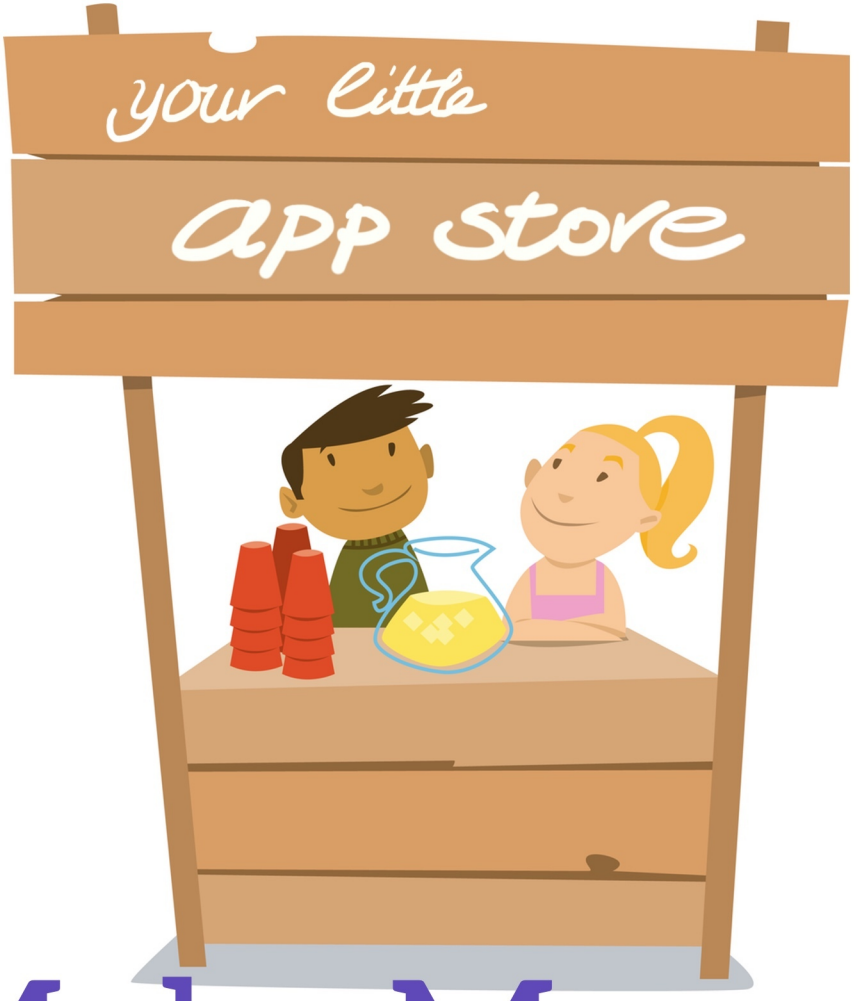
v2.0

Swift 5.1



Make Money

Outside the Mac App Store

— Christian Tietze —

Save Days of Research:
Start selling your App with FastSpring, secure it against piracy, & offer time-based trials ***today!***

Make Money Outside the Mac App Store

How to Sell Your Mac App with FastSpring, Secure It With License Codes Against Piracy, and Offer Time-Based Trial Downloads

Christian Tietze

This book is for sale at
http://leanpub.com/sell-mac-app-fastspring-cocoafob-license-trial

This version was published on 2020-07-14

To my Mastermind Deacon, Rob, and Tristan: I owe you for your support and inspiration.

Contents

We appreciate Christian's efforts in creating a guide that enables Mac developers to sell applications through FastSpring's award-winning e-commerce platform. He has provided detailed instructions to help developers configure key elements of their online sales process. The spirit of community captured in his book reflects FastSpring's mission to connect people globally in the digital economy.

— Mike Smith, CTO FastSpring, December 2015

So You Want To Sell Your App Outside the Mac App Store …

> Writing your own app store and licensor is a rite of passage.
> —Wil Shipley

I wrote this book to save you a ton of work. Depending on your experience level, this book might save you days or weeks if you had to start from scratch and research, implement, and test everything on your own. I know because it took me ages to get confident in the process back in 2013 when I figured things out for my first commercial macOS app, *WordCounter*. It was a lot of extra work I'd rather have skipped for my first commercial product. I can still remember my high levels of anxiety when I started to implement the parts that make an app purchase-able. It's very embarrassing if things go wrong, especially when it comes to other people's money. Understandably, people get angry when their credit card is charged and they don't receive a working product in return. So I wanted to make sure that absolutely nothing could go wrong. How-to's on the intricacies are few and far between. In this book, you'll learn everything to turn your app into a product that can be sold.

With the Mac App Store, you only have to provide a working binary, upload it, pass the review, and that's it. You can focus on the features you want, no need to worry about the infrastructure. That sounds like a compelling argument at first since it removes much of the perceived burden, but you'll see that the things you learn from this book aren't complicated at all. They're just not talked about a lot and thus result in a lot of wasted time fiddling around. This book fixes that problem for you.

Remember the decision to sell *outside* the Mac App Store does not prevent you from selling *inside* the Mac App Store. It could make sense to ditch the App Store completely, but it can also make sense to try to reach different audiences via different platforms in parallel.

I'm all for empowering developers to achieve their goals. The Mac App Store might hinder you on your way; here's a short rundown of things you should consider when you figure out how you want to sell your app so we're both on the same page:

- Can you pull of your own marketing? The App Store may be an uninspiring warehouse, but it's a warehouse your old aunt might learn to operate and find stuff in. The discoverability on the App Store might be minuscule without any marketing, but its greater than 0.
- Can your app be Sandboxed easily? If not, the App Store won't be viable.
- Do you want to participate in bundle sales? App Store doesn't do that.
- Do you want to provide education or bulk discounts?
- Does your business target other businesses, and do you regularly need to provide non-personalized bulk licenses? If so, you will want the flexibility of an e-commerce provider like FastSpring.
- Do you want to experiment with business models like cross-platform subscriptions? Might be way easier to pull off on your own.
- Do you want to make piracy of your app harder, for example by validating how many machine someone can run your app on? Then you will want to activate apps via your own servers.
- Will your business suffer if Apple pulls your app from the Store, even if by mistake? It's a numbers game: a sufficiently large corporation like Apple will eventually make mistakes. PayPal and amazon are known for closing accounts a bit too proactively. Maybe you remember the incident when the popular API documentation app *Dash* was pulled from the App Store and its dev was very happy to have another way to sell his app. I strongly advise you set up a safety net to make your business less fragile.
- Do you want to cut cost? FastSpring has a flat fee of 8.9% per transaction, as opposed to Apple's 30% cut *after deducting value-added tax (VAT).* Say you sell an app for €9.99 to a German customer. After a 19% VAT reduction of €1.90, Apple takes 30% of the remaining €8.09 and you'll earn a depressing €5.66 – whereas the same app sold via FastSpring would leave you with €8.14.

The folks of MacPaw and later Setapp conduct regular surveys about the state of the Mac developer happiness you might want to check out if you want to know more about the obstacles developers face.[1]

As of this edition's release, I am selling four macOS apps with the techniques outlined in this book, one of them also available on the Mac App Store, and I helped clients set

[1]You can find the results online: https://devmate.com/mac-dev-survey. It was re-released and got an update in 2017 and 2018: https://setapp.com/mac-market-survey-2018.

up their apps for sale on FastSpring, too. Meanwhile, this book goes beyond the mere personal preference of yours truly and will teach you the implicit industry standards of selling outside the App Store I discovered from interviews with senior independent Mac developers.

Goal of this Book

You'll learn everything you need to run your own app store. You'll be pretty much set up to sell you software immediately: a web presence is all you need in addition.

The ultimate goal of this book is to get all the obstacles out of your way so you can start using FastSpring in your app today. Depending on your reading speed, you can start this book in the morning and have working license verification implemented by the end of the day. All you need is in this book.

Why FastSpring?

FastSpring isn't the only service on the market to sell apps. You have lots of options to handle digital purchases nowadays, depending on what you need. Opposed to simpler approaches involving PayPal, Stripe, or gumroad, FastSpring does offer much more flexibility. You can generate license codes to protect your app, and if you want to add in-app purchases or subscriptions, you can interact with their API to retrieve customer information. All the while, they handle value-added tax (VAT) for you and have a low flat service fee of 8.9% per purchase. Their service is comfortable to use and powerful at the same time.

In 2013, I picked FastSpring to sell my apps, mostly because of social proof. Lots of indie teams I respected were using FastSpring since before the Mac App Store even existed. I asked around, gave it a spin and immediately liked how things worked.[2] And I still love their service. I bet you'll like them as well.

What You'll Learn

From a technical perspective, you'll learn how to perform the following tasks:

[2] You can read about a small survey from 2009 where FastSpring scored quite high.

- Set up a product for sale with FastSpring, including license code generation and sending order confirmation e-mails with activation links.
- Guard your app against software piracy to some extent, requiring license code and name to let users pass.
- Add a time-based trial to your application. *You can copy this code right into your existing app.*
- Implement an activation handler in your app which verifies the license information before unlocking the app. *This, too, can be copied right into your existing app.*

I'll show you all these things in detail so you can be certain that everything is taken care of:

- how to set up FastSpring as the store where people buy your software,
- add automatic license key generation to the checkout process,
- implement license code verification in your app,
- create a time-based trial,
- offer in-app purchases,
- set up app updates,
- and how to figure out a price and prepare the release.

Optionally, you can even learn how to handle in-app purchases or offer a store front from right within your application using FastSpring's *Embedded Store SDK* in the appendix.

All in all, you only have to take the sample code of this book and copy the parts you want to use into your own project to get started. If your app is already functional, it may take you no more than an hour or two to put your app under copy protection and add a time-based trial – depending on your existing code, of course.

The sample code will be written in the latest stable version of Swift. I'll update the book regularly, so check back for a new e-book version download when you notice Swift has changed.

Structure of the Book

The process I'll show you isn't complicated, but I separated the implementation into multiple steps so you can learn how to scale everything. Here's a quick synopsis of the upcoming chapters.

"**Copy Protection with License Codes**" establishes a few technical terms used throughout this book and prepares you for the implementation ahead.

Set Up Your App for Sale on FastSpring is about getting comfortable with FastSpring. You'll set up your web store and add your product in such a way that people could start buying it. Having this part of the process ready helps when you test if the application code does its job.

The implementation in your app is split up in two parts explained in the next two chapters, so you can take whatever you need and leave the rest. They are very practical and there's an accompanying example app for each chapter for quick reference.

Sample App 1 will teach you how to add the code for app activation. When you have done this, you can verify that the FastSpring code generator works as expected. You can upload the resulting app for sale already, but people would have to buy a license up front, much like paid-up-front downloads in the App Store. But that's not very attractive, so …

Sample App 2 will show the process of implementing a time-based trial mode in your application to enable users to test-drive your app for a while. This chapter is technical, and we'll discuss different trial options. The trial mode is purely your app's responsibility and will not affect your setup with FastSpring.

The **Appendix** is a collection of self-contained sections that explain how to implement optional and advanced features.

Get Free Stuff from My Lab

I can keep you in the loop for projects I'm working on.

Sign up for my very low-volume newsletter to get a private preview of my upcoming programming books, be invited to software betas, and get priority access to my online courses:

https://christiantietze.de/newsletter

My Other Books

Clean Cocoa is my mission: uncovering best coding practices so you can create great apps for macOS and iOS with beautiful, maintainable, and testable code.

Check out my other books online: https://christiantietze.de/books/

Available titles as of this release:

- **Exploring Mac App Development Strategies**, where I show how a modularized and scalable app can be built even when you decide to use Core Data for persistence, which is usually entangled everywhere.
- In **Creating Multi-Process Mac Applications** I show how an app can utilize and communicate with XPC services effectively to perform tasks in the background and sync data between different processes.

Copy Protection with License Codes

Maybe you didn't expect the term "copy protection" here. The term is closely related to Digital Rights Management (DRM) which has gained a very bad reputation on the internet. From my experience, it seems most developers (or "hackers", as some still call themselves in the tradition of Eric S. Raymond) loathe the notion of copy protection in general. In its most general form, the sentiment states that information should be free, music should be free, video and games and code, too. And when you absolutely have to pay to obtain it, then nobody should hinder you from storing and sharing the data as you like, they say. That's where most people think the music industry screwed up, and why some people prefer to buy games on GOG.com (no DRM) instead of Steam (has DRM).

I can get behind the latter easily: when you buy something, you should be able to use it for as long as humanly possible. That's why I have qualms with most, but not all, subscription-based business models. You can find my take on the topic in the appendix.

