Managing complexity

Bending Spoons @ PoliMi - 2019
The case of A “Simple” Application
A “Simple” Application
A “Simple” Application
A “Simple” Application
A “Simple” Application
What can go Wrong?
State Duplication

- A local variable in a class
  (let’s say: workout_frequency)
- The same local variable in another class
- The user interacts with the app to update a value.
- Then navigate in the other one and it’s out of sync

```javascript
const userProfile = {
  user_name: "Riccardo",
  user_image: "https://66.medla.tumblr.com/avatar_122b61272c86_128.png",
  age: 31,
  workout_frequency: 2
}

const planManager = {
  user_level: 1,
  workout_frequency: 2,
  difficulty: "hard",
  duration: 30
}
```
The user is interacting with the app (e.g. saving a workout he just finished)

A background update arrives (e.g.: healthkit telling you that the user has burn calories walking)

Both the app and HK tries to update the same variable.

Possible outputs?
My state changed, my UI doesn’t!

- Similar to the other cases.
- The user is seeing the app, and there is a timer running.
- The timer ticks, you see it in the console.
- The UI is stuck.

```javascript
let liveManager = {
  currentTime: 0,
  currentExercise: "push_up",
  currentInterval: undefined
}

const start = (liveManager) => {
  liveManager.currentInterval = setInterval(() => {
    liveManager.currentTime += 1;
  }, 1000);
}

const stop = (liveManager) => {
  clearInterval(liveManager.currentInterval);
}

function Timer(props) {
  return <h2>{props.time}</h2>
}

liveManager.start();
return <Root>
  <Timer time={liveManager.currentTime} />
</Root>
```
Data passing

- The root element holds an object.
- Children require properties of the object.
- You need to pass the property down to the most inner child.
- The situation worsen when a child has to bubble a result to the parent!
Debugging

- Your app is crashed.
- Luckily the debugger was turned on.
- You are looking at a line
- "This can't be right. That value can never be set!"
- ...
- "Oh... It's a race condition"
- Good luck!

6 STAGES OF DEBUGGING

1. That can't happen
2. That doesn't happen on my machine
3. That shouldn't happen
4. Why does that happen?
5. Oh I see
6. How did that ever work?
How can we manage it?
The main issues

Mutability & asynchronicity

Immutability & predictability
Single source of truth

- Single State
- Can be fetched from a server/backup
- Easier to inspect/debug
- Can be serializable thus stored and restored.

The state of your whole application is stored in an object tree within a single store.

```
<<Interface>>
Store
- state: AppState
- getstate(): AppState
```
Immutability

- No one but the store can update the state
- The changes are serialized (ordered, no race condition)
- Action: PO(J)O that describe how the state changes

The only way to change the state is to emit an action, an object describing what happened.

```java
// <<Interface>>
interface Store {
    AppState state;
    AppState getState();
    void dispatch(Action action);
}
```
No side effects

- Pure function: return value is only determined by its input values
- Reducer - Inputs:
  - action
  - current state
- Reducer - Output:
  - a modified copy of the state

To specify how the state tree is transformed by actions, you write pure reducers.

```javascript
function reduce(action, state) {
    switch (action.type) {
        case "ADD"
            return [
                ...state,
                {
                    title: action.title,
                    description: action.description,
                    completed: false
                }
            ]
        // other cases
    }
}
```
Positive Property: Observability

Given
- A single source of truth
- A set of views and object
- The state

Easy to implement the observer pattern:
- Listener: (state) => {}

<table>
<thead>
<tr>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store</td>
</tr>
<tr>
<td>- state: AppState</td>
</tr>
<tr>
<td>- listeners: Listener[]</td>
</tr>
<tr>
<td>- getstate(): AppState</td>
</tr>
<tr>
<td>- dispatch(Action)</td>
</tr>
<tr>
<td>- add(Listener): Id</td>
</tr>
<tr>
<td>- removeListener(Id)</td>
</tr>
</tbody>
</table>
Positive Property: Predictability

- Given
  - An initial state
  - A reducer
  - A sequence of action

- You can always determine the new state

\[ S_f = f(S_i, r, \sum a) \]
Positive Property: Predictability

- Given
  - An initial state
  - A reducer
  - A sequence of action

- You can always determine the new state

\[ S_f = f(S_i, r, \sum a) \]
const storeFactory = (state, reducer) => {

    let _state = state;
    let _listeners = {};
    let _reducer = reducer;

    return {
    }
}
const storeFactory = (state, reducer) => {

    let _state = state;
    let _listeners = {};
    let _reducer = reducer;

    return {
        dispatch: function(action) {
            const newState = _reducer(_state, action)
            _state = newState

            const keys = Object.keys(_listeners)
            for (k in keys){
                let key = keys[k]
                let listener = _listeners[key]
                listener(_state)
            }
        },
    }
}
const storeFactory = (state, reducer) => {

