Member info-block

How does App Store purchase button work?

A state-based approach in SwiftUI

Pablo Baxter · Follow
3 min read · Oct 18, 2020

50 1

Author: Pablo Baxter

I guess you have hit this **App Store purchase button** a lot of times.

This tiny button supports the whole download process, displaying beautiful transitions & animations, and allowing simultaneous downloads on the same screen, but...

How does it work?

In this article, I will describe **how to design and build the App Store purchase button using SwiftUI.**

```
Table of contents:
1. A button made of states: States definition
2. The foundations: Action kind, capsule & initial state
3. Styling the button: Visual styles per state
4. States:
  Initial state
  Preparing to download
  Downloading (with progress)
  Done
5. Conclusion
Where to go from here
```

1. A button made of states

The *App Store purchase button* is about the process of downloading content, and as this process goes forward, the appearance changes. Every step of the process can be associated with an only state:

`Get`

The initial state. It has *three possible use cases:*

- The app was not downloaded previously.
- The app was downloaded previously and it is installed.
- The app was downloaded *previously* but it is not installed anymore.

`Preparing to download`

A loading indicator is displayed while pre-downloading operations like Applier's authentication are executed.

`Downloading`

A progress bar shows the download progress in real time. It also includes a button inside to cancel the download.

`Done`

Final state. A regular button that can execute any action needed (open the download app, etc.)

Status: Get, Preparing to Download, Downloading, Done

Translating the states into code, they are declared in an enumeration called `AutomaticButtonStates`:

```
enum AutomaticButtonStates {
    case getIsDownloaded: Bool
    case preparingDownload
    case getDownloadingProgress: Double
    case done
}
```

purchase-button-states.swift hosted with ❤ by GitHub view raw

2. The foundations

To handle the possible states correctly, I will use a View Model. This structure will help me to separate some business logic from the button view.

```
class PurchaseButtonViewModel: ObservableObject {

    /// ...

    /// Published var states AutomaticButtonStates
    var isDownloaded: Bool
    var isDone: Bool

    // ...

    var progress: Bool, isDownloaded: Bool {

        self.isDone = isDone
        self.isDownloaded = isDownloaded

        func updateStateFrom(_ dones, isDownloaded: Bool) -> AutomaticButtonStates {
            if isDone {
            return .done
            } else if isDownloaded {
            return .getIsDownloaded(isDownloaded)
            }
            ...

        self.state = .getIsDownloaded(isDone, isDownloaded: isDownloaded)
    }
}
```

purchase-button-viewmodel.swift hosted with ❤ by GitHub view raw

It's important that the current state can be observed, so the button responds in consequence to the updated state value [1].

The initial state will be set by checking if the app was already downloaded (`isDownloaded: Bool`) and if it is also installed (`isDone: Bool`)[2].

With this in mind, our initial `AutomaticStates` structure will be composed by:

- The View model (1)
- Actions: As mentioned in the state definition, during the process some external actions can be executed: **Preparing to download, download, cancel, set downloaded(1)**
- The init method (2) and the button declaration (4).

A percent sign can instantiate the button like very (3).

```
struct PurchaseButtonV2 {
    // 1.
    @ObservedObject var model: PurchaseButtonViewModel

    // 2.
    private enumAutomaticButtonStates: Observable, Identifiable, Equatable {
    var automaticStates: Observable, Identifiable, Equatable, Hashable = .init()
    var currentDownloadStates: [U :A =]
    var currentDownloadProgress: [U :A =]

    // 3.
    initializeAutomaticStates: Bool {
        // ...
        preparingDownloadStates: Observable, Identifiable, Equatable, Hashable = .init()
        downloadStates: Observable, Identifiable, Equatable, Hashable = .init()
        doneStates: Observable, Identifiable, Equatable, Hashable = .init()
        initializeStates: Observable, Identifiable, Equatable, Hashable = .init()
        ...
        self.model = PurchaseButtonViewModelInstance(isDone: isDone,
                                                     isDownloaded: isDownloaded)

        self.preparingDownloadStates = preparingDownloadStates
        self.downloadStates = downloadStates
        self.doneStates = doneStates
        self.initializeStates = initializeStates
    }

    // 4.
    var body: View {
        AutomaticButton {
        ...
        }, label: {
        ...
        }
    }
}
```

purchase-button-states.swift hosted with ❤ by GitHub view raw

Our `AutomaticStates` can be called like this by now:

3. Styling the button

Once the states and the main structures are defined, let's take a look at how to set the visual style for every possible state.

To configure every button style, I define a protocol with the visual properties to display:

```
protocol AutomaticButtonStates {
    // ...
    /// Maybe some states don't need to define all the visual properties. For that reason,
    /// an extension set default values.

    /// Following these AutomaticButtonStates protocol, let's define the structures
    /// that hold the visual properties for each state.

    /// AutomaticButtonStates has two different visual layouts, depending on whether
    /// the content was downloaded previously.

    /// Once the styling is defined, let's declare the variable properties in
    /// AutomaticStates, to return the style properties needed depending on the
    /// selected state.

    /// How to apply these styles in the button? Let's create a custom AutomaticStyle,
    /// using the custom button state for getting the right properties in every
    /// moment.

    /// Only a "GET" is displayed... Let's configure the states!

    // ...
}
```