So that's what we're aiming for: some hurdle to force users to pay once for using your application. It sounds harsh, but yes, it's about *forcing* people to pay. You could give your app away for free and add a tip jar if you need money, after all. But the business model of paying to unlock features is gating users from having their computer execute parts of the binary application that is already stored on their computer. It's very artificial, if you think of it. Thankfully, it's a simple exchange, and people generally seem to understand it very well: you give money and get software. Regular folks compare this to buying groceries, which is technically not the same, but their frame of mind is what really counts here.

Before we dive into the technical implementation of making sure all the people one day either buy a license or stop using your app, we'll start with some general explanations of the process.

Terms Used Throughout This Book

We'll be talking about "keys" and "codes" a lot. To prevent confusion, I want to introduce a couple of important technical terms.

FastSpring
: That's your e-commerce solution. It's your storefront and product management system. It is also the hub of your own little customer support center where order details are stored. You'll be the person in charge when people request refunds, so you need access to all orders quickly.

CocoaFob
: A set of algorithms, available as Open Source scripts, to generate and verify license codes. It's supported by FastSpring and very easy to implement.

  Source is available at: https://github.com/glebd/cocoafob

Public Key
: OpenSSL-generated random number used for 3rd parties to verify the data comes from you. Used for e-mail to tell recipients it was really you who sent the mail. Used for license code verification in our case.

Private Key
: OpenSSL-generated random number you have to, well, keep private. It's needed to create and sign the license key data. Should be kept private because you can infer the public key from it.[3] However, we're going to need to give it to FastSpring for them to generate license codes. So please don't re-use this private key for anything else. We'll discuss the details in the upcoming chapter.

License Code
: Also called "registration code", this is the string of characters your customers will receive after they buy your app.

  CocoaFob-generated license codes should look familiar if you ever bought a Mac app outside the App Store. They tend to look like this, broken into multiple lines for this page's layout:

[3] Thanks to Jorge D. Ortiz Fuentes (https://powwau.com/en/) for corrections of the key descriptions (and a lot more). For encryption/decryption of information, both keys can be used for both actions. But always keep the private key secure.

```
GAWQE-FCUGU-7Z5JE-WEVRA-PSGEQ-Y25KX-9ZJQQ-GJTQC-
CUAJL-ATBR9-WV887-8KAJM-QK7DT-EZHXJ-CR99C-A
```

Licensee
: Also called "license name" or "registration name", this is the name of your customer your app is licensed to. We will set up license code generation to be based on the customer's name, but you can omit this information from the process or use company names if that better suits your business. We'll talk about your options later.

License Code Verification
: Also called "activation", this is the process where the user enters a license code and the app unlocks itself or additional features if the code is valid.

  This is the part you have to add to your app, where you use the public key to decrypt the license code and test the extracted information for matches. We'll discuss the implementation in Chapter 2.

Licensing Is All About a Big State Change

The whole topic of protecting access to features of your application unless people pay for it can be boiled down to handling a transition of your app from "unactivated" state to "activated" state.

If you want, you could have a global variable in your code called `appIsUnlocked` and set it to `true` upon activation. We'll be using a more sophisticated approach, but in the end, a single boolean value is a very good representation already.

To manage the state change in a clean way, and since it's a very important part to get right, it pays off to implement all this as expressively as possible. You don't want to guess when and why your app is locking and unlocking itself in two years from now. We'll try our best to avoid creating tomorrow's legacy code when we implement the activation process.

Think about a license code-activated application with a 30 day time-based trial. During the trial, users should have access to the full feature set. After the trial has ended, some or all features should be inaccessible unless the user enters a valid license

code. In some sense, the activated app is identical to the app in trial mode, only without a clock ticking. How do you represent this?

The state the app is in is determined at least during app launch. Changes can happen during the app's runtime when the trial period is up or the user enters a license code. The app has to represent its current state and its transitions.

With Swift, it's trivial to represent the state of an app that has to be activated at first launch as an enum with an associated value for the details:

```
enum Licensing {
    case registered(License)
    case unregistered
}
```

It only gets a bit more complex if you want to support a time-based trial:

```
enum Licensing {
    case registered(License)
    case trial(TrialPeriod)
    case trialExpired
}
```

The associated `License` type is based on license code and licensee name for the sample apps of this book:

```
struct License {
    let name: String
    let licenseCode: String
}
```

And the `TrialPeriod` type can be equally simple:

```
struct TrialPeriod {
    let startDate: Date
    let endDate: Date
}
```

That's the foundation of everything we're going to do: to give the application a means to find out which state it's in at each moment, and to handle transitions. This is a first representation of the technical terms from the last section in code.

You could capture different details in your `License`, like a company name, an expiration date, or whatever else you like. For the sake of this book, we'll stick to the very common combination of license code and licensee name.

Tyler Hall researched the rates of piracy of his apps and found out 83% of the users of VirtualHostX were using a pirated copy. Piracy was strong for quite some time – until he added server-side validation to unlock license codes. I remember how badly gamers reacted and still react when single-player computer games required an internet connection on first launch. That was a lot of years ago. Nowadays you can assume permanent connectivity for most of your customers. Doesn't mean they'll be happy about it, because once your server is gone for good, they cannot unlock the product anymore.

In his blog post on piracy[4], Tyler listed a few other measures against software piracy that did lower the number a bit, too, although not by much. Bugging the user with a personalized guilt-laden information dialog converted about 5%. Giving pirates a one-time discount to convert to real customers worked for about 11%. Craig Scott of toketaWare talked about "good will discounts" to turn angry prospects into happy customers when you surprise them with a discount. It seems to work with people who want to use your app for longer than just the trial period, too. The deal has to be good from their point of view.

Here's a rehash of the battle plan for the next chapters in case you missed it in the intro:

We will start with setting up the app in your FastSpring dashboard. Then we'll implement a simple license check in the app's startup routine which determines if

[4] https://tyler.io/experimenting-with-piracy/

the app is allowed to run, or if the user only sees the dialog to enter license details. In the final chapter, we're going to add a trial mode to enable users to test drive our software before they decide to buy.

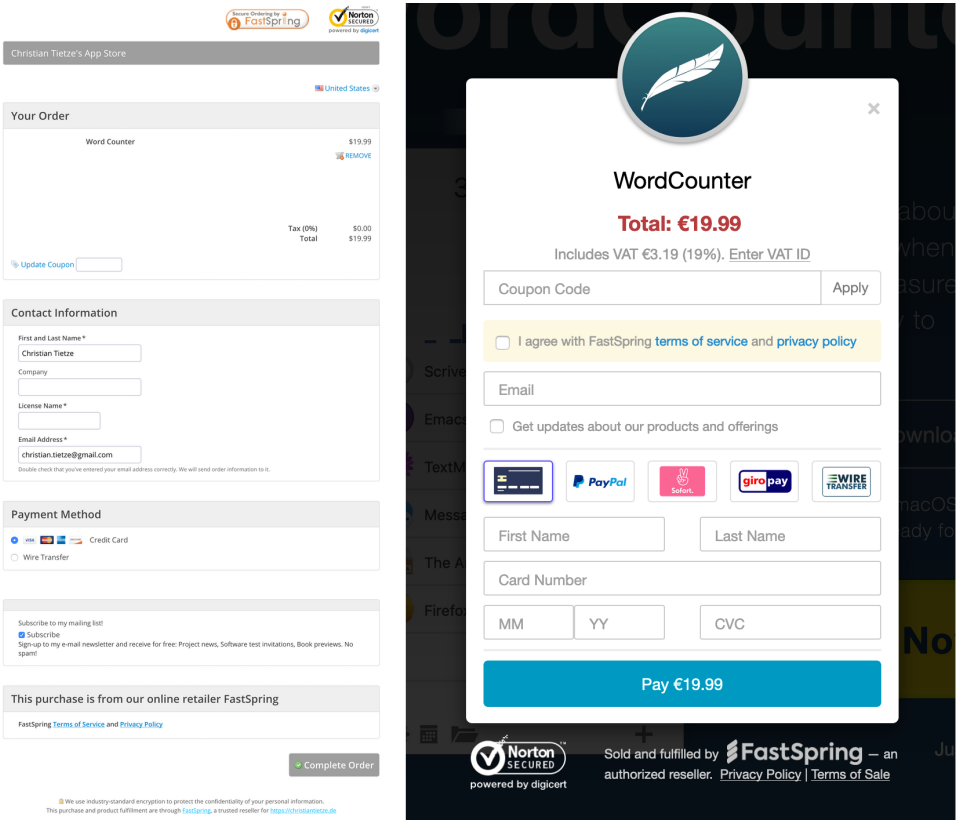
Set up Your App for Sale on FastSpring

In this chapter, we'll configure your app listing for your very own store. FastSpring will process your purchases. Their backend will take care of all the license code generation during the checkout process. This is part of the "order fulfillment" we're going to configure.

FastSpring offers two store backends in parallel: The old one, called "Classic Commerce", and the new one introduced in 2016, called "Contextual Commerce". Both systems differed for quite a while, but nowadays, I think it's safe to say that Contextual is the clear winner for app developers because it is on par in features and even exceeds Classic. You get a proper REST API in Contextual, and a much nicer checkout experience. Order confirmation emails aren't ugly anymore, either!

I'll show you how you can set up both store backends, but will focus on the functionality of Contextual because it's clearly the way forward.

To see what you can do with the Contextual store, have a look at the examples and demos on https://www.fastspringexamples.com/. You'll see that the FastSpring Contextual Commerce backend can power a shopping cart *on your own website.* The examples include a preview of different settings, too, for example how the checkout form differs from selling digital to selling physical goods.



Classic's checkout page from the 90s to the left, Contextual's modern popup checkout UI displaying on top of a website to the right, both for one of my apps

Contextual is the new default when you create an account. You can later opt-in to add access to the Classic store to your account via a support ticket. The same applies if you find you didn't get access to the Contextual backend initially: just ask the support team to put you on the other backend, too.

To get started, head to FastSpring's sign-up page[5] if you don't have an account already. Account activation may take a little while because it involves a human.

If you already have an account set-up, great! Proceed to the next section in this chapter.

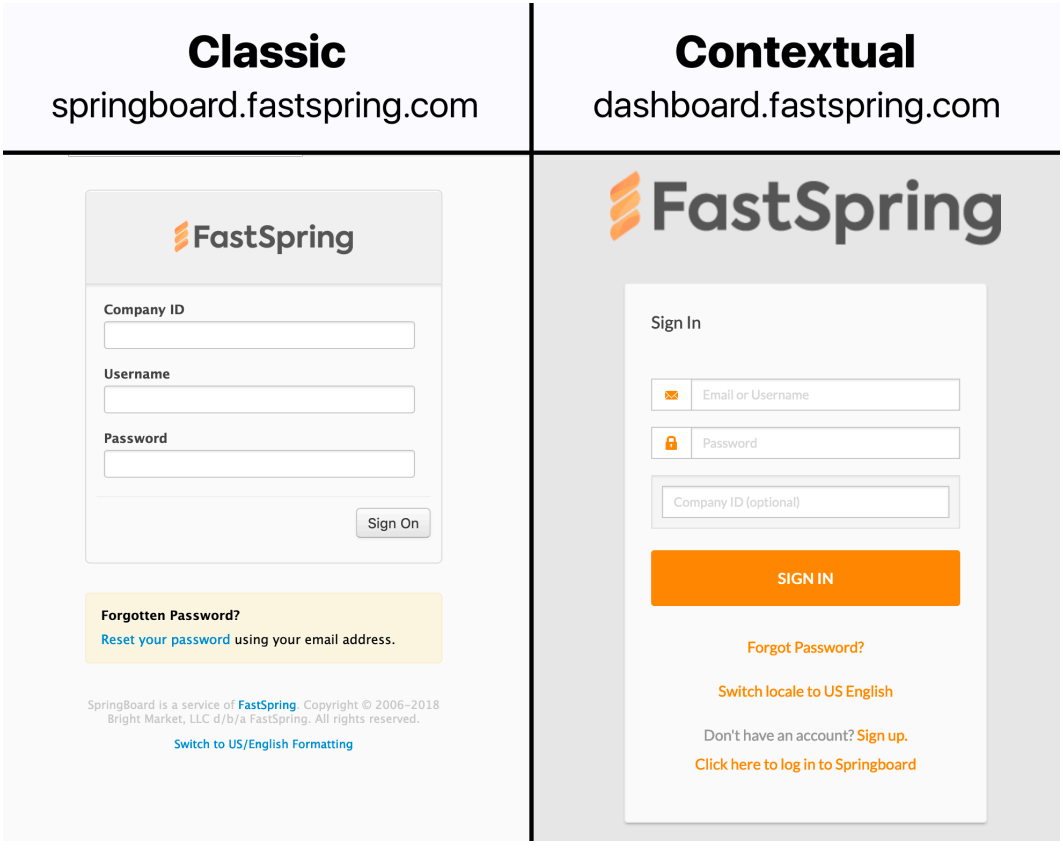
If you still wait for account approval, you may want to skim the following sections to get a feel for the process and the parts involved, and come back later to create

[5] https://www.fastspring.com/sign-up

the store listing. In the meantime, head to the next chapter and learn how you can prepare your app to be activated via license codes.

