iPraktikum 2018 Intro Course

App 2

— Session 04 —

What you will build

You will implement the Model-View-Controller (MVC) architectural style in an iOS app.

What you will learn

Why MVC is useful and how it is used in iOS apps
How to persist data locally in text files
How to make the UI dynamically react to user input
How to add a Tab Bar to your application

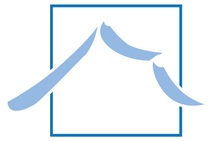
Prerequisites

- Class vs Struct
- Protocols
- Computed Property
- Extension
- JSON

Key Vocabulary

- Observer Pattern
- Cocoa Touch
- Entity & Boundary Objects
- Codable
- Tab Bar Controller

Copyright, Content & Referrals and Links



Lehrstuhl für
Angewandte Softwaretechnik

- **Copyright 1980 - 2018 Technische Universität München - Chair for Applied Software Engineering (shortened TUM LS1)**
  - Unless explicitly stated otherwise, all rights including those in copyright in the content of this document are owned by or controlled for these purposes by TUM LS1.
  - Except as otherwise expressly permitted under copyright law or TUM LS1's Terms of Use, the content of this document may not be copied, reproduced, republished, posted, broadcast or transmitted in any way without first obtaining TUM LS1's written permission.
- **Content**
  - TUM LS1 reserves the right not to be responsible for the topicality, correctness, completeness or quality of the information provided.
  - Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected.
  - All offers are not-binding and without obligation. Parts of the pages or the complete publication including all offers and information might be extended, changed or partly or completely deleted by the author without separate announcement.
- **Referrals and Links**
  - The author is not responsible for any contents linked or referred to from this document.
  - If any damage occurs by the use of information presented there, only the author of the respective pages might be liable, not the one who has linked to these pages.
  - Furthermore the author is not liable for any postings or messages published by users of discussion boards, guestbooks or mailing lists provided on his page.
- **By obtaining and reading this document, you agree to this copyright statement.**

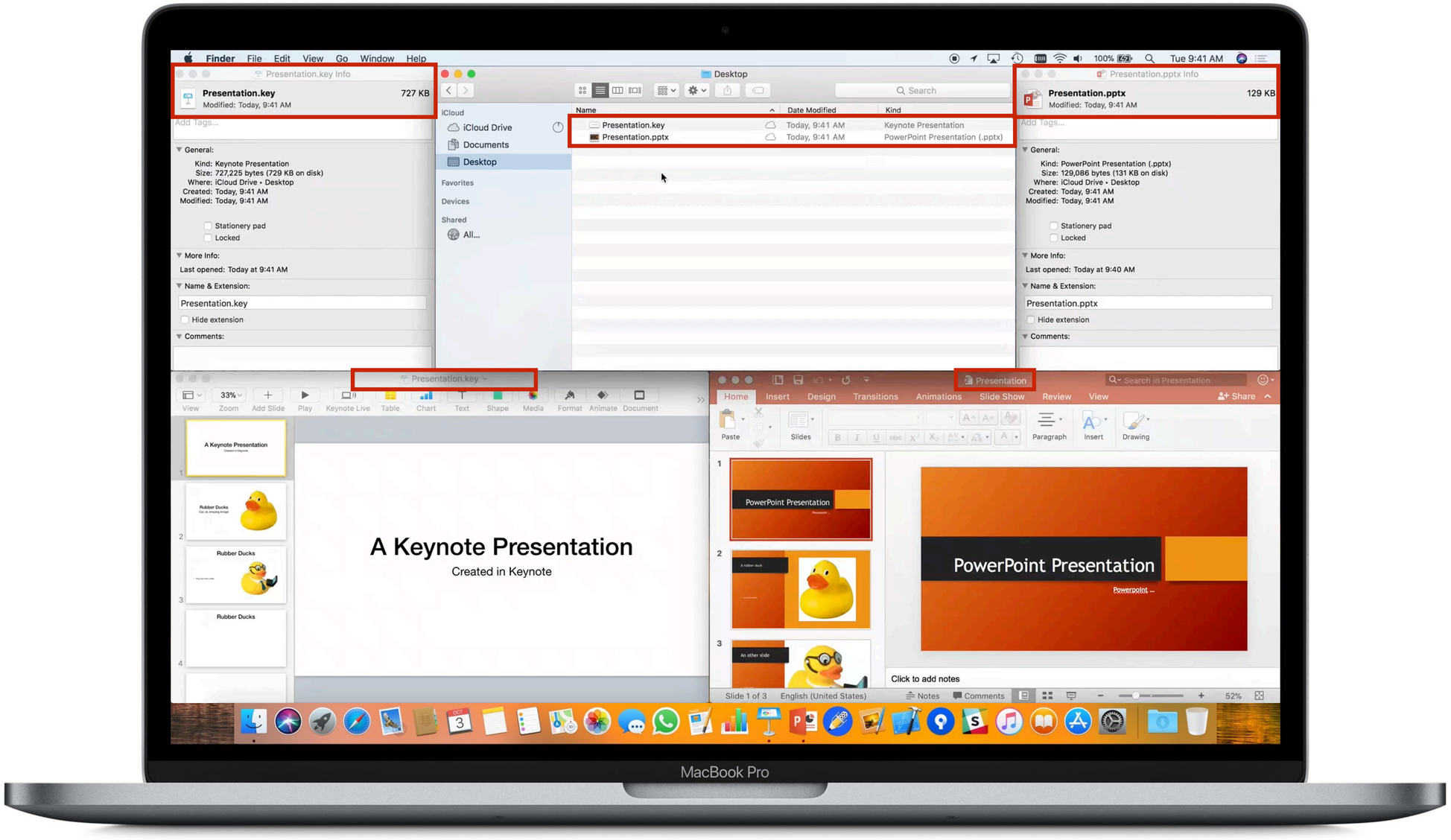
Overview

- **Model-View-Controller**
  - **Motivation**
  - Application in Cocoa Touch
- MVC implementation in the Xpense app
- Persistent on-device data storage
- UI Improvements
- Intermediate Interface Builder
  - Adding a tab bar and an additional scene
  - Examining the view hierarchy