  let _state = state;
  let _listeners = {};
  let _reducer = reducer;

  return {
    dispatch: function(action) {
      const newState = _reducer(_state, action)
      _state = newState

      const keys = Object.keys(_listeners)
      for (k in keys) {
        let key = keys[k]
        let listener = _listeners[key]
        listener(_state)
      }
    },

    getState: function() {
      return { ..._state }
    }
  },
}
const storeFactory = (state, reducer) => {

  let _state = state;
  let _listeners = {};
  let _reducer = reducer;

  return {
    dispatch: function(action) {
      const newState = _reducer(_state, action)
      _state = newState

      const keys = Object.keys(_listeners)
      for (k in keys){
        let key = keys[k]
        let listener = _listeners[key]
        listener(_state)
      }
    },

    getState: function() {
      return { ..._state }
    },

    add: function(listener) {
      const uuid = guid()
      _listeners[uuid] = listener
      return uuid
    },

    remove: function(uuid) {
      delete _listeners[uuid]
    }
  }
}
const storeFactory = (state, reducer) => {

    let _state = state;
    let _listeners = {};
    let _reducer = reducer;

    return {
        dispatch: function(action) {
            const newState = _reducer(_state, action)
            _state = newState

            const keys = Object.keys(_listeners)
            for (k in keys) {
                let key = keys[k]
                let listener = _listeners[key]
                listener(_state)
            }
        },

        getState: function() {
            return {..._state}
        }
    },

    add: function(listener) {
        const uuid = guid()
        _listeners[uuid] = listener
        return uuid
    },

    remove: function(uuid) {
        delete _listeners[uuid]
    }
}

// A sample of a reducer
const reducer = (state, action) => {
    if (action === "add") {
        return { counter: state.counter + 1 }
    } else if (action === "sub") {
        return { counter: state.counter - 1 }
    } else {
        return state
    }
}
A very simple ReactRedux application
How do we solved it!
One of the first redux-inspired Swift framework
Brings all the positive aspects of redux and pushes them further
  - Open source
  - Github
Katana Pros

- Isolated Actions: StateUpdaters and SideEffect
- Interceptors
- Better Modularization
Managers

- Implements the business logic
- Only pure functions ⇒ no internal state!
- Reusable
- Testable
- Cannot dispatch!
- Side Effects join the managers functions

```swift
class PlanGenerationManager {

    func generateWeekPlan(
        from input: PlanGenerationInput) throws
    -> WeekPlan

    func generateChallenge(
        from input: ChallengeGenerationInput) throws
    -> Challenge

    }

struct PlanGenerationInput {

    let userID: String
    let userGoal: Goal
    let userFitnessLevel: FitnessLevel
    let workoutsPerWeek: Int
    let gender: Gender
    let preferredWorkoutDuration: WorkoutDuration
    // ... other props
    let allWorkouts: [Workout.ID: Workout]

    }
```
Promises!

- Used to simplify asynchronicity
- Promise that an async function will resolve
- Write async code as if sync
- No callback hell
- Note: several languages have them embedded in the language - C# was the first, JS

```swift
struct ReadDatabase: AppSideEffect {
    func sideEffect(_ context: SideEffectContext) throws {
        let dbManager = context.databaseManager
        // start reading the DB
        dbManager.extractSevenDaysOfHourlyData() { hourly in
            dbManager.extractDailyData() { daily in
                dbManager.extractWeeklyData() { weekly in
                    dbManager.extractMonthlyData() { monthly in
                        dbManager.extractYearlyData() { yearly in
                            let historyState = ActivityHistoryState(
                                hourlyEntries: hourly,
                                dailyEntries: daily,
                                weeklyEntries: weekly,
                                monthlyEntries: monthly,
                                yearlyEntries: yearly)
                            let updateState = UpdateState(
                                historyState: historyState)
                            try await(context.dispatch(updateState))
                            context.dispatch(DataManagerLogic
                                .ReadRemainingHourlyData())
                        }
                    }
                }
            }
        }
    }
}
```
Promises!

- Used to simplify asynchronicity
- Promise that an async function will resolve
- Write async code as if sync
- No callback hell
- Note: several languages has them embedded in the language - C# was the first, JS

```swift
struct ReadDatabase: AppSideEffect {

    func sideEffect(_ context: SideEffectContext) throws {

        // start reading the DB
        let hourly = context.databaseManager.extractSevenDaysOfHourlyData()
        let daily = context.databaseManager.extractDailyData()
        let weekly = context.databaseManager.extractWeeklyData()
        let monthly = context.databaseManager.extractMonthlyData()
        let yearly = context.databaseManager.extractYearlyData()

        try await all([hourly, daily, weekly, monthly, yearly])

        let historyState = ActivityHistoryState{
            hourlyEntries: hourly.result!,
            dailyEntries: daily.result!,
            weeklyEntries: weekly.result!,
            monthlyEntries: monthly.result!,
            yearlyEntries: yearly.result!
        }

        let updateState = UpdateState(historyState: historyState)
        try await context.dispatch(updateState)

        context.dispatch(DataManagerLogic
            .ReadRemainingHourlyData())
    }
}
```
Marty McFly