Setting Up Your App as a Product Listing

Let's make sure you got access to the Contextual store, first. It's pretty easy to discern Classic from Contextual by their URL and looks. When you get your login link, you can infer the backend you got access to by the URL and the looks of the login form.



Classic vs Contextual: login forms and URLs compared

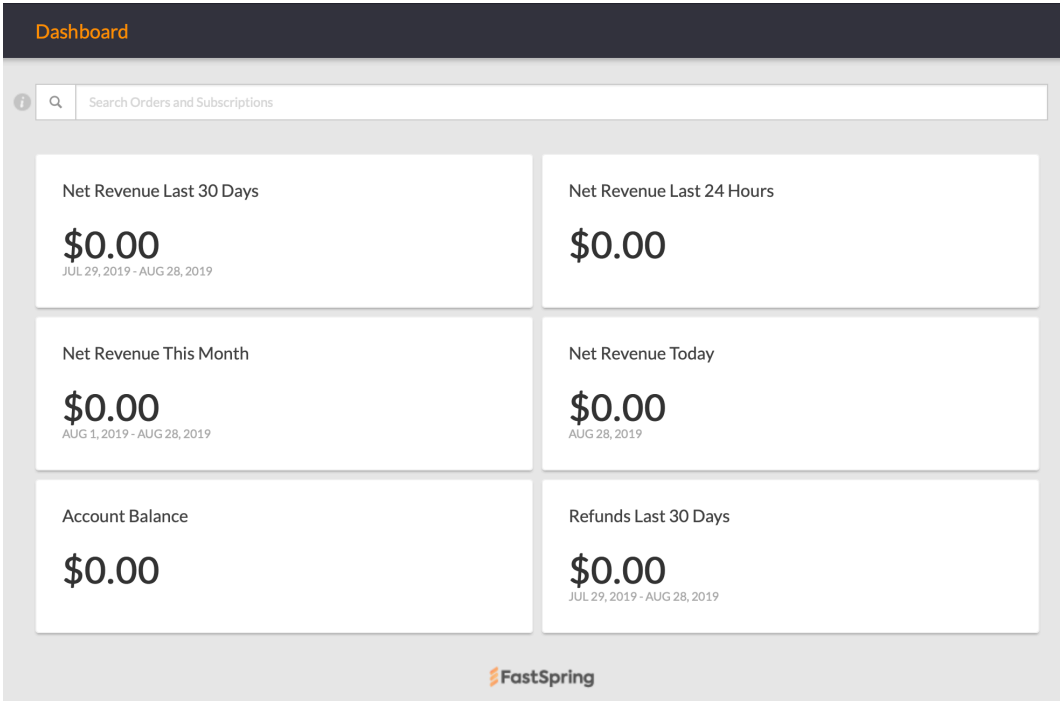
Sign in to your Contextual Dashboard[6]. After signing in, you have to set up your app for sale. In the process, I'll help you find your way around in the FastSpring backend.

[6] https://dashboard.fastspring.com/

This is our action plan:

1. We will create a new product listing with details for your app, plus set up the license generator.
2. We will configure two separate store fronts, so you know how to do this whenever you need to. One is a regular checkout page with a shopping basket you can link people to, the other is a popup you can embed on your website for a faster checkout experience.
3. Finally, we'll be putting the app on both stores's virtual shelves, so to speak.

I'll be saying "product listing" and "app" interchangeably here. Keep in mind that you can sell all kinds of goods on FastSpring. It can be any digital good, like software, music, e-books, service subscriptions – you name it. FastSpring even allows to configure production and shipment of CDs, for example. Or you can create a jewelry store. Naturally, FastSpring's wording is very general to fit all users. So don't look for the words "application" or "software" when you navigate around on FastSprings dashboard or in the docs, they use the more general term "product" everywhere.



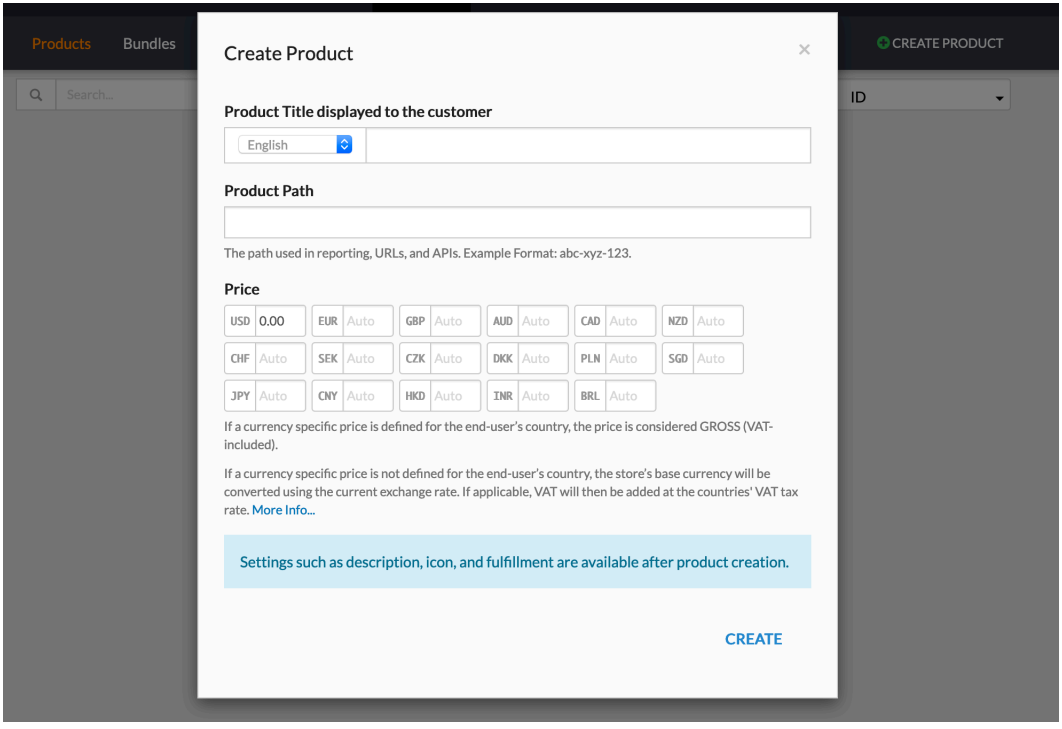
The empty dashboard after logging in for the first time. Imagine one day logging in and the sales numbers being surprisingly high in the top-left tile.

Most navigational tasks in the backend are split up in two tiers: the topmost navigation bar, which is static, and a page sub-menu just below it. In the Dashboard right after logging in, the sub-menu is empty save for the page title, “Dashboard”. You’ll want to keep an eye on this part of the navigation since it will present options, save and cancel buttons, and sometimes “go back” navigation action.

Create your first product listing

Click the “Products” main menu item to go to the listing of all your products in the current store.

The sub-menu now displays different tabs, “Products” (active), “Bundles”, and “Subscriptions”, which show your product listings by the respective type. To the right, it has a “Create Product” button. Click that.

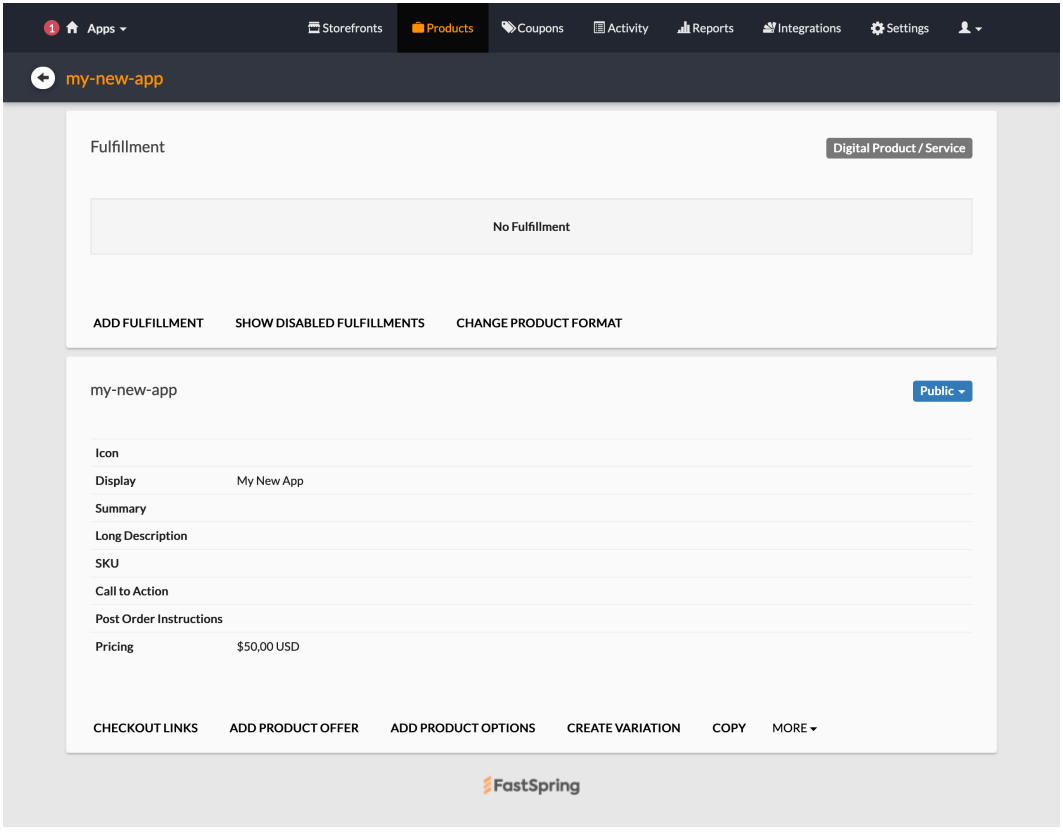


Product creation dialog.

We are going to call the app "**My New App**" in this book. That's the "Product Title" displayed to the customer. You may customize the app's product path for store URLs or leave the default.

Make sure to add a price. (I'll use USD 50.) The default reference currency is USD. You can leave other currencies empty to have them automatically calculated during the order checkout process based on current exchange rates, but not USD. When you remove the USD value, the product will be *free* for U.S. customers. Exchange rates are not used for conversion from e.g. EUR to USD, only the other way around! So do make sure to never leave the USD field blank.

Hit the "Create" button and have a look at the product detail page you are redirected to.



Empty product detail page right after creation

The sub-menu now displays the product path ("my-new-app") and a backward navigation button.

In the content part, there are two groups.

The first is titled "Fulfillment"; that's a list of actions performed by the system when an order completes. We will be setting up the license generator here in a minute.

The second group has the product path again as title and shows the details of the product listing for store fronts. You can customize the product icon, title, short summary for product listings, and long description for product detail pages in the store. It's not mandatory, but I suggest you write different dummy strings into these fields to see where the details are actually displayed.

Keep an eye on the "Post Order Instructions" field in the second group. That's

Markdown-formatted text customers will see when they complete a purchase. You can include app activation instructions and a thank you note here, as we will be doing later.

In the top-right corner of the product listing group is a button titled "Public". This affects whether customers can see the product in your live storefronts. I think it's good practice to immediately switch that to "Private" until you're finished with the product setup.

Product customization options explained

The action links at the very bottom of this group are:

- "Checkout Links": click on that to quickly access a test storefront with the product details.
- "Add Product Offer": use this to display special offers in your store fronts. Once this product is in the basket, these offers will be displayed to the visitor as a cross-sell or upsell. It can make sense to have a discounted product variant that you exclusively use for these scenarios but which are otherwise hidden from the store.
- "Add Product Option": use this for example to model different pricing tiers for subscription services. Additionally, you can offer multiple choice options to e.g. pick parts of a whole. I don't know what possible use this has for software apart from bundles, which I suggest you create differently, but it's there if you need it.
- "Create Variation": use this to prepare a variant of the product that inherits all of the base product's properties, but where you can override the price, icon, descriptions, etc. One scenario is to create a variation with the same price set at a 30% discount as an *internal offer*, so it isn't available in the storefronts directly, and then use this variation for cross-selling or upselling from other products. Like upsell your ebook at a huge discount when prospects add your online video course to their shopping basket.
- "Copy" really should be titled "Duplicate", because it creates a new base product based on the current settings.
- "More" hides the product deletion link, and "Custom Attributes", which is a custom key–value dictionary tied to the current product variant. This could be useful for API calls and other automation tasks.