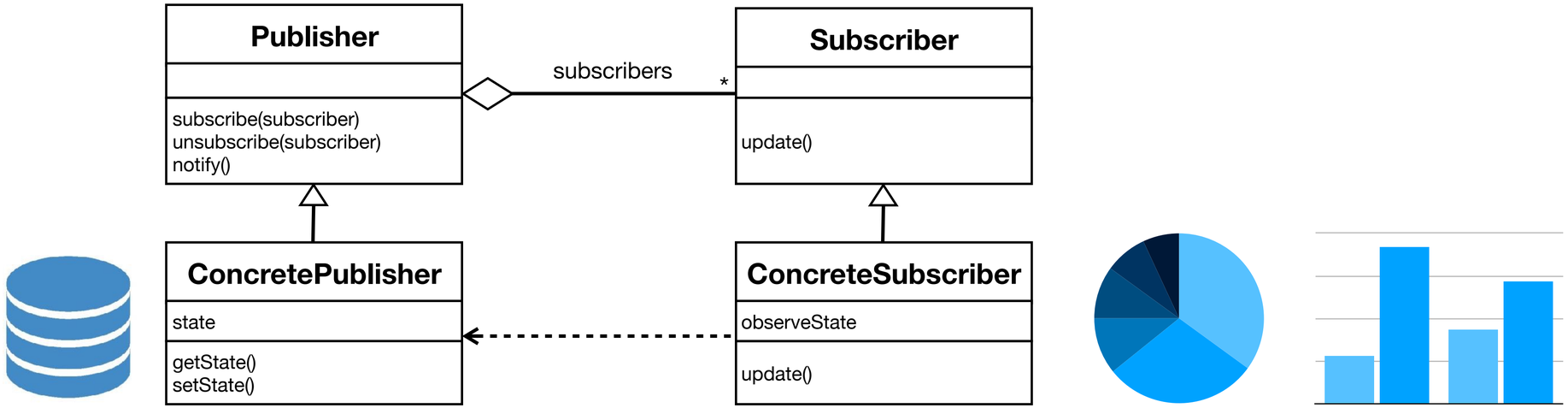
MVC - Motivation

- Goal: Provide multiple views of the current state of an instance, e.g. Histogram view, pie chart view, timeline view and decouple data and the view presenting it
- Problem with objects that change state quite often (e.g. tracking website data)
- Requirements
  - Maintain consistency across (redundant) views, whenever the state of observed objects change
  - The system design should be highly extensible
  - It should be possible to add new views without having to recompile the observed object or the existing views.

Example: Redundant views of an object

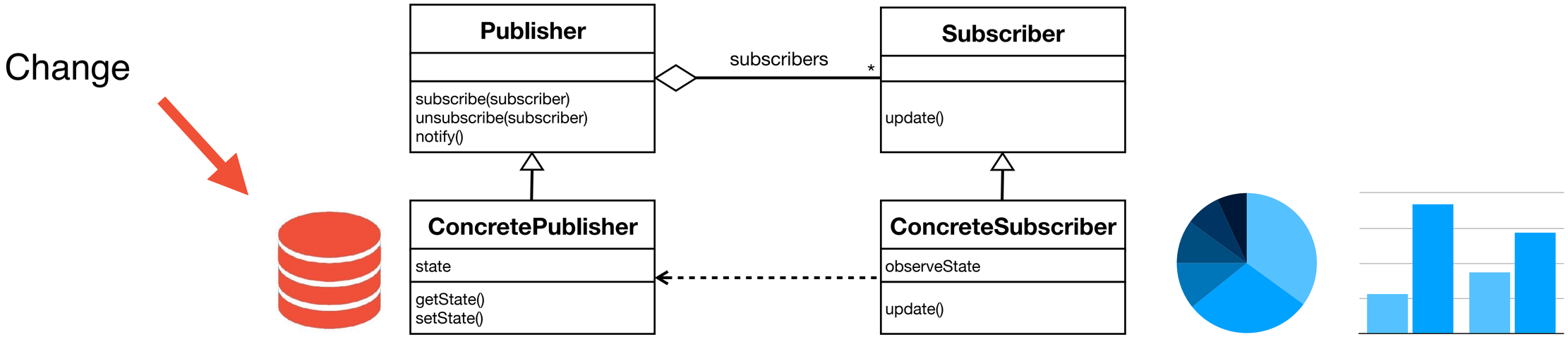


Solution: Observer Pattern (Decouple an abstraction from its views)



