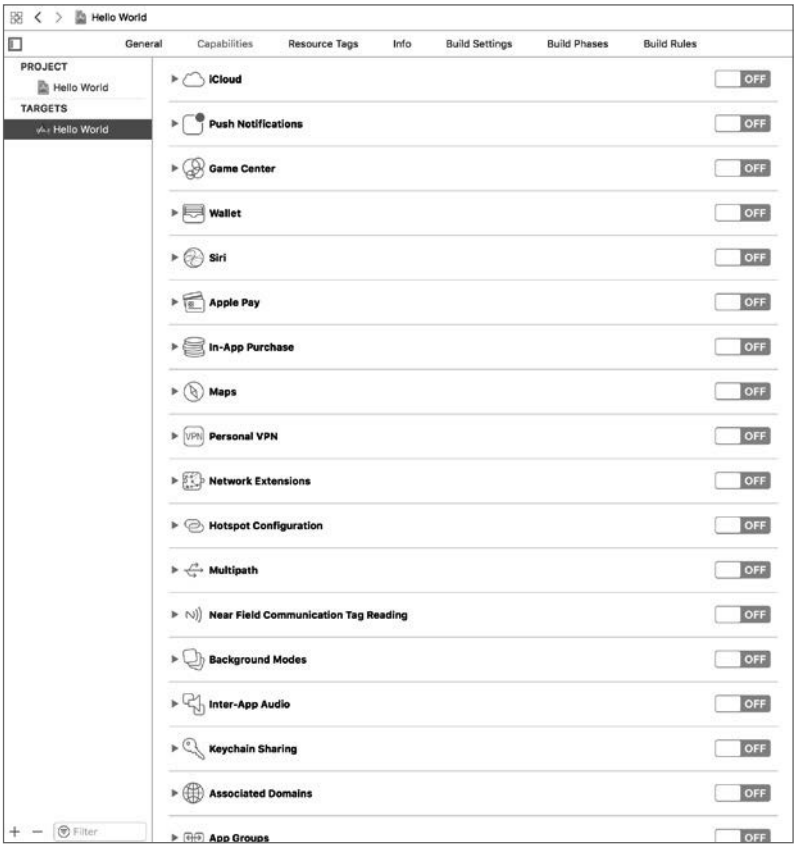


Bild 23.143 Im Capabilities-Tab legen Sie fest, welche zusätzlichen Services Ihre App nutzen soll.

Index

D

E

F

G

H

I

O

P

Q

R

S

T

U

Z