You see, FastSpring is very flexible when it comes to designing product variants. With the "Checkout Links" action, you can preview most things in your store quickly – once you have a storefront set up, that is, which we haven't. We'll get there right after we finish the product setup.

I played around with offers and product variants for a while and would like to share my experience so far to give you some perspective.

Product variants can be used for marketing campaigns

Product variants at a discount are useful for campaigns like newsletter or blog sponsorships.

You can share the direct link to the variant with a selected group of people while everyone else will see the default price. You can also customize the variant's description and title to reflect this is a special offer for this special audience, only.

Little things like that can make the checkout experience feel more personal. In more technical terms, you could say this is a potential means to heighten community loyalty when you make them feel special and acknowledge their being in-group.

Coupon codes offer a similar device: you can add an "Applied Discount Reason" that's displayed right below the discounted price and is a good opportunity to say "Hi" to readers of a blog.

On top, a discounted variant's price reduction can also be time limited: then the discount is good for the duration of your campaign, and then the price goes up again to the default value. You don't need to delete the product variant in this case, and thus will not break the precious incoming links.

Cross-sell, upsell, and app bundles

The cross-sell and upsell options are nice to offer incentives like "buy my book at 50% off when you buy my video course". I think that these options shine when you have different types of products in your store that people might not shop for together. They are also useful when you have a very broad roster of products, for example when you want to recommend a particular pair of gloves that goes well with a merino wool scarf in your custom-tailored clothes shop.

For my apps, I found bundles to be more useful than the cross-sell or upsell, though.

Here's an example: I want to offer my apps TableFlip and WordCounter together at a discount. Say each costs $15; then buying both should cost $25 instead of the regular $30. Using upsells for this, I'd have to create a $10 variant for both apps to lower the sum total, and then set up an offer for each product that uses the respective variant. This may work for two apps, but the possible combinations grow exponentially.

Bundles are a much better device to implement this. You can set a bundle price and lump together all the apps, effectively overriding their price. When the customer picks any app that is also available in the bundle, you can offer to "complete" the bundle and swap the single items from the shopping cart with the bundle product. Because this is so useful, we'll discuss the setup and implications of bundles in a dedicated section in the appendix.

If you want to see how bundles, product alternatives, upsell and cross-sell options look in a store, head to https://www.fastspringexamples.com/ and have a look at the "Advanced Selling Products" examples at the bottom of the left navigation sidebar.

Product fulfillments in detail

To complete the app setup in your store, we'll need to wire the license generator to create a new license for every purchase. Automatic actions on order completion are called "fulfillment".

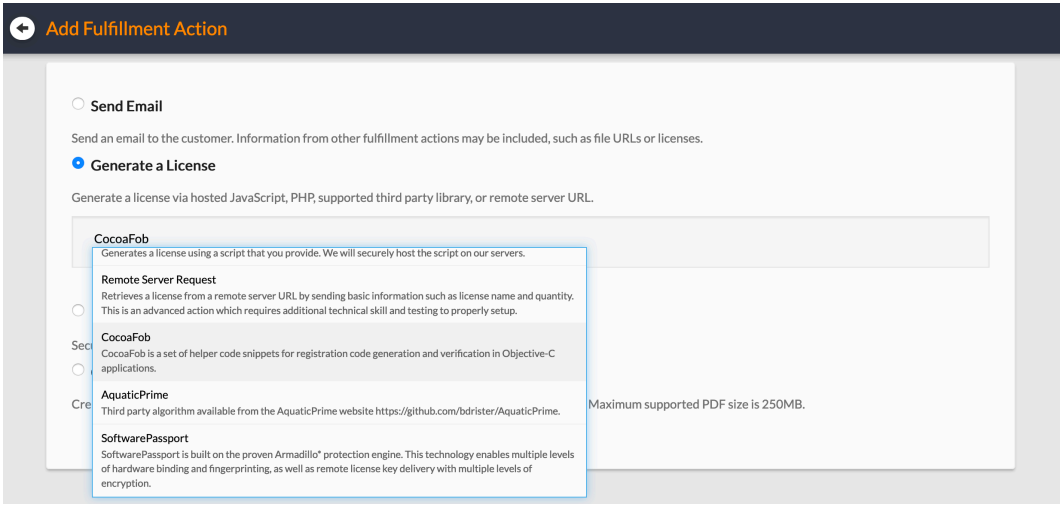
Have a closer look at the "Fulfillment" group on the product page. By default, the group displays "Digital Product/Service" in a label in its top-right corner, a "No Fulfillment" placeholder in the middle, and a couple of actions at the bottom.

"Add Fulfillment" and "Show Disabled Fulfillments" are self-explanatory, and we'll have a look at available fulfillments in a second. The third action, "Change Product Format", allows you to switch from digital to physical goods and enable shipment. The choice you make here affects the available fulfillment options. If you sell amazing swag for your brand like coffee mugs and t-shirts, or if you offer printed books, this might be worth a second look one day. For now, we'll stick to digital products without physical shipment.

When you pick "Add Fulfillment", you'll be presented with four options:

- "Send Email", which is useful to send a personalized message to customers. You can also include the license keys and other niceties like app activation links in these emails. We will have a look at this once the license generator is working.
- "Generate a License" is exactly what we're going to pick; but first have a look at the other options.
- "Provide a File Download" is meant for a different business model involving protected, time-limited downloads. Since we're locking our app's functionality behind a paywall that requires a license, it doesn't make sense to *also* limit the download. Quite the contrary: I suggest you make the app free to download and test-drive for a while instead to create more interest in your app. With the approach laid out in this book, this option is not very useful.
- "Create a Signed PDF" is not useful for our purposes, but you can pick this option if you sell ebooks in your store. You can have the customer name printed on every page of the PDF, which is supposed to prevent file sharing to some extent. I do like that this feature exists, but I doubt it'll really protect your content from being pirated.

Configure License Code Generation via CocoaFob

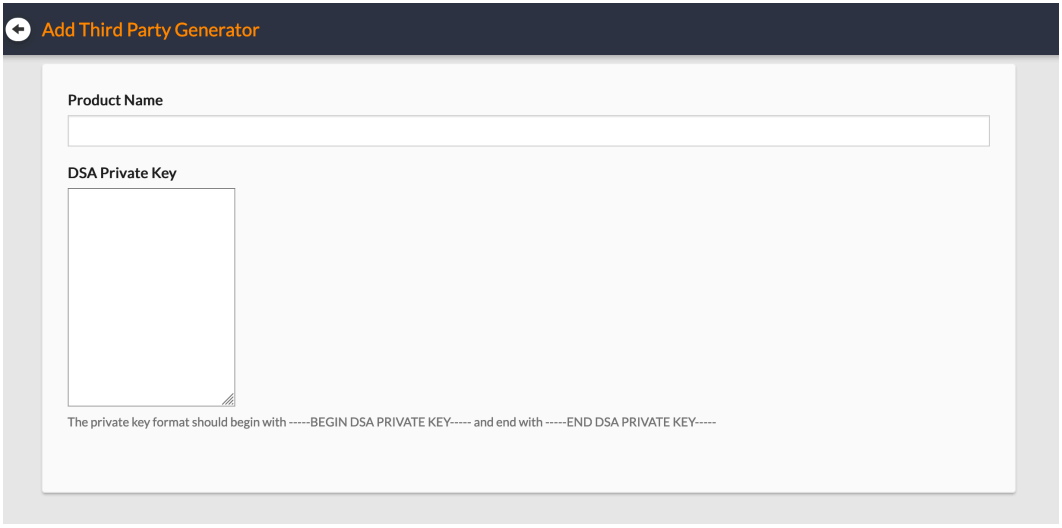


Choose CocoaFob-based license generation from the fulfillment options

In case you haven't followed the fulfillment details so far, click on the "Add Fulfillment Action" button in the top group of your app product listing screen. There, pick "Generate a License", and then choose "CocoaFob" from the list of license generators.[7] Hit the "Next" button to set up the generator.

In the setup form, pick a "Product Name" for CocoaFob to use for the license generator when encoding the license details into a license code. This could be the same as the product path in FastSpring, e.g. "my-new-app", a numeric product identifier, or any other unique string.

[7] Yeah, I know the description text still says "Objective-C", but it really is more versatile than that. I personally take care of the Swift port to work as expected, since I rely on the library in my own apps, too. Skin in the game.



The license generation setup form

You will be using the same string in your app later to decode a license code and then verify its contents. For example, if I decode a license in my Mac app TableFlip and the resulting product name turns out to be "duke-nukem-forever" or something, I don't want the license to pass even though I was able to decode it successfully. Don't worry, though, this example is totally made up. In practice, a mixup like this shouldn't be possible because the keys used during license code creation will be different for each app and the decoding step should fail. You cannot unlock app A with a code for app B since the apps supposedly use different keys during the encoding step.

What is a good product name for a license code generator and it's counterpart, the license code verifier in your app? After all, you set up all this to prevent software piracy to some extent. Thinking about potential hackers of your verification code, they might indeed an easier time scanning app for a string that corresponds to the app's name, like "`mynewapp`". A random product name like "`/g}eJ3U8bG29r+wfcZ;`" will be much harder to guess, and then locate in a hex dump of the application binary. I think this is a bit overkill, though. The potential security upside is negligible: if someone has the means to perform these kinds of actions to crack your code, you likely won't hinder them much by obfuscating the CocoaFob product name string. As mortal human beings, we app developers prefer sensible strings naturally, and I think it's okay to roll with this impulse. It's much easier to detect the typos in "mnyweapp" than it is in a nonsensical string of random characters.

But please, do consider both approaches carefully. The obfuscation may appeal to you for good reason. Then again, with every action you take on the copy protection front, you also send a message to yourself and reinforce your beliefs. Do you want this message to say "I am very freaked out about my things being stolen" and "the world is dangerous", or rather one along the lines of "people should be incentivized to pay for quality products" and "the world deserves an app as good as mine" and stop there? – I personally want to spread my stuff. But I cannot afford to do it for free, or I'll starve. I'm not worried about piracy. Folks who really use my apps professionally will pay anyway. It seems people are inclined to hunt for paid apps and sustainable businesses instead of free apps because they are worried about the continued development and support of tools they rely on. This is true for professionals looking for better tools; less so for gamers, I would think. I also don't create freeware applications most of the time and try to charge even a small amount not because I'm greedy, but because one of my core beliefs is that nobody can afford not to pay for professional app development in the long run. At least not in the world we inhabit, where you need to work to make money to survive. This is just another opportunity to do some soul-searching, get clarity about what you value, and be honest with yourself.

With a product name set, the "DSA Private Key" finally comes next.