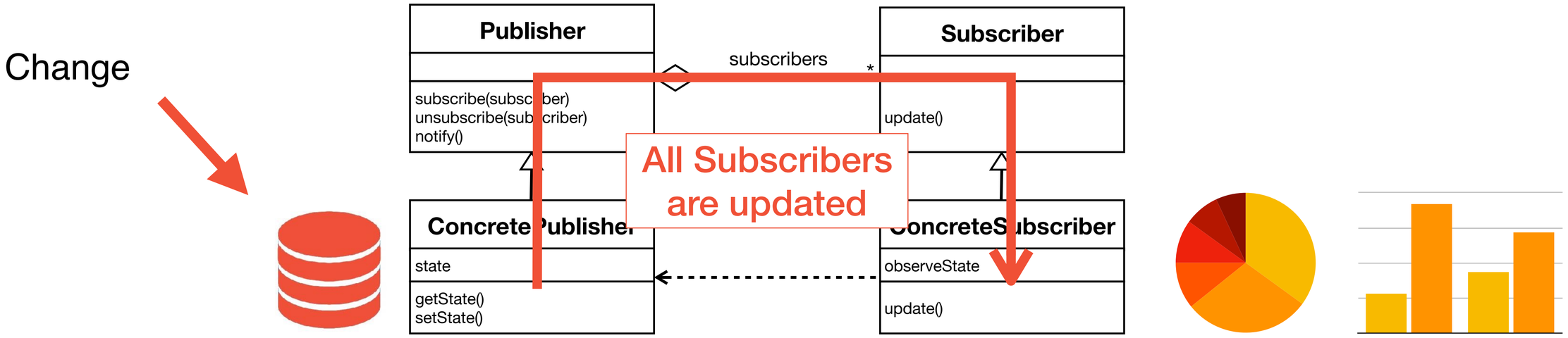
- The Publisher represents an entity object
- Subscribers attach to the Publisher by calling `subscribe()` and detach from the Publisher by calling `unsubscribe()`
- Each subscriber has a different representation of the state of the Publisher
  - The state is contained in the subclass `ConcretePublisher`
  - The state can be obtained and set by subclasses of type `ConcreteSubscriber`

Solution: Observer Pattern (Decouple an abstraction from its views)



- The Publisher represents an entity object
- Subscribers attach to the Publisher by calling `subscribe()` and detach from the Publisher by calling `unsubscribe()`
- Each subscriber has a different representation of the state of the Publisher
  - The state is contained in the subclass `ConcretePublisher`
  - The state can be obtained and set by subclasses of type `ConcreteSubscriber`

Solution: Observer Pattern (Decouple an abstraction from its views)



- The Publisher represents an entity object
- Subscribers attach to the Publisher by calling `subscribe()` and detach from the Publisher by calling `unsubscribe()`
- Each subscriber has a different representation of the state of the Publisher
  - The state is contained in the subclass `ConcretePublisher`
  - The state can be obtained and set by subclasses of type `ConcreteSubscriber`

Observer Pattern and its Limitations

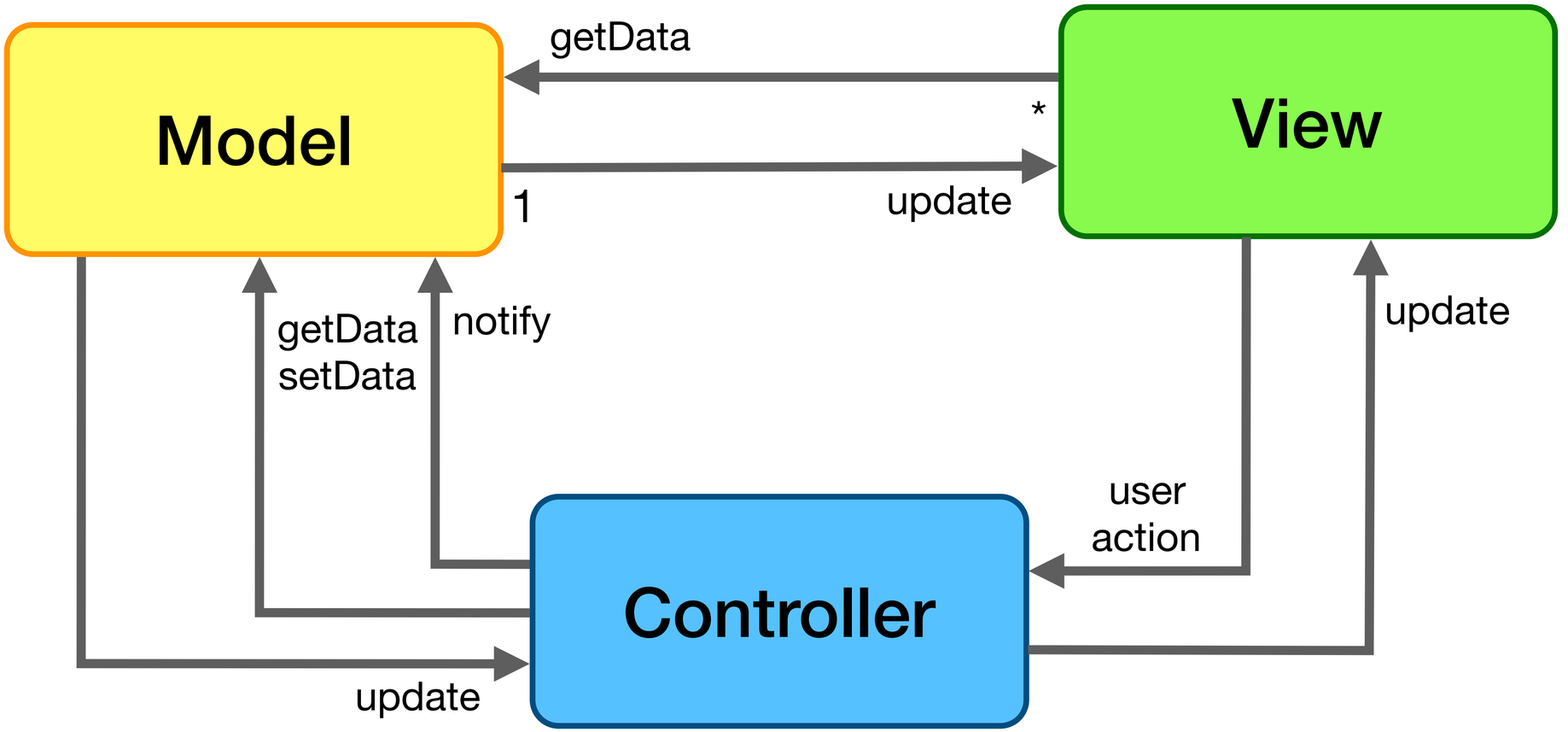
- Models a 1-to-many dependency between objects. Connects an observed object (and its state), the publisher, with many observing objects, the subscribers.
- Consistency of multiple subscribers with the publisher is maintained through:
  - push: publisher notifies all subscribers of the change; subscribers decide themselves whether they obtain information of the changed state
  - push-update: published notifies subscribers and also sends changed state
  - pull: subscribers inquire about the state of the publisher themselves
- Typically high coupling between entity object (data = publisher) and boundary objects (UI = subscribers)
- Problems of high coupling:
  - Data objects cannot be changed without adapting the UI
  - Sometimes boundary objects need to be publishers and the entity objects need to be subscribers