Custom button states

Time to set the button style in our `AutomaticButton` and also some frame constraints and the animation type between states.

```
// Only a "GET" is displayed... Let's configure the states!

// ...

// 4. States
// It's time to automate the process! Let's discover, one at a time, the states that
// compound this button:

// Get

// ...

// Get: Initial state. The download process is triggered when the button is tapped.

The button has added a switch statement to distinguish between cases in the
// the block and the label view.

If the download is happening for the first time, a Text element is displayed
// (1). In case the download was previously made, an Image(systemImage) image is
// displayed (2).

// ...

// Preparing to download

// ...

// Preparing to download: A loading indicator is displayed while a pre-downloading
// operation like Applier's authentication is being executed.

// Creating the loading indicator
In ProgressViewStyle, a stroked circular loading indicator rotates regularly,
// using a scheduled timer. This timer starts when the circle appears on the
// screen, and stops when it disappears.

// CircleProgress: creates the gap in the circle (1)
```

Member info-block

Back to the `PurchaseButton`:

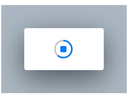
When the button is tapped, the state switches to `prepareUpdateUnlock`, and the `prepareUpdateUnlock` is executed (1).

Once the state is updated, the `CircularProgressBar` is displayed (2).

> *To test the button, I included in my parent view's custom actions a `Timer` that simulates the request-response process (3).*

• • •

Downloading



Downloading: *A progress bar shows the download progress in real-time. It also includes a button inside to cancel the download.*

To display the downloading state, I create an internal component called `CircularProgressBarView`, that is composed of:

- A stroked circular progress bar that is filled as the download progress moves along.
- An inner rectangle inside the progress bar that can be tapped to cancel the download.

> *As in the loading state, `.drawCircle()` creates a gap in the circle, displaying the current progress.*

Back in `PurchaseButton`:

After `prepareUpdateUnlock` succeeds, it time to start the download: the state switches to `.downloading` and the external action for download `downloadUnlock(_ result ...)` is triggered (5).

The download action is complete when the returned progress value reaches `1.0`, At that moment the state switches to `.done` (6).

To reflect the download process, the button becomes the circular progress bar (3).

Canceling the download

As it was mentioned before, the download process can be canceled by tapping on the loading indicator inner rectangle.

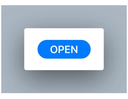
The cancellation provokes the button to switch back to `.get` state (4). Also, the custom `cancelDownloadAction` (5) is triggered for logic outside the button.

> *This cancellation process is also triggered when the pre-downloading (2) or the download (3) action fails.*

The call for `PurchaseButton` gets updated in its `downloadAction` and `cancelDownloadAction` code blocks:

• • •

Done



Done: *Final state. A regular button that can execute any action needed (open the download app, etc.)*

When the download finishes, the user-defined `downloadAction` (1) is executed and a `.open` element is displayed (2).

> *The custom code block for `downloadAction` can be filled as needed (normally open the downloaded content).*

• • •

Conclusions

Creating a complex button like the App Store's one can be a tough job if it is not clear and structured.

Swift enumerations and a declarative programming language like SwiftUI allow great results, making the creation of visual elements much easier and creative.

Where to go from here

The button could be improved by including some nice features like:

- Dark mode support
- Free / Paid app differentiation (texts, behaviors, etc.)

The biggest things are yet to come! Stay tuned!

App Development · iOS · Development · Swift · SwiftUI

Written by Pablo Blanco

Follow

187 Followers

iOS/Angular at Sparks

More from Pablo Blanco

Pablo Blanco in Better Programming · How to Validate Complex Passwords in SwiftUI · Simple yet elegant way to do it · 3 min read · Nov 05, 2020

Paul Pamplaster · UML Class Diagram Arrows Guide · Most people are confused by all kind of lines, arrows and shapes in UML class diagrams · 2 min read · Sep 23, 2020

Medium · The "Those who can't do, teach" fallacy · Why the century-old century-old quote by George Bernard Shaw's quote perpetuates the reputation of... · 3 min read · Sep 7, 2020

Pablo Blanco in The Startup · How to Retain Your Users While Loading Data in SwiftUI · Simple yet elegant way to do it without creating the wait time problem · 3 min read · Nov 5, 2020

See all from Pablo Blanco

Recommended from Medium

Pavel Cherets (L) in Prototypr · UX/UI Design Trends Going Into 2020 · Every year, we have a line-up of new design trends that not only look good, but also stick... · 10 min read · Dec 10, 2020

Jacob Bennett | Learn to Code | The 5 paid subscriptions I actually use in 2020 as a software engineer · Tools I use that are cheaper than Netflix. · 3 min read · Jan 4

Lists

Growth Marketing · 12 stories · 19 saves

Modern Marketing · 10 stories · 18 saves

Medium's Huge List of Publications Accepting... · 392 stories · 2796 saves

Staff Picks · 379 stories · 1154 saves

Jonathan Dietl · Apps I Use And Why You Should Use Them · Hey, I'm a young entrepreneur, and all of my friends are, or higher-ability minded or should I state a... · 3 min read · Sep 16, 2020

Adrian Carbonaro (L) in Life Planet · 10 little UI design rules that make a big impact · Simple, actionable and easily to remember an example user interface using logical rules or... · 10 min read · Mar 16, 2020

Dylan Raga in J. Developer.com · Agile Has Failed, Officially · Agile is the perfect scrum, and all of my friends are, or higher-ability minded or should I state a... · 3 min read · Dec 3, 2020

James Prendeville in Product in Public · These Words Make It Obvious That Your Team Is Working As A Cult · Your Teams accomplishments are in the... · Feb 2, 2020

See more recommendations

Help · Status · About · Careers · Blog · Privacy · Terms · Text to speech · Teams