- Undo/Redo pattern made easy
- Add checkpoints to the state
  (UndoRedoActions.Checkpoint)
- To undo: UndoRedoActions.Undo
- To Redo: UndoRedoActions.Redo
- There are actions to clean the stacks
Marty McFly

- Undo/Redo pattern made easy
- Add checkpoints to the state
  $(\text{UndoRedoActions.Checkpoint})$
- To undo: $\text{UndoRedoActions.Undo}$
- To Redo: $\text{UndoRedoActions.Redo}$
- There are actions to clean the stacks
struct AppState: State {

    // MARK: Persisted slices
    var meState: MeState = MeState()
    var settingsState: SettingsState = SettingsState()
    var monopolyState: MonopolyState = MonopolyState()
    var reviewState: ReviewState = ReviewState()
    var internalMigration: Int = 0
    var userProfileInfo: AppState.UserProfileInfo = AppState.UserProfileInfo()
    var scenesState: AppState.ScenesState = AppState.ScenesState()
    var soundsState: AppState.SoundsState = AppState.SoundsState()

    // MARK: Partially Persisted slices
    var onboardingState: AppState.OnboardingState = .landing
    var turnstilesState: AppState.TurnstilesState = AppState.TurnstilesState()
    var contentState: AppState.ContentState = AppState.ContentState()

    // MARK: Not Persisted slices
    var emporium: EmporiumState = EmporiumState()
}

Katana: Initialization

typealias Options = [LaunchOptionsKey: Any]?
func application(_ application: UIApplication, launchOptions: Options)
  -> Bool {
    self.window = UIWindow()
    self.window?.makeKeyAndVisible()

    self.store = Store(
      interceptors: self.storeInterceptors,
      stateInitializer: self.persistStore
        .katanaStateInitializer
    )

    // Tempura!
    self.store?.dependencies?
      .navigator.start(
        using: self,
        in: self.window!,
        at: Screen.appSetup.rawValue
      )

    return true
  }

private var storeInterceptors: [StoreInterceptor] {
  return [
    middlewareToInterceptor(self.persistStore.katanaMiddleware),
    ObserverInterceptor.observe(self.observers),
  ]
}

private var observers: [ObserverInterceptor.ObserverType] {
  return [
    .onStart([
      ContentLogic.ShuffleContentIfNeeded.self,
      ScenesLogic.FixCustomConfigurationsIfNeeded.self
    ]),
    .onNotification(UIResponder.keyboardWillShowNotification, [KeyboardLogic.KeyboardWillShow.self]),
    .onNotification(UIResponder.keyboardWillHideNotification, [KeyboardLogic.KeyboardWillHide.self]),
    .onStateChange(AudioPlayerLogic.audioPlayerStateChangeObserver, [BackgroundAudioLogic.UpdateNowPlayingInfo.self]),
    .onStateChange(MonetisationLogic.premiumStateChangeObserver, [DownloadLogic.DownloadAllNecessaryResources.self]),
    .onDispatch(ShowSurvey.self, [ReviewLogic.ShowSurvey.self]),
    .onDispatch(ShowReview.self, [ReviewLogic.ShowReview.self])
  ]
}
Katana: How a project looks

- PlayondApp
  - Products
  - App
    - Application
    - Assets.xcassets
  - Logic
  - Metrics
  - Models
  - Resources
  - State
  - UI
    - .ambrogio
    - Frameworks
    - Pods
  - State
    - AppState.swift
    - EmporiumState.swift
    - EnvironmentState.swift
    - LoginFlowState.swift
    - MonetizationState.swift
    - OnboardingState.swift
    - PlayondState.swift
    - TurnstileState.swift
Katana: How a project looks
Katana: How a project looks
Tips’n’Tricks

- Start from the State
- Keep high separation of concerns
  - Use Managers to encapsulate the logic
  - Keep SideEffects simple: chain of managers functions and dispatches
- Prefer composition over inheritance
Tips’n’Tricks

- Start from the State
- Keep high separation of concerns
  - Use Managers to encapsulate the logic
  - Keep SideEffects simple: chain of managers functions and dispatches
- Prefer composition over inheritance
References

- Redux: https://redux.js.org/introduction/motivation
- Katana: https://github.com/BendingSpoons/katana-swift
- Promise: https://github.com/malcommac/Hydra
- Composition over inheritance: https://bit.ly/1FgTj4x
Questions?
Thank you!