Solution: Model-View-Contoller (MVC)

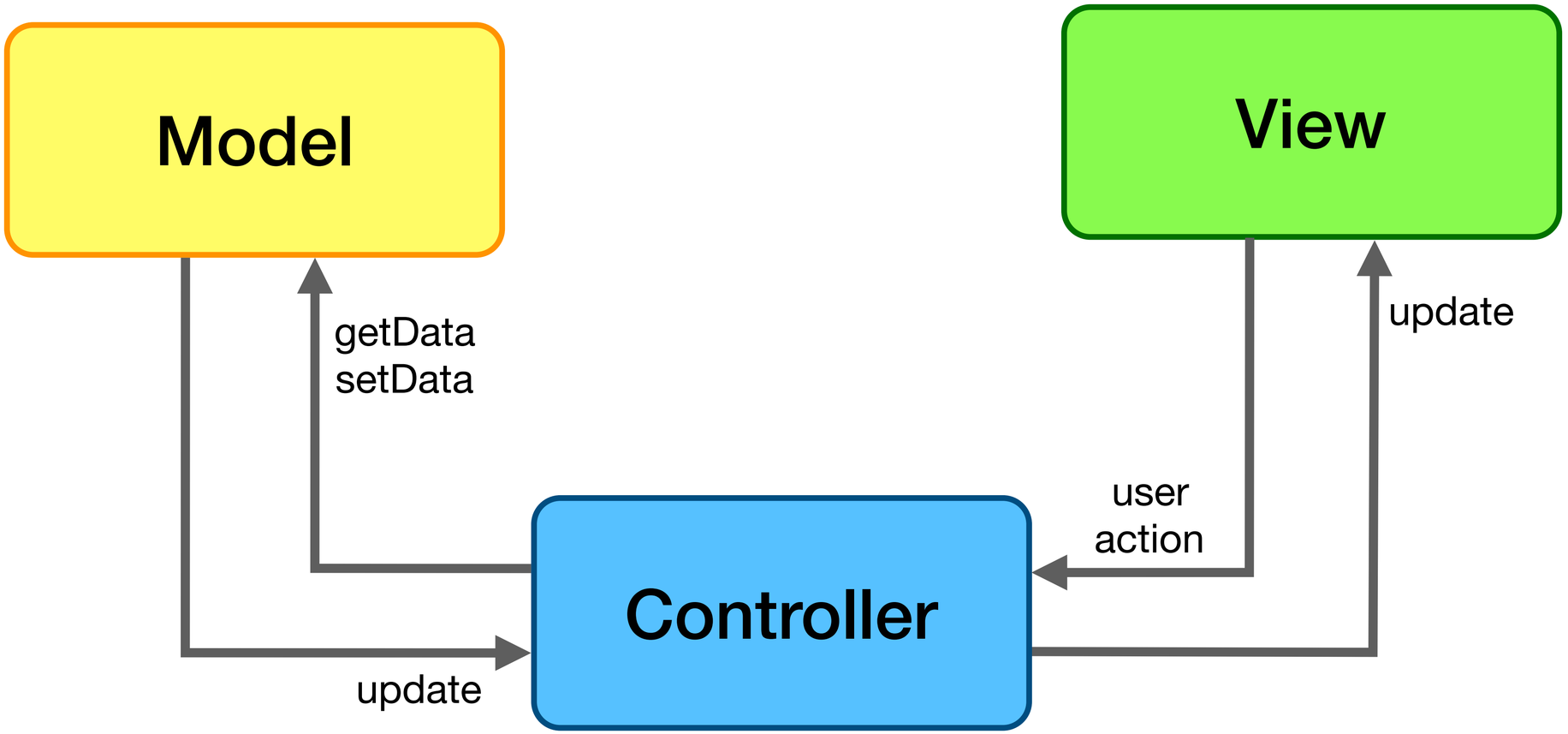
- Model-View-Contoller (MVC) is an **architectural style**
- It Decouples **data access** (entity objects) and **data presentation** (boundary objects)

Model	• **Data access** subsystem • Includes objects that represent the application domain knowledge
View	• **Data presentation** subsystem • Includes objects that are a **visual representation of the model**
Controller	• Mediates between View and Model • Includes objects that **link model objects with their views**