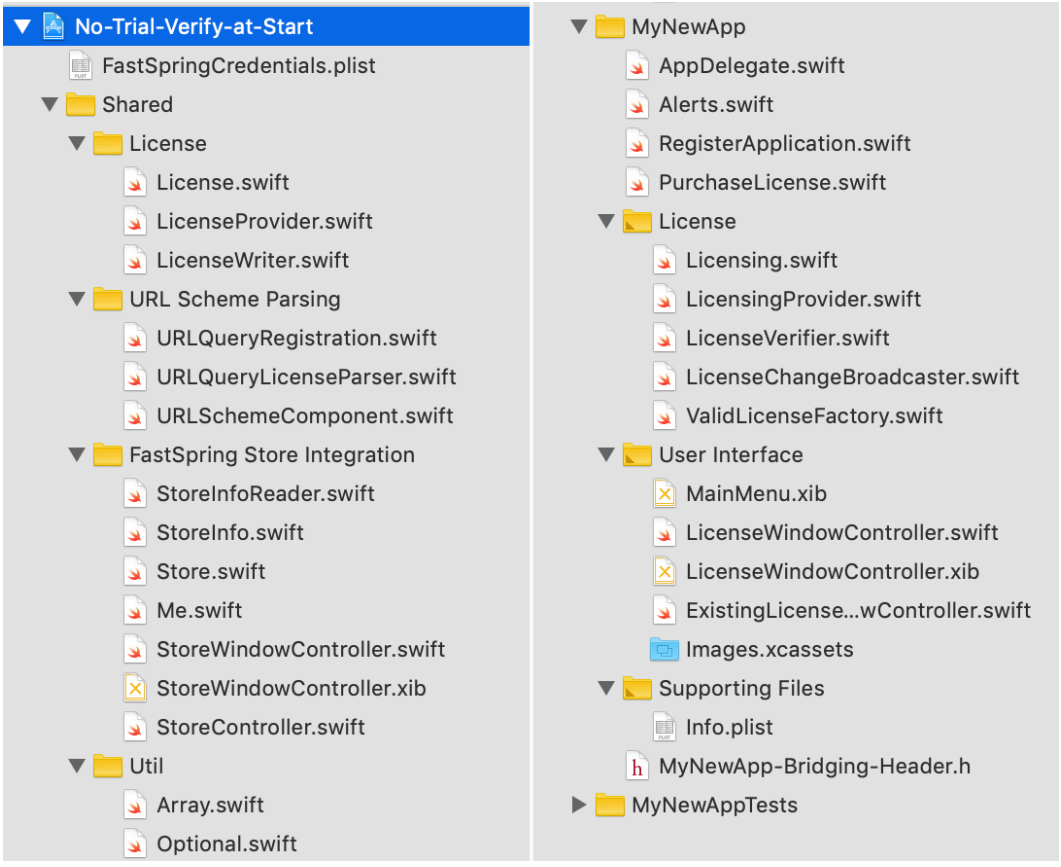
Heads Up: This Is an Excerpt for the Book Sample!

Only a selection of the full e-book are available in this chapter. For the book sample, I picked some of the most interesting things so you get a feeling what is covered in the book. Keep in mind that some parts might not make sense on their own because context is missing from the samples.

In the full book, you'll learn how to:

- fully customize your FastSpring store,
- offer additional in-app purchases,
- support app activation via e-mail,
- use your web server for license activation,
- use the embedded in-app store of FastSpring to buy license codes,
- and much more!

Sample App 1: How to Verify License Codes in Your App



Project organization of the demo project

According to the license template I recommended earlier, `#{product},#{name}` produces a license code that is just an encrypted form of, for example, "`MyNewApp,Christian Tietze`".

A license code is invalid when checked against that template if it satisfies at least one of the following criteria:

- The license code is not a string that was encrypted by a private key which matches the public key used by the app for decrypting.
- Parsing the decrypted string doesn't meet expectations because the expected product name doesn't match or is missing, or
- because the extracted licensee name doesn't match what the user entered.

As we've discussed, the algorithm can be adjusted a fair bit. You can try different keys to decrypt; and you can try different license templates upon failure to support both personalized and non-personalized license codes.

We need to utilize the CocoaFob algorithm to decrypt a code and then create an abstraction of all the info we need in the application so we have something to work with. This can look like the types I sketched a few chapters ago. Missing from the short segue into modeling a state machine's state back then was the implementation of services that perform the work and react to changes. This chapter is dedicated to implement all that, but without the additional work of implementing a trial mode. You will learn what the essence of locking functionality away behind license code paywalls means. In the next chapter, we will build upon all this and add a time-based trial.

You can find a fully functional sample app in the book's code repository. It's inside the aptly named `No-Trial-Verify-at-Start` folder.[8] Should any direct link ever break, head over to my website which should be around for as long as the internet and I stay alive in one way or another: `https://christiantietze.de/`

Embedding CocoaFob

Since Swift 2, CocoaFob provides native implementations. We're all set for Swift 5 and will be prepared for Swift 6 once it hits, too.

If you require instructions about how to set up the lib for Objective-C, please refer to the appendix.

[8] https://github.com/CleanCocoa/mac-licensing-fastspring-cocoafob/tree/master/No-Trial-Verify-at-Start

The original repository is located at `https://github.com/glebd/cocoafob`.

You can consume the code in your app in different ways. To reduce the amount of build tools you need to follow along, I'll stick to a manual checkout with git submodule. We'll do that in a minute.

If you have CocoaPods, you can instead add an entry to your `Podfile`. Since CocoaFob is not registered in the CocoaPods base repository, refer to its GitHub location directly:

```
use_frameworks!

source 'https://github.com/CocoaPods/Specs.git'
platform :osx, '10.10'

pod 'CocoaFob',
     :git => 'https://github.com/glebd/cocoafob',
     :branch => 'master',
     :inhibit_warnings => true
```

If you use Carthage (like I do for most of my macOS projects), then this will be enough:

```
github "glebd/cocoafob" "master"
```

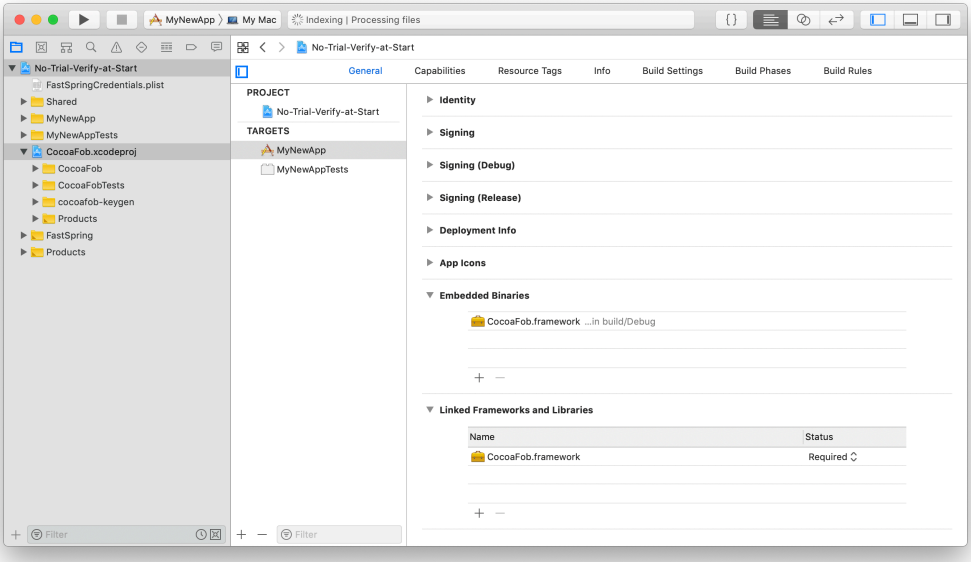
The sample code[9] uses git submodules directly. I tend to group all external dependencies into an "External" folder according to a convention I picked up years ago. You might be used to put external dependencies into "Vendor". We'll be using my old convention for the sample projects because … that's the way I set them up initially back in 2015, duh.

From the project root of your project, these are the commands you'd have to enter at the command line to checkout cocoafob:

[9] https://github.com/CleanCocoa/mac-licensing-fastspring-cocoafob

```
$ mkdir External
$ cd External/
$ git submodule add git://github.com/glebd/cocoafob.git
```

In the dark ages, you had to add a couple of files from the Swift source code folder manually to your project, thus exposing the license verification code to the whole app. Nowadays, all the cool kids use proper Swift modules for their libraries, and so shall we.



The CocoaFob project is part of the app project, and the resulting Framework is embedded and linked against from the app target

The Swift 5 version of the framework is located in the `External/cocoafob/swift5/` subfolder. There is an Xcode project that you can use to build the framework, and then embed the build product in your app.

If you don't know how this works, these are the steps I recommend to add CocoaFob to your project in a reliable way:

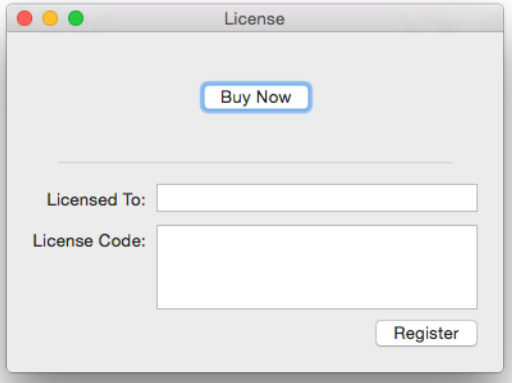
- Drag & drop the `CocoaFob.xcodeproj` project file into your app's Xcode project navigator. (I suggest you drop it in the project root somewhere below the

production code and test group.) This will not yet create a dependency. You need to link against the framework product of the newly added subproject, and embed the framework inside the app.

- Click on your app project root item in Xcode,
- select the app target in the project details view,
- then select the "General" tab to show the app target's settings.
- At the bottom you have the "Embedded Binaries" and "Linked Frameworks and Libraries" sections. Click on the "+" button of "Embedded Binaries" and select any of the `CocoaFob.framework` entries the displayed dialog will offer. If you've never done this before, you'll notice how adding the framework to "Embedded Binaries" has also affected the "Linked Frameworks and Libraries" section. It won't work the other way around, though: linking does not imply embedding, but embedding implies linking, as far as Xcode is concerned.
- As a side effect, this will also make Xcode build the CocoaFob framework when needed when building the app. You can verify this by heading to the "Build Phases" tab of the project navigator and then check the topmost "Target Dependencies" section, which will now list `CocoaFob.framework`.

Now you're all set. For reference, you can have a look at the sample apps from the book's accompanying source code material. They use the same approach.

User Interface to Enter License Information



The "Enter a License" window of New App.

With the `LicenseVerifier` service object we can test if license information is valid, and `RegisterApplication` executes the appropriate command. Your application will need to provide a form for the user to provide this information. This section focuses on how to wire UI components to do the job.

First, where do you place the form?

In most apps, I've discovered a "Register APPNAME ..." or "License APPNAME ..." main menu item somewhere. Sometimes it's buried in the Help menu, sometimes it's below the menu item for preferences in the submenu with the app's name. Dockless apps that only add an icon to the menu bar have this, too. It's typically, but not always, located in a contextual menu that opens under the menu bar icon.

You can also have a preference pane. Most apps have some sort of preferences window anyway, so adding a pane dedicated to entering license information can make sense. You can, and I think *should*, combine this with a main menu item, even though the preferences already have a standard menu item. The main menu offers full-text search via the help menu and thus is super accessible.

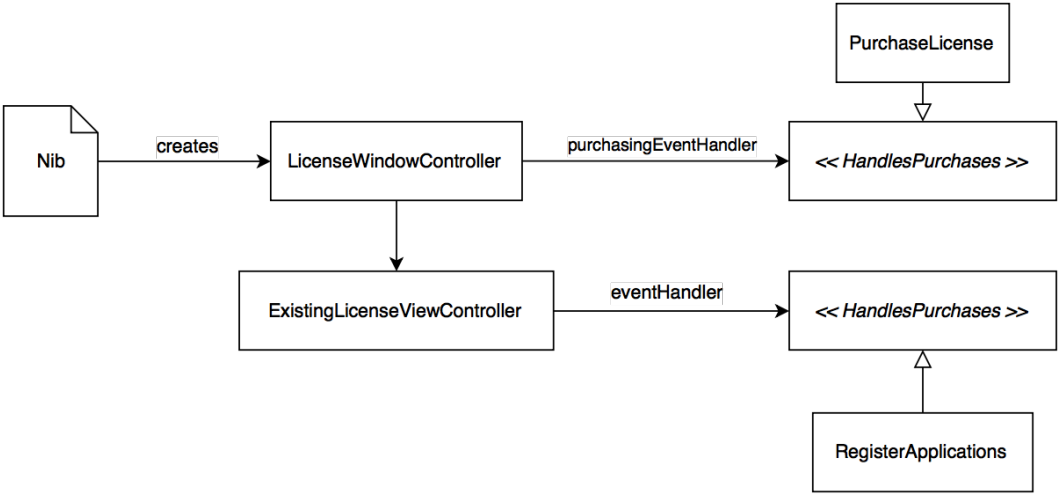
To maximize flexibility in case you copy and paste the interface implementation to

your app, I decided to roll with a dedicated license window for this book's sample app. You can copy it as is and display it as a dialog window or as a window-modal sheet, or you only take the content view and paste it into a preference view controller.

Creating a window like the one pictured above is very easy in Xcode's Interface Builder, so I won't bore you with the details of adding labels and text fields to a view.

When you set the interface up, you could, in theory, use Cocoa Bindings to populate the form fields with values from `UserDefaults` directly. I suggest you don't. Changes to the text fields would be stored immediately while you type. That renders the "Register" button useless, because no matter if you press it or not, the text fields's contents are saved immediately. That's weird, to say the least. You could implement validation of live user input, but I cannot imagine a straight-forward solution that doesn't bother the user while they enter their name. Also, you don't want accidental key presses to immediately persist the now invalid license information. I think it's not worth the hassle; don't use Cocoa Bindings here and validate input on button press instead. Only store valid license information via `RegisterApplication`, which we already have implemented.