Model-View-Controller



Model-View-Controller in Cocoa Touch



Overview

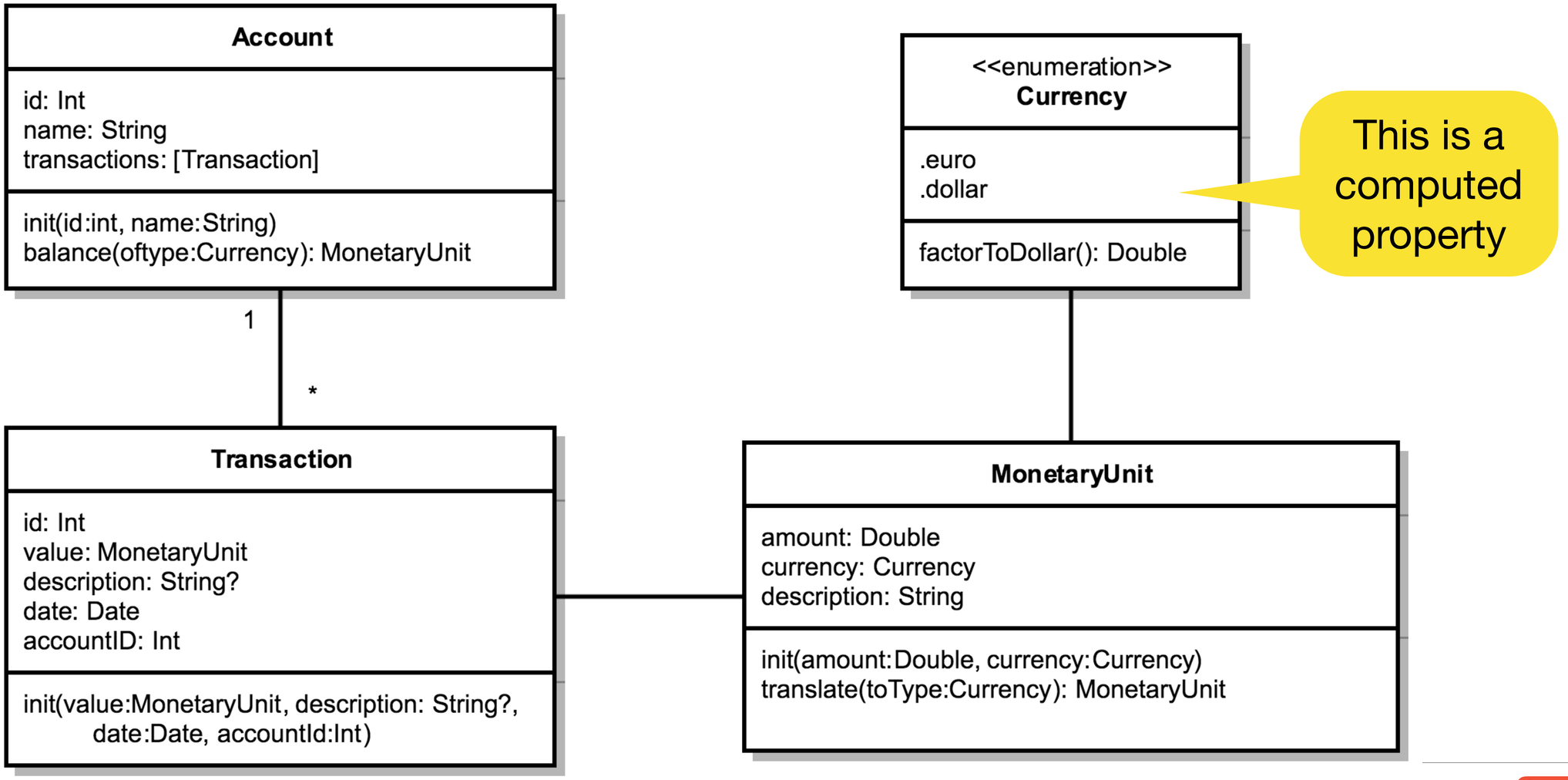
- Model-View-Controller
  - Motivation
  - Application in Cocoa Touch
- **MVC implementation in the Xpense app**
- Persistent on-device data storage
- UI Improvements
- Intermediate Interface Builder
  - Adding a tab bar and an additional scene
  - Examining the view hierarchy

What's new since App 01?

- In-between sessions, we add some things to the Xpense app to make it more convenient for you.
- These are things that we don't code together with you in the session, but we might explain them. For those that we don't explain, you can check the code yourself, understand it and use the same concepts in your apps!
- Since **App 01**, we added the following things:
  - An Extension for UITextField which contains a method to mark the field's border red. We will use this later in the session!
  - The "Save Transaction" button and the amount lable animates up when the keyboard is displayed

! Always start with our version of the app at the beginning of a session!

The model we will create today



Exercise 1: Create your Model

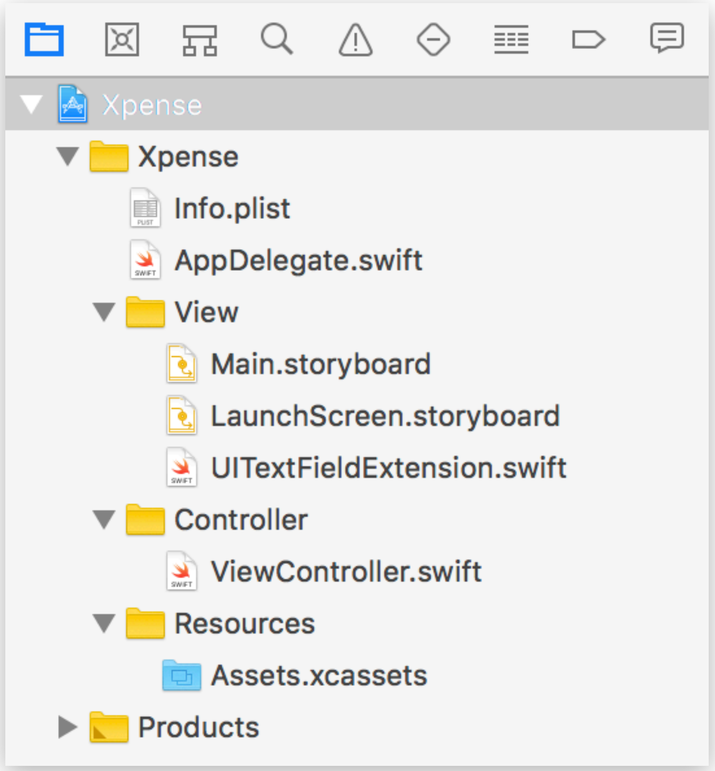
Task 1.1: Inspect the new project structure