Window and event handler architecture



Class diagram of UI Components and their event handlers

I tried to both maximize reusability and show useful macOS architectural patterns at the same time here. The basic message is:

- treat your window controllers and view controllers as view components, and make them as dumb and short as possible,
- separate window controller from sub-view controllers where possible, and
- put your windows in separate Nibs or Storyboards instead of sticking them in the `MainMenu` that comes with the project template.

For pragmatic reasons, it can make sense to not have both a `NSWindowController` and a `NSViewController`. If your window has a single user interface element, then why bother? These heuristics have proven to be useful in the community and during my practice. Apply them and deviate from them with reason.

The `LicenseWindowController` object is the main entry point and created through the Nib. This could nowadays have been a Storyboard just as well. The point is to extract it into its own Interface Builder file: it makes you suffer less when the files get corrupted or Xcode doesn't behave anymore and you need to recreate the UI, and it reduces merge conflicts in team projects. The window controller handles "Buy Now" button presses directly and delegates to a `purchaseEventHandler`, a service that conforms to the `HandlesPurchases` protocol. I didn't extract a `PurchaseViewController` to handle this event because the overhead isn't worth the design purity in my book. We'll have a look at an implementation of the `HandlesPurchases` type later.

An `ExistingLicenseViewController` then actually handles form input events and reacts to the "Register" action. It's owned by `LicenseWindowController`. It delegates license verification through its `HandlesRegistering` protocol to the `RegisterApplication` service that we will make conform to it.

View controller implementation

We focus on `ExistingLicenseViewController` here to have a closer look at user input handling.

```
protocol HandlesRegistering: class {
    func register(name: String, licenseCode: String)
}

class ExistingLicenseViewController: NSViewController {
    // MARK: UI Components

    @IBOutlet weak var licenseeTextField: NSTextField!
    @IBOutlet weak var licenseCodeTextField: NSTextField!
    @IBOutlet weak var registerButton: NSButton!

    // MARK: Action handler

    var eventHandler: HandlesRegistering?

    @IBAction func register(sender: AnyObject) {
        let name = licenseeTextField.stringValue
        let licenseCode = licenseCodeTextField.stringValue
        eventHandler?.register(name: name, licenseCode: licenseCode)
    }

    // MARK: Display Commands

    func displayEmptyForm() {
        licenseeTextField.stringValue = ""
        licenseCodeTextField.stringValue = ""
    }

    func displayLicense(license: License) {
        licenseeTextField.stringValue = license.name
        licenseCodeTextField.stringValue = license.key
    }
}
```

This, again, should appear to be a very simple implementation. This is what view controllers ought to do: react to user input, and delegate to other objects for more meaty actions as soon as possible. In your applications, you will find that method after method is added to view controllers over time to implement the various delegate methods already. You don't want to have more in the view controllers than absolutely

necessary.

As a benefit, this kind of approach to programming view controllers makes them easily testable. Have a look at the tests in the sample app code to see how I test the Interface Builder connections and the action-handler's entry points. Here's just a couple:

```
class ExistingLicenseViewControllerTests: XCTestCase {
    var controller: ExistingLicenseViewController!

    override func setUp() {
        // Load the window controller from its nib ...
        let windowController = LicenseWindowController()
        _ = windowController.window
        // ... and then get the initialized controller:
        controller = windowController.existingLicenseViewController
    }

    func testLicenseeTextField_IsConnected() {
        XCTAssertNotNil(controller.licenseeTextField)
    }

    func testLicenseCodeTextField_IsConnected() {
        XCTAssertNotNil(controller.licenseCodeTextField)
    }

    // ...

    func testDisplayEmptyForm_EmptiesTextFields() {
        controller.licenseeTextField.stringValue = "something"
        controller.licenseCodeTextField.stringValue = "something"

        controller.displayEmptyForm()

        XCTAssertEqual(controller.licenseeTextField.stringValue, "")
        XCTAssertEqual(controller.licenseCodeTextField.stringValue, "")
    }

    func testDisplayLicense_FillsLicenseTextFields() {
        let license = License(name: "a name", licenseCode: "a code")
```

```
34          controller.licenseeTextField.stringValue = ""
35
36          controller.display(license: license)
37
38          XCTAssertEqual(controller.licenseCodeTextField.stringValue, "a code")
39      }
40  }
```

In the last section, you might wonder why we didn't start writing the registration code into the "Register" button's action handler, directly into the view controller. Having seen the implementation of the view controller, this might still be a puzzle. It's because encapsulating this simple piece of logic telegraphs to any reader of your code that this is a cohesive sequence. To put it into a function with a name, or a type, makes the sequence a *thing*. It's not just a concept in your head, but a thing with a name in the realm of your code. All this is the foundation of object-oriented programming, of course: to encapsulate things that belong together in objects and then treat these as cohesive wholes. Here, having a service object at hand makes form field validation a trivial matter. That's nice.

React to license change events to unlock functionality

You know `RegisterApplication` already. Adding conformance to `HandlesRegistering` is a one-liner because the method signatures match up already.

To have the app do something interesting, I've added sending a notification when the registering succeeds:

```
1   class RegisterApplication: HandlesRegistering {
2       // ...
3
4       // Newly added dependency:
5       let changeBroadcaster: LicenseChangeBroadcaster
6
7       func register(name: String, licenseCode: String) {
8           guard let license = licenseFactory.license(
9                   name: name,
10                  licenseCode: licenseCode) else {
11              displayLicenseCodeError()
```

```
            return
        }

        licenseWriter.store(license)
        // This call is new:
        changeBroadcaster.broadcast(.registered(license))
    }
    // ...
}
```

The broadcaster is a very simple object that wraps a call to `NotificationCenter.post` in `broadcast`:

```
class LicenseChangeBroadcaster {
    let notificationCenter: NotificationCenter

    public init(notificationCenter: NotificationCenter = .default) {
        self.notificationCenter = notificationCenter
    }

    public func broadcast(_ licensing: Licensing) {
        notificationCenter.post(
            name: Licensing.licenseChangedNotification,
            object: self,
            userInfo: licensing.userInfo())
    }
}
```

This makes it easy to write test doubles or mocks to verify the call to `LicenseChangeBroadcaster.broadcast` with the proper `Licensing` object, instead of having to verify that an `NSNotifications` is being passed on, and inspecting what it looks like, including the `userInfo` dictionary. A test double to verify calls of `broadcast` is much simpler to write.

Because the service sends a notification, other components can loosely couple to the success of registering the app. In the sample application, we do not allow using the main application interface until the app is unlocked. `AppDelegate` is the gatekeeper here, but you may subscribe different components separately, and put the main

unlock functionality anywhere else. The `AppDelegate` in the sample app subscribes to the `licenseChangedNotification` and locks or unlocks the main app accordingly, like this:

```
@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {
    func applicationDidFinishLaunching(_ aNotification: Notification) {
        observeLicenseChanges()
        // Unlocking upon app launch, either by reading from UserDefaults
        // and processing the value directly, or by going
        // through `RegisterApplication`, which will trigger the
        // callback for you.
    }

    private func observeLicenseChanges() {
        NotificationCenter.default.addObserver(
            self,
            selector: #selector(AppDelegate.licenseDidChange(_:)),
            name: Licensing.licenseChangedNotification,
            object: nil)
    }

    @objc func licenseDidChange(_ notification: Notification) {
        guard let userInfo = notification.userInfo,
            let licensing = Licensing.fromUserInfo(userInfo)
            else { return }

        switch licensing {
        case .registered(_):
            unlockApp()

        case .unregistered:
            // If you support un-registering, or use this callback
            // to figure out the initial state, lock the app again here:
            lockApp()
        }
    }

    private func lockApp() { ... }
```

```
36      private func unlockApp() { ... }
37  }
```

Now that's a demonstration of a monolithic license-based lock. You could also lock certain features, and have specific components lock/unlock themselves based on license change events. That's a good idea if your `AppDelegate` grows unwieldy already. Keep this in mind if you think about implementing in-app purchases of any kind. That amounts to the same: features are locked/unlocked separately.

In the sample app, I added code to read license information from `UserDefaults` and set up the initial state accordingly. In most of the apps I shipped so far, I transitioned to a different approach. During the app launch sequence, I make the existing services send a `licenseChangedNotification`, too. So the app launch and a user-initiated change event are treated the same. The loose coupling helps with that since the notification subscriptions have to be written anyway, and I don't need to establish a new route just for the launch sequence. I start with locked components at first, and since the app launch triggers a synchronous notification dispatch on the main thread, the components will unlock instantly. There's no visible flicker or anything.

I suggest you strive for the same: unify the locking and unlocking wherever possible.

For the sake of experimenting and learning how to treat the events, I found it immensely useful to have two different entry points to feature unlocking at first, though. There's not much to learn, so once you get the hang of it and know that things work as expected, you can easily unify the processes.

With the `AppDelegate` responding to license change events, and the view controller–event handler architecture from above, we can now display the license info UI, register the app, and unlock features without tight coupling.

In fact, you can put the exact code from the sample into your own application, change a few parameters, and then write appropriate locking and unlocking logic. That's all there is in terms of the most basic copy protection.

Heads Up: This Is an Excerpt for the Book Sample!

Only a selection of the full e-book are available in this chapter. For the book sample, I picked some of the most interesting things so you get a feeling what is covered in the book. Keep in mind that some parts might not make sense on their own because context is missing from the samples.

In the full book, you'll learn how to:

- fully customize your FastSpring store,
- offer additional in-app purchases,
- support app activation via e-mail,
- use your web server for license activation,
- use the embedded in-app store of FastSpring to buy license codes,
- and much more!

Sample App 2: How to Add a Time-Based Trial

In the sample app we've developed in the previous chapter, registering and unregistering now is fully functional. You can put your app online, and nobody can access the license-protected features without having valid license information.

The thing is that test-driving apps is very important in practice when you target customers directly. It is less important if you target businesses, pitch their IT department, and have them install the app on devices on behalf of users. Business-to-customer indie shops will fare better with a demo to help prospects decide if they want to buy the app. You can also frame it in a less appealing way: when you already have a foot in the door and manage to make your app part of a users workflows, they are more inclined to become customers. For a price of US$2, your business model can be paid-up-front and appealing to impulse buyers. If your app costs US$50, people will be far less likely to buy without test-driving the app. I don't have numbers to back this claim. This advice is solely based on anecdotes. Search your feelings, and find out where the tipping point from "might buy" to "definitely not without testing" is. Not being able to test iOS apps was a serious source of complaints for ages. Mac developers could still provide a demo on their website for direct download, even though the Mac App Store didn't support trials or demos. Now, developers and Apple have adapted and turn to a freemium pricing model, with a free download and premium features behind an in-app purchase paywall.

To help users decide if they want to buy, time-based trials are one battle-tested approach: You offer new users a test drive for a limited time so they can decide if they want to keep using the product. Your trial period should be long enough to make your app part of your user's lives or they will shrug the purchase dialog off too easily. For *The Archive*, a note-taking app, we've settled for a generous 60-day trial period to help newcomers implement the tool into their lives. For the *WordCounter*, a menu-bar app that tracks how many words you type, I figured 14 days would suffice to massage users into the beat of the app. Ask around and talk to your users to find out what works best for you.

Time-based trials are the de facto standard to limit functionality of what was used to be called "shareware" in the 1990s. Next to time-based trials, it's also popular to offer feature-limited trials. There, you offer a demo version with limited capability from the start but potentially running forever. Users have to pay to unlock the full potential of the app. We won't cover that here, because it'll turn out that this is essentially an in-app purchase. When you learn how to implement in-app purchases to unlock features in the appendix, you can create feature-based trials, too.

To add a time-based trial to the sample app, we will end up with a couple of changes that all revolve around the primary state model, `Licensing`, to look more like this:

```
enum Licensing {
    case registered(License)
    case trial(TrialPeriod)
    case trialExpired
}
```

After I experimented with different `TrialPeriod` implementations, I settled for a value type with both start and end date. To use a start date and a duration instead of an end date came natural, but this was very clumsy to use in practice. To find out if the trial period is expired *right now* requires a helper to get the current time, then add the duration to the start date and compare it to the current time. Since most timer-based operations, you have to compute the absolute end date, too, I suggest dropping the duration in favor of a fixed expiration date. `TrialPeriod`, in its most basic form, then looks like this:

```
struct TrialPeriod {
    public let startDate: Date
    public let endDate: Date
}
```

I asked around for feedback on this approach to modeling the licensing state and got a lot of great feedback and suggestions, some of which is reflected in the current names of the types. I actually favor a different base type:[10]

[10]Thanks a ton to Josh Caswell. He suggested many different type names that flow way better, and from which I picked a couple, and also suggested the pattern matching implementation I really like below.

```
enum Licensing {
    case licensed(License)
    case trial(TrialPeriod)
    // `TrialPeriod.isExpired == true` replaces .unregistered
}
```

This variant communicates clearly that it's all about moving the app from trial to licensed state. The third state above makes it harder to infer the transition rules in regard to `.registered`. Instead of a third state, an expired trial is just a variant of the `.trial` case. The associated `TrialPeriod` object knows if the trial is expired.

With Swift's modern and very powerful pattern matching, you can then have very succinct `case` statements. This requires the addition of a pattern matching operator, `~=`, and an enum that represents the `isExpired` state of `TrialPeriod`. First, have a look at the resulting API:

```
func handleLicensingChange(licensing: Licensing) {
    switch licensing {
    case .trial(.valid):
        // ...
    case .trial(.expired):
        // ...
    case .registered(let license):
        // etc.
    }
}
```

This reads as if `TrialPeriod` itself was an enum. But it isn't. The `~=(pattern:value:)` operator definition below makes an inner enum available in `TrialPeriods` place:

```
extension TrialPeriod {
    // Enum that's used by the ~= operator
    enum Validity { case valid, expired }

    var validity: Validity {
        return self.isExpired ? .expired : .valid
    }

    // Current date falls outside the covered range
    var isExpired: Bool {
        return !(self.start...self.end ~= Date())
    }
}

func ~= (pattern: TrialPeriod.Validity,
         value: TrialPeriod) -> Bool {
    return pattern == value.validity
}
```

If you never worked with custom pattern matching, this might look very weird at first. Even though the `Licensing.trial` case does not have an associated enum value but a struct value object, the pattern matching operator allows us to write `.trial(.expired)` as a shortcut to reach into a `TrialPeriods` `validity` property nevertheless.

This is a very cool feat, but also very advanced Swift territory, so I ultimately dropped it from the sample code for this book. But if you understand the code and prefer it, you're welcome to change the implementation!

From here on, this chapter will teach you how to adjust the existing sample application to take time-based trials into account.

You can find a fully functional sample app incorporating a trial mode in the book's code repository. It's in the `Trial-Expire-While-Running` folder.[11] You can use it as a template to quick-start your development process and refer to this chapter for details where needed.

[11] https://github.com/CleanCocoa/mac-licensing-fastspring-cocoafob/tree/master/Trial-Expire-While-Running

Store and Read Trial Information

You have to store the `TrialPeriod` somewhere. We will start by storing the trial dates in the `UserDefaults` next to the license information – mostly so you can inspect the result easily from one place. Persisting a date for the sample app could produce these entries:

```
$ defaults read de.christiantietze.MyNewApp
{
    "trial_ending" = "2019-08-30 09:17:25 +0000";
    "trial_starting" = "2019-08-25 09:17:25 +0000";
}
```

If you want to make bypassing the time-based trial limitation harder than overwriting date information that's in plain sight, you have a couple of options.

First, you may try to encode the expiration date and put it into `UserDefaults`. But if users simply delete the related defaults, the app will likely start a new trial period, just as if it was launched for the first time.

Then you might try to find a secret hiding spot instead, like a file stored somewhere. With Sandboxed applications, savvy users will know that the trial information is very likely contained in the app's Group Container. That's not a good hiding spot.

Lastly, you might come to the conclusion that the `UserDefaults` are just as good a place to store the trial period unencoded as I do in the samples if your options are this limited, anyway. But you do have another option: you can use a different `UserDefaults` suite or domain. macOS apps don't have to use defaults with their bundle ID as the domain, like `de.christiantietze.MyNewApp` above. You can take advantage of this and store your trial info in a totally different defaults domain, like `com.google.project-infinity` or whatever.

On top of these offline solutions, you can require users to register on the web and store the trial duration on your server. This essentially results in your server dishing out a time-limited license. That's exactly how devs model this feature: you get one free time-limited license from the start but can purchase unlimited licenses anytime. Both actions result in the user's account being associated with specific

license information. We won't be looking at an online license verification and trial activation service in this edition of the book; that's way beyond the scope.

Remember not to become paranoid, but do weigh the cost and benefits carefully. For an app for US$5, it won't pay off to make it harder for cheap users to extend the trial or hack the license mechanism. Every hour you spend on copy protection against a handful of people is an hour you won't spend improving the app for paying customers. The cost/benefit-analysis can easily change when you start to sell premium software for a couple hundred dollars. Just try not to punish the wrong people.

Since we use the `UserDefaults` for license information storage already, I wrote `TrialProvider` and `TrialWriter` in a way that mimics the license counterparts. If you remember that code, this one's pretty close to a copy & paste job, changing some names and keys, really.

```
extension TrialPeriod {
    struct DefaultsKey: RawRepresentable {
        let rawValue: String

        init(rawValue: String) {
            self.rawValue = rawValue
        }

        static let startDate = DefaultsKey(rawValue: "trial_starting")
        static let endDate = DefaultsKey(rawValue: "trial_ending")
    }
}

// Convenience methods to store Date objects for TrialPeriod keys
extension Foundation.UserDefaults {
    func date(forTrialKey trialKey: TrialPeriod.DefaultsKey) -> Date? {
        return self.object(forKey: trialKey.rawValue) as? Date
    }

    func set(_ date: Date, forTrialKey trialKey: TrialPeriod.DefaultsKey) {
        self.set(date, forKey: trialKey.rawValue)
    }
}
```

```
24
25  class TrialProvider {
26      init() { }
27
28      lazy var userDefaults: Foundation.UserDefaults = .standard
29
30      var trialPeriod: TrialPeriod? {
31          guard let startDate = userDefaults.date(forTrialKey: .startDate),
32              let endDate = userDefaults.date(forTrialKey: .endDate)
33              else { return nil }
34
35          return TrialPeriod(startDate: startDate, endDate: endDate)
36      }
37  }
38
39  class TrialWriter {
40      init() { }
41
42      lazy var userDefaults: Foundation.UserDefaults = .standard
43
44      func store(trialPeriod: TrialPeriod) {
45          userDefaults.set(trialPeriod.startDate, forTrialKey: .startDate)
46          userDefaults.set(trialPeriod.endDate, forTrialKey: .endDate)
47      }
48  }
```

This is the backbone of the infrastructure that makes time-based trial period persistence real.

You know the `TrialPeriod` model type already, and now you also some services to read from and write to `UserDefaults`. With these important prerequisites taken care of, we can focus on adding actual functionality or logic to the app to react to trial expiration dates, and lock and unlock access to features.

Obtain and Work with Current Time

For the sample app, I want to work with a trial duration of 5 days. Not because I think that's the best duration you could use, but because it keeps date calculations pretty

simple. To create a `TrialPeriod` in the app, we need the current time as `startDate`, and then add the duration to calculate `endDate`.

In Foundation, the current time can be obtained by creating a `Date` object with the default initializer. But that's going to bite us during tests, where we want to have constant inputs and outputs, not dynamically changing values.

The go-to solution when you express your concepts as objects is to introduce a representation of a thing that tells the time. In other words, a clock:

```
protocol Clock {
    func now() -> Date
}
```

To get a real clock, we can rely on the default parameterless initializer of `Date`:

```
class SystemClock: Clock {
    init() { }

    func now() -> Date {
        return Date()
    }
}
```

And we can provide a clock substitute that always tells the same date and time. We will use that both in manual tests to rewind time and see how the trial-based functionality reacts, and in unit tests.

```
class StaticClock: Clock {
    let date: Date

    init(date: Date) {
        self.date = date
    }

    func now() -> Date {
        return date
    }
}
```

With this, we can add a convenience initializer to `TrialPeriod` that takes a `Clock` to tell the current time, then adds the trial duration to compute the expiration date. I also want to introduce a type that expresses the duration of a day in terms of the app, and call it `Days`. With daylight savings time and leap seconds and what not affecting the length of a calendar day, the following is not a good representation for accurate calculations in calendar app – but it's good enough to express how long a trial should be. And, as you will see in the code, you can attach all sorts of useful behavior to it that a mere `Int` would not be suited to.

```
struct Days {
    let amount: Double

    init(_ anAmount: Double) {
        self.amount = anAmount
    }

    // Compatible with Date calculations
    var timeInterval: TimeInterval {
        return self.amount * 60 * 60 * 24
    }
}

extension TrialPeriod {
    public init(numberOfDays duration: Days, clock: KnowsTimeAndDate) {
        let startDate = clock.now()
        let endDate = startDate.addingTimeInterval(duration.timeInterval)
        self.init(startDate: startDate, endDate: endDate)
    }
}
```

It's very easy now to verify that `TrialPeriod(numberOfDays:clock:)` actually adds the appropriate time interval to the current time in tests:

```
class TrialPeriodTests: XCTestCase {
    class TestClock: Clock {
        var testDate: Date!
        func now() -> Date {
            return testDate
        }
    }

    func testCreation_WithClock_AddsDaysToCurrentTime() {

        let date = Date(timeIntervalSinceReferenceDate: 9999)
        let clockDouble = TestClock()
        clockDouble.testDate = date
        let duration = Days(10)

        let trialPeriod = TrialPeriod(numberOfDays: duration, clock: clockDoubl\
e)

        let expectedDate = date.addingTimeInterval(duration.timeInterval)
        XCTAssertEqual(trialPeriod.startDate, date)
        XCTAssertEqual(trialPeriod.endDate,   expectedDate)
    }
}
```

You see how easy it is to work with a `Clock` object to produce `Date` objects and provide test doubles. We can put additional behavior and logic in `TrialPeriod` to encapsulate trial expiration logic using clocks, too:

```
extension TrialPeriod {
    // Important piece of logic to tell if the trial is
    // still valid.
    func isExpired(clock: Clock) -> Bool {
        let now = clock.now()
        return endDate < now
    }

    // Convenient factory of remaining days, useful to
    // display the information in the app.
    func daysLeft(clock: Clock) -> Days {
```

```
12          let now = clock.now()
13          let remainingTime = now.timeIntervalSince(endDate)
14          return Days(timeInterval: remainingTime)
15      }
16  }
17
18  extension Days {
19      init(timeInterval: TimeInterval) {
20          self.init(fabs(timeInterval / 60 / 60 / 24))
21      }
22  }
```

As you can see, we do not interface with Foundation's `Date` directly but acquire dates exclusively through `Clock` objects. A parameterless `daysLeft` computed property would read nicer, but the method signature `daysLeft(clock:)` works really well in practice, too, once you get used to it. The upside of making the code testable is immense. Both `daysLeft(clock:)` and `isExpired(clock:)` would be a pain to test if you called `Date.init()` directly.

The main lesson we encounter with Dependency Injection of this kind is this: try not to create objects in your code, because their creation is hard to test, and instead defer object creation to dedicated service objects, like the Factory that `Clock` really is. This is especially important when it comes to types you do not own, like `Date` and similar Foundation types. It pays off to not worry about the unknown implementation details and introduction of subtle bugs because some of your assumptions turn out to be wrong.

Heads Up: This Is an Excerpt for the Book Sample!

Only a selection of the full e-book are available in this chapter. For the book sample, I picked some of the most interesting things so you get a feeling what is covered in the book. Keep in mind that some parts might not make sense on their own because context is missing from the samples.

In the full book, you'll learn how to:

- fully customize your FastSpring store,
- offer additional in-app purchases,
- support app activation via e-mail,
- use your web server for license activation,
- use the embedded in-app store of FastSpring to buy license codes,
- and much more!

Appendix

There are some things which don't naturally fit into the other chapters of this little book. And some which would fit, but detract from the chapter's message. I collected relevant but optional information here, so you can focus on the core licensing functionality and store setup in the main chapters of the book, then pick whatever feature you're interested in afterwards.

Self-Hosted License Generator

If you participate in a bundle sale that you don't host yourself, you cannot rely on the fulfillment actions offered by FastSpring. That means you need to generate licenses for bundle customers using your own license generator script. Here's how.