Task 1.2: Create classes and structs needed for the Model

Task 1.3: Clean up your project structure

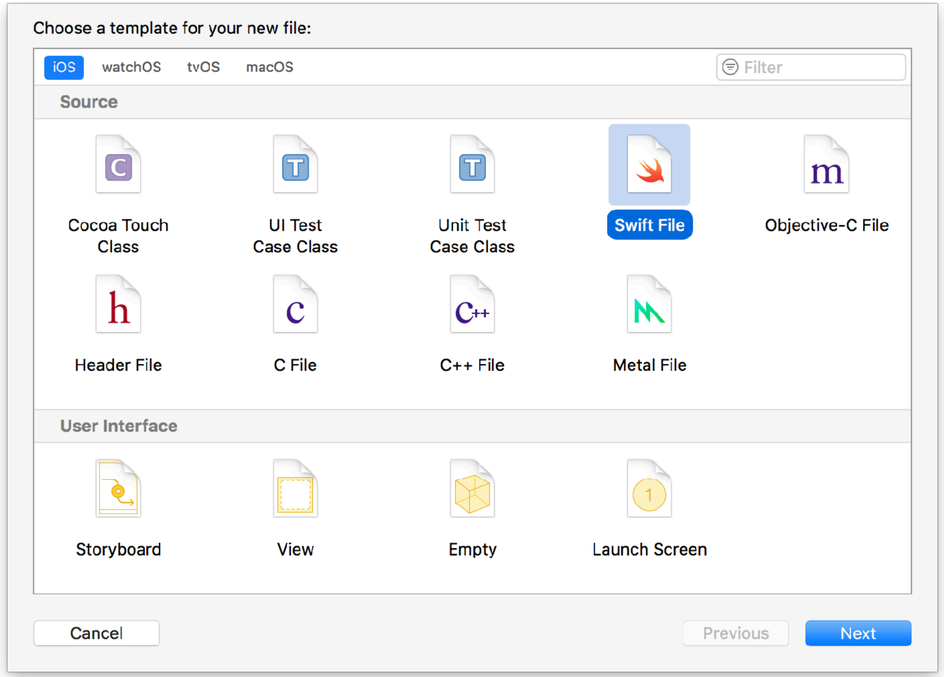
Solution 1.1: Inspect the new project structure

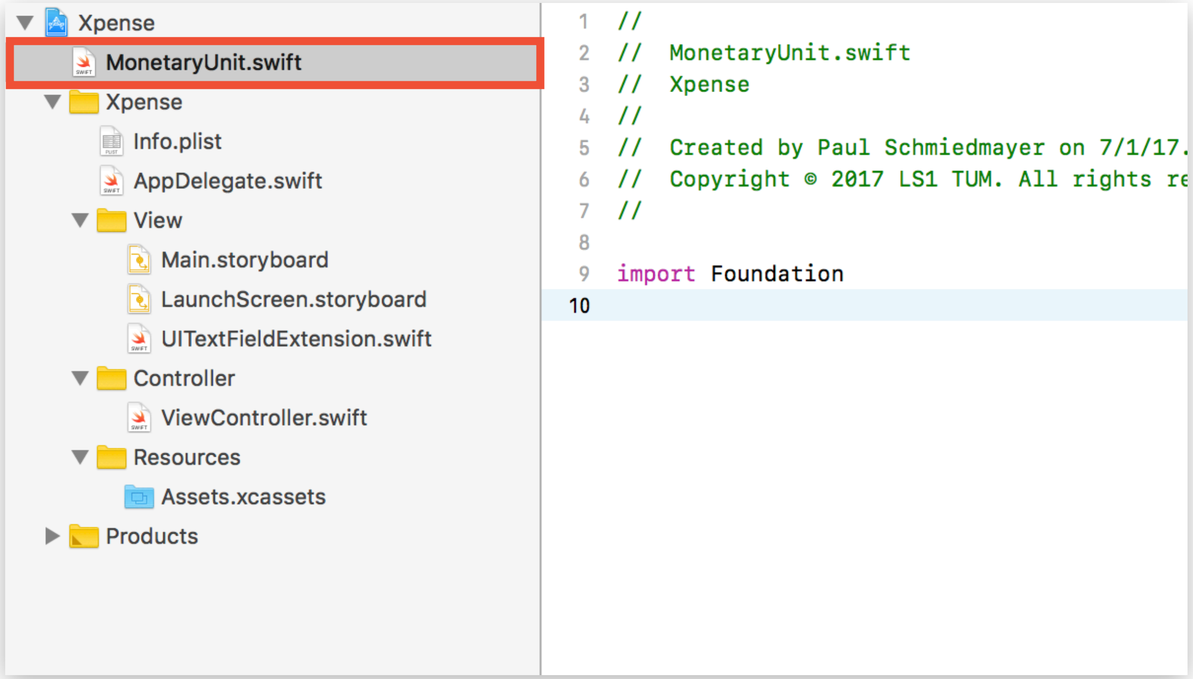
- Open the project template for this session (copy it to a different folder first!)
- We have rearranged the files to make the MVC structure visible
- note that there are no Model classes yet



Solution 1.2: Create classes and structs needed for the Model

- Click ⌘ + N or File > New > File…
- Choose to create a Swift File
- Put it in your project folder and name it `MonetaryUnit.swift`