With all the work you have to go through to implement a license generator on your server, why not switch your FastSpring setup over to your self-hosted service, too? Granted, there's no upside to doing so if you're doing offline license activation and the CocoaFob algorithm works well for you. That's why I'll treat this as a guided exercise to better grasp what's involved in deploying your own licensing server. In the next sections, we'll have a look at in-app purchases and come back to the topic of customizing license generators, so having the background knowledge will come in handy.

This step is not mandatory at all, but I encourage you to experiment with FastSpring's license generator requests to understand what's going on. This way, you can eventually replace CocoaFob with your own license generator scheme, one that's shorter or encodes more information.

You can host the CocoaFob license generator on any server that runs PHP. The GitHub repository sports license verification and generator scripts. Upload these to your server so you can access them from your generator script. You can find the PHP scripts here: https://github.com/glebd/cocoafob/tree/master/php

Let's say your license generator will be available at `https://api.example.com/license/generato`
For the sake of this section, I'll assume we'll be looking at a file in `/license/generator/index.php` as the implementation of the endpoint to accept requests.

Upload and configure CocoaFob's PHP implementation

Most of CocoaFob's PHP files are example endpoints or dependencies. We will need these:

- `License/base32.php`, a Base32 encoding implementation;
- `License/license_generator.php`, a template for the license generator.

You can put those two PHP files in the same sub-directory as your endpoint, at `/license/generator`, and then create `.htaccess` rules to prevent direct access to them from the web. Or you put the CocoaFob files into a directory outside your htdocs and include them from there. The latter is recommended, but not possible on all hosting services.

You will need to upload the same private key you used for FastSpring to your web server, too. Before you do anything else, make sure you'll prevent access to this file. You don't want to have it surface on the web *at all*. Put it outside your public htdocs and restrict access. If your hosting provider does not allow this, ask them to make an exception. If everything else fails, `chmod` the file permissions and put it next to the PHP scripts.

A comment in the CocoaFob PHP file actually suggests to use a database instead to store and fetch the keys. If your web hosting service provides database access, this is actually a very good idea. You can encrypt databases easily and you cannot screw access up like you can screw up file read permissions.

Provided you have a way to obtain the private key as a string, by reading it from disk or database or from wherever, we can now tweak the license generator.

Have a look at `license_generator.php`: the private and public keys are used there in the constructor. We will not need the public key, so you can delete the statement that loads the public key, the field on the `License_Generator` class, and the `verify_license` method. You'll be left with the `make_license` method implementation and

the private key field. Change `make_license` to suit your license template. The default encodes product name, license name, and email. I usually drop the email parameter here, and if you followed along with the FastSpring setup, you'll only be needing the product name and license name, too.

The license template is represented in the `$stringData` variable that is constructed at the top of `make_license`'s method body. Adjust that statement to read `$stringData = $product_code.",".$name;`, or whatever template you are using.

If you were to remove all the unnecessary parts, the whole license generator can be represented in a simple function:

```
include('base32.php');

function generate_license(
    $private_key,
    $product_code,
    $name)
{
    ## ADJUST TEMPLATE ###################
    $stringData = $product_code.",".$name;
    ######################################

    $binary_signature = "";
    openssl_sign(
        $stringData,
        $binary_signature,
        $private_key,
        OPENSSL_ALGO_DSS1);
    $encoded = base32_encode($binary_signature);

    // Replace O with 8 and I with 9
    $replacement = str_replace("I", "9", $encoded);
    $replacement = str_replace("O", "8", $replacement);

    // Remove padding
    $padding = trim(str_replace("=", "", $replacement));
    $dashed = rtrim(chunk_split($padding, 5, "-"));
    $theKey = substr($dashed, 0 , strlen($dashed) -1);

```

```
29      return $theKey;
30  }
```

That's all there is to generating CocoaFob license codes! You might use this function instead, if you want.

Use the license generator in your endpoint

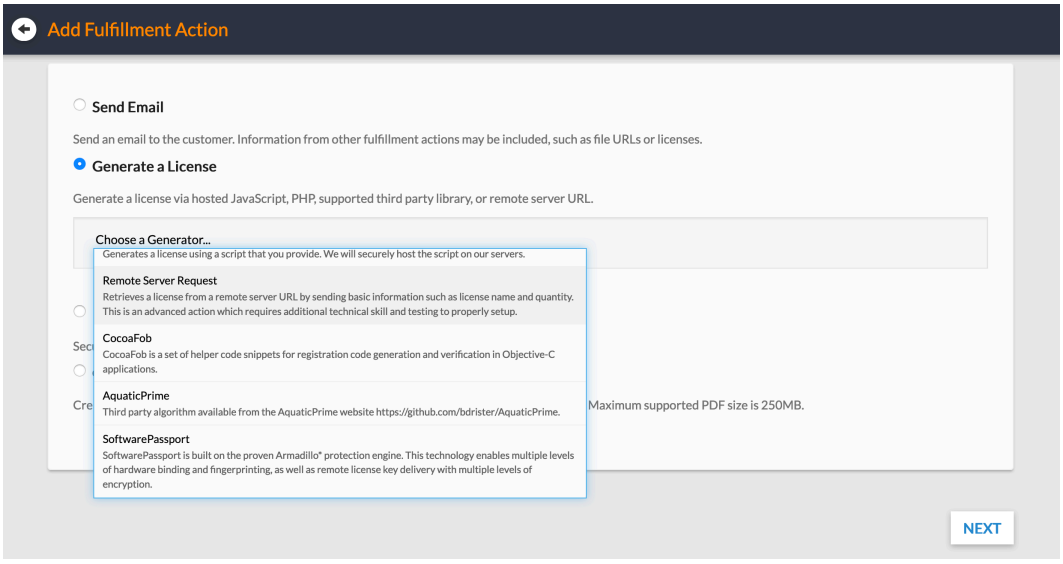
This is the data you have access to from the license generator request performed by FastSpring, represented as a PHP variable dump:

```
Array
(
    [company] => Licensee's Company
    [email] => licensee.email@example.com
    [internalProductName] => mynewapp_featureXYZ
    [name] => LICENSEE NAME
    [quantity] => 1
    [reference] => FASTSPRING_ORDER_REFERENCE_NO
    [subscriptionReference] => FASTSPRING_SUBSCRIPTION_REFERENCE_NO
    [security_request_hash] => xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
)
```

You could use `internalProductName` to customize the license generator setup if needed. So far, I have only hard-coded the product name as a parameter for CocoaFob's license generator and never used that request parameter.

If you want to keep a record of your customers, for example for subscriptions, online license activation, or other shenanigans, you'll want to create a database record with the `email` plus the order `reference` or `subscriptionReference` request parameters.

To set up the license generator for offline activation in the app, we are going to use the `name` parameter for the license name, and `quantity` to compute enough codes for the order.



Switching to a self-hosted license generator

When you switch to a self-hosted license generator in FastSprings product fulfillments, they provide a template with these instructions:

> Sort all parameters passed from our system to your system by parameter name (ordinal value sorting). Concatenate the value of each parameter into a single string. Ignore the parameter named "security_request_hash" when generating this string. Append the private key value to the end of the string and generate an MD5 hex digest on the final combined string. The resulting MD5 digest should be identical to the value passed in the parameter named "security_request_hash".

`security_request_hash` is passed in as a request parameter to your license generator to verify the integrity of the data. Computing this parameter is simple and can be done by malicious parties, too. But to pass the check, they would also need to know a private key. FastSpring shows the expected private key on the generator setup page. The private key is not part of the request. It's used by FastSpring and you to compute the hash only. That is enough to secure a request.

Here's the template sans instructions:

```php
<?php
ksort($_REQUEST);
$hashparam = 'security_request_hash';
$data = '';
$privatekey = 'PASTE FASTSPRING SECURITY TAB PRIVATE KEY HERE';

/* USE urldecode($val) IF YOUR SERVER DOES NOT AUTOMATICALLY */
foreach ($_REQUEST as $key => $val) {
  if ($key != $hashparam) { $data .= stripslashes($val); }
}

if (md5($data . $privatekey) != $_REQUEST[$hashparam]){
  return;  /* FAILED CHECK */
}

/* SUCCESS - YOUR SCRIPT GOES HERE */
```

The success case should result in passing one license code per line in the HTTP response body, via `echo`ing or `print`ing the codes.

You can add this code to the top of your endpoint implementation at `/license/generator/index.php` already.

With the request integrity validated, what else goes into the `index.php` file to use the generator? A modified version of an actual PHP script I used during a bundle sale a couple of years ago looks like the following:

```php
<?php
// ... request integrity check here ...

include_once('path/to/cocoafob/license_generator.php');
$name = $_REQUEST["name"];
$product = "mynewapp"; // Replace with your CocoaFob app name
$generator = new License_Generator;

$quantity = intval($_REQUEST["quantity"]);
if ($quantity == 0) {
    $quantity = 1;
}

```

```
14 for ($i = 0; $i < $quantity; $i++) {
15     $code = $generator->make_license($product, $name);
16     echo $code."\n";
17 }
```

This calls the license generator multiple times if the customer orders multiple licenses.

When you have uploaded and customized the `index.php` properly, you can run interactive test requests from the configuration page of the self-hosted license generator of FastSpring's product fulfillments. This is pretty simple, yet super useful to verify your setup is working without having to place test orders all the time.

Using a license generator for bundle sales

In the previous section, we've looked at our options to host product bundle sales on our own. Chances are you want to participate in bundle sales organized by other parties, like BundleHunt, MacHeist, HumbleBundle, and what have you. To work with these, you might need to change your generator setup a fair bit.

For example, in one bundle sale, I didn't have access to the customer's full name. The bundle organizers didn't collect user information apart from email and credit card, so I had not much choice: either use email instead of full name, or drop personalized licenses in favor of non-personalized license codes altogether. I didn't want to support both personalized and non-personalized license information at the time, so I rephrased the registration label in my app from reading "Name" to "Registered to" and went with the email-based approach. So far, nobody complained about this, so I like to think it worked out well.

Also, FastSpring's hash-and-key-based request verification could be too complicated. For another bundle sale, we set up HTTP BASIC AUTH instead, so the script didn't check the request again. For another sale, I shared a random string as the authentication token which they needed to pass as a request parameter. This doesn't provide much in terms of protecting integrity of the request, but it does prevent HTTP requests from the browser to end up dishing out license codes. You could guess the endpoint's URL, and maybe the `email` parameter, too, but the actual random string? Not very likely. Ask for your options, and prefer HTTP BASIC AUTH over shared

secret strings that are part of the request. FastSpring's approach above is pretty clever, so you might want to pitch that, too.

Here, I want to share a customized endpoint implementation with you that I used in one of the bundle sales, and that relies on a secret authentication token.

```
1  <?php
2  function isEmpty($var) {
3    return !isset($var) || strlen(trim($var)) == 0;
4  }
5
6  $expected_password = 'top secret passphrase';
7  if (isEmpty($_GET["auth"]) || $_GET["auth"] !== $expected_password) {
8    http_response_code(401);
9    exit();
10 }
11
12 $name = $_GET["email"]; // Not license name but email
13 if (isEmpty($name)) {
14   http_response_code(400);
15   exit();
16 }
17
18 include_once('path/to/cocoafob/license_generator.php');
19 $generator = new License_Generator;
20 $product = "mynewapp";
21 $code = $generator->make_license($product, $name);
22
23 echo $code;
```

Lines 5–9 implement the very simple authentication check with a shared secret string. Lines 11–15 check if the required request parameter for the personalization is provided.

And that's all you need to do already. If you're paranoid, you can remove the string literal to define `$expected_password` and read the passphrase from a file outside your public htdocs. I don't see much benefit in that since this approach is comparatively weak, though.

Heads Up: This Is an Excerpt for the Book Sample!

Only a selection of the full e-book are available in this chapter. For the book sample, I picked some of the most interesting things so you get a feeling what is covered in the book. Keep in mind that some parts might not make sense on their own because context is missing from the samples.

In the full book, you'll learn how to:

- fully customize your FastSpring store,
- offer additional in-app purchases,
- support app activation via e-mail,
- use your web server for license activation,
- use the embedded in-app store of FastSpring to buy license codes,
- and much more!

My Other Books

Clean Cocoa is my mission: uncovering best coding practices so you can create great apps for macOS and iOS with beautiful, maintainable, and testable code.

Check out my other books online: https://christiantietze.de/books/

Available titles as of this release:

- **Exploring Mac App Development Strategies**, where I show how a modularized and scalable app can be built even when you decide to use Core Data for persistence, which is usually entangled everywhere.
- In **Creating Multi-Process Mac Applications** I show how an app can utilize and communicate with XPC services effectively to perform tasks in the background and sync data between different processes.