Solution 1.2: Create classes and structs needed for the Model

- We want to support Euro and US Dollars as currency. Add an `enum` to the file:

```
enum Currency: String {
    case euro = "€"
    case dollar = "$"
}
```

- A `Monetary Unit` consists of an `amount` and the `currency`

```
struct MonetaryUnit {
    var amount: Double
    let currency: Currency

    init(amount: Double, currency: Currency = .euro) {
        self.amount = amount
        self.currency = currency
    }
}
```

In production code, it's preferable to save monetary units as `Int`, as adding/subtracting with `Double` or `Float` could lead to rounding errors. We want to keep it simple here 😉

Solution 1.2: Create classes and structs needed for the Model

- We want to be able to convert from USD to EUR and vice versa. Add a computed property to `Currency`:

```
var factorToDollar: Double {
    switch self {
    case .dollar:
        return 1.0
    case .euro:
        return 1.143
    }
}
```

- We can then do the conversion in `MonetaryUnit`:

```
func translate(toType type: Currency) -> MonetaryUnit {
    return MonetaryUnit(amount: self.amount * self.currency.factorToDollar /
                                type.factorToDollar,
                        currency: type)
}
```

Solution 1.2: Create classes and structs needed for the Model

- Let's create another file named Transaction.swift. The transaction consists of a value represented by a monetary unit, a description and a date

```
struct Transaction {

    // Stored Properties
    var id: Int
    var value: MonetaryUnit
    var description: String?
    var date: Date
    var accountId: Int

    // Initializer
    init(value: MonetaryUnit, description: String?, date: Date, accountId: Int) {
        self.id = 0
        self.value = value
        self.description = description
        self.date = date
        self.accountId = accountId
    }
}
```

Why are we using a **struct** here?

We will set this properly later.

Solution 1.2: Create classes and structs needed for the Model

- To compare different transactions and store transactions in a Set we need `Transaction` to conform to the `Equatable` and `Hashable` protocol.
- Add the following extensions to `Transaction`:


```
extension Transaction: Equatable {
    static func ==(lhs: Transaction, rhs: Transaction) -> Bool {
        return lhs.id == rhs.id
    }
}
extension Transaction: Hashable {
    var hashValue: Int {
        return id
    }
}
```

Solution 1.2: Create classes and structs needed for the Model

- We need one more class: an `Account`. Add another file and add the properties and initializer:

Why are we using a **class** here?

Why do we use a **set**?

```
class Account {

    var id: Int
    var name: String
    var transactions: Set<Transaction> = []

    // Initializer
    init(id: Int, name: String) {
        self.id = id
        self.name = name
    }
}
```

Solution 1.2: Create classes and structs needed for the Model

- We also want to be able to compute the balance, which sums up all transactions belonging to the account and converts them to a specified currency.
- Add the following method to `Account`:

We use argument labels for readability!

```
func balance(ofType currencyType: Currency = .euro) -> MonetaryUnit {
    var balance = 0.0
    for transaction in transactions {
        balance += transaction.value.amount
    }
    return MonetaryUnit(amount: balance, currency: currencyType)
}
```

Solution 1.2: Create classes and structs needed for the Model

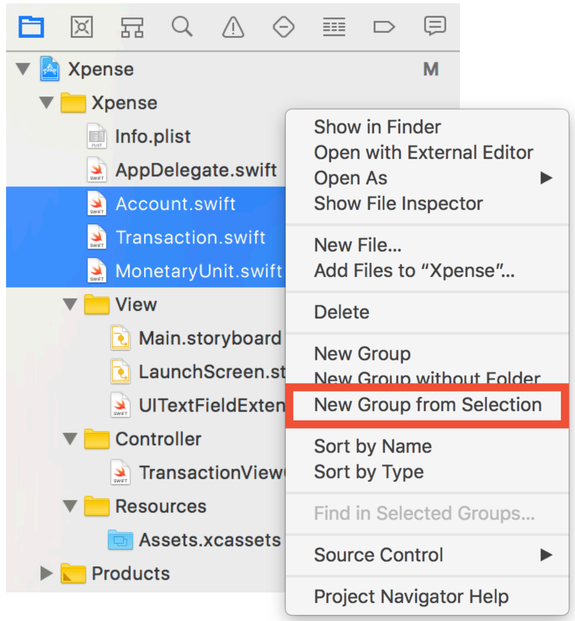
- To compare different accounts and store accounts in a **Set** we also need `Account` to conform to the `Equatable` and `Hashable` protocol.
- Add the following extensions to `Account`:

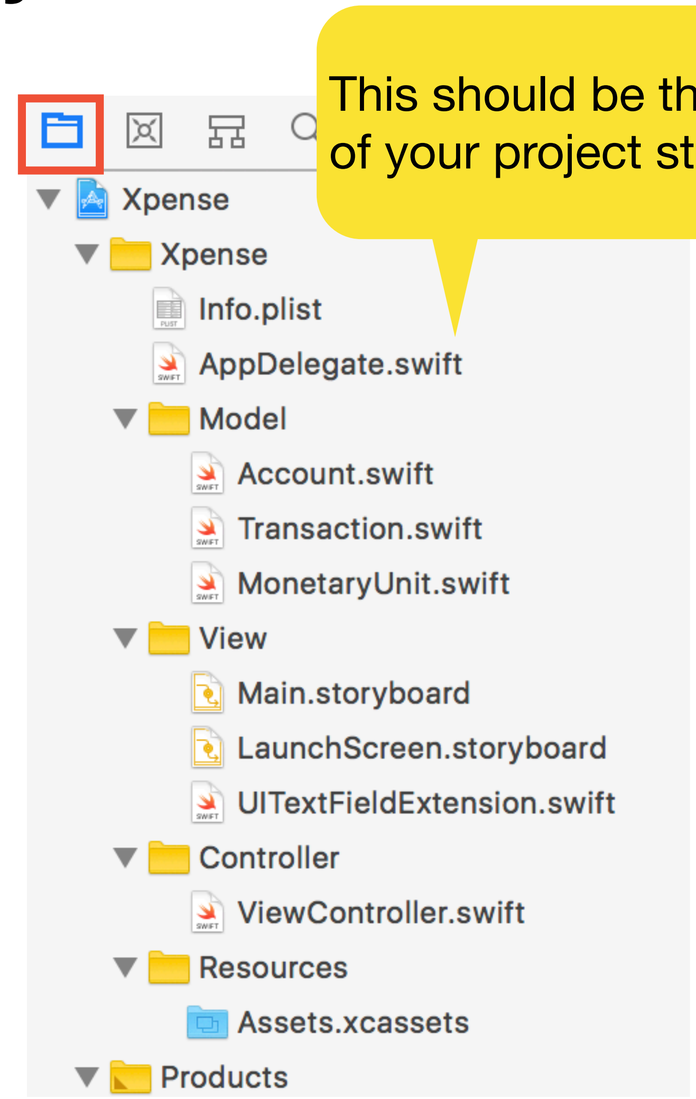
```
extension Account: Equatable {
    static func ==(lhs: Account, rhs: Account) -> Bool {
        return lhs.id == rhs.id
    }
}
extension Account: Hashable {
    var hashValue: Int {
        return id
    }
}
```

You can copy the implementation of Equatable and Hashable from Transaction.

Solution 1.3: Clean up your project structure

- Move the files into a new folder named "`Model`"
  - Select the three files
  - Right click and choose New Group from Selection
- Tidy up the project structure by moving the folders to their logical place





Exercise 2: Connect Model and View

Task 2.1: Add an Account property to your ViewController

Task 2.2: Use the Model when adding a transaction

Task 2.3: Get the balance from the Account

Solution 2.1: Add an Account property to your ViewController

- We will add a static property to our ViewController

```
private var account = Account(id: 1, name: "Wallet")
```

- Head to the `saveTransaction` method to use our newly created Account:

```
@IBAction func saveTransaction() {
    guard let transactionAmountString = transactionAmountField.text,
        let transactionDescriptionString = transactionDescriptionField.text,
        let transactionAmount = Double(transactionAmountString),
        let balanceLabelString = balanceLabel.text,
        let balance = Double(balanceLabelString) else {
            return
    }

    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                    -transactionAmount : transactionAmount

    balanceLabel.text = String(amount + balance)

    // ...
}
```

How do we locate a method again?

We can get rid of these lines because we are not getting the balance from the label any more

This will also be calculated with the newly created Account

Solution 2.2: Use the Model when adding a transaction

```
@IBAction func saveTransaction() {
    guard let transactionAmountString = transactionAmountField.text,
        let transactionDescriptionString = transactionDescriptionField.text,
        let transactionAmount = Double(transactionAmountString),
        let balanceLabelString = balanceLabel.text,
        let balance = Double(balanceLabelString)    else {
            return
    }

    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                    -transactionAmount : transactionAmount

    balanceLabel.text = String(amount + balance)

    transactionAmountField.resignFirstResponder()
    transactionDescriptionField.resignFirstResponder()
    showSavedAlert()
    resetViewController()
}
```

Solution 2.2: Use the Model when adding a transaction

```
@IBAction func saveTransaction() {
    guard let transactionAmountString = transactionAmountField.text,
        let transactionDescriptionString = transactionDescriptionField.text,
        let transactionAmount = Double(transactionAmountString) else {
            return
    }



    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                    -transactionAmount : transactionAmount



    transactionAmountField.resignFirstResponder()
    transactionDescriptionField.resignFirstResponder()
    showSavedAlert()
    resetViewController()
}
```

Solution 2.2: Use the Model when adding a transaction

```
@IBAction func saveTransaction() {
    guard let transactionAmountString = transactionAmountField.text
        let transactionDescriptionString = transa
        let transactionAmount = Double(transactio
            return
    }

    let description = transactionDescriptionString == "" ?
                                nil : transactionDescriptionString
    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                    -transactionAmount : transactionAmount



    transactionAmountField.resignFirstResponder()
    transactionDescriptionField.resignFirstResponder()
    showSavedAlert()
    resetViewController()
}
```

An empty text field will give "" as a value; we want to assign nil

Solution 2.2: Use the Model when adding a transaction

```
@IBAction func saveTransaction() {
    // ...

    let description = transactionDescriptionString == "" ?
                                    nil : transactionDescriptionString
    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                  -transactionAmount : transactionAmount

    let newTransaction = Transaction(value: MonetaryUnit(amount: amount),
                                             description: description,
                                             date: transactionDatePicker.date,
                                             accountId: account.id)
    account.transactions.insert(newTransaction)

    transacti                    signFirstResponder()
    transacti                    ld.resignFirstResponder()
    showSavedAlert()
    resetViewController()
}
```

…and use the new Account class!

Use the Model: create a Transaction

Solution 2.2: Use the Model when adding a transaction

```
@IBAction func saveTransaction() {
    // ...

    let description = transactionDescriptionString == "" ?
                                    nil : transactionDescriptionString
    let amount = transactionTypeSegmentedControl.selectedSegmentIndex == 0 ?
                    -transactionAmount : transactionAmount
    let newTransaction = Transaction(value: MonetaryUnit(amount: amount),
                                             description: description,
                                             date: transactionDatePicker.date,
                                             accountId: account.id)
    account.transactions.insert(newTransaction)
    balanceLabel.text = String(describing: account.balance())

    transactionAm        nFirstResponder()
    transactionDe        esignFirstResponder()
    showSavedAlert()
    resetViewController()
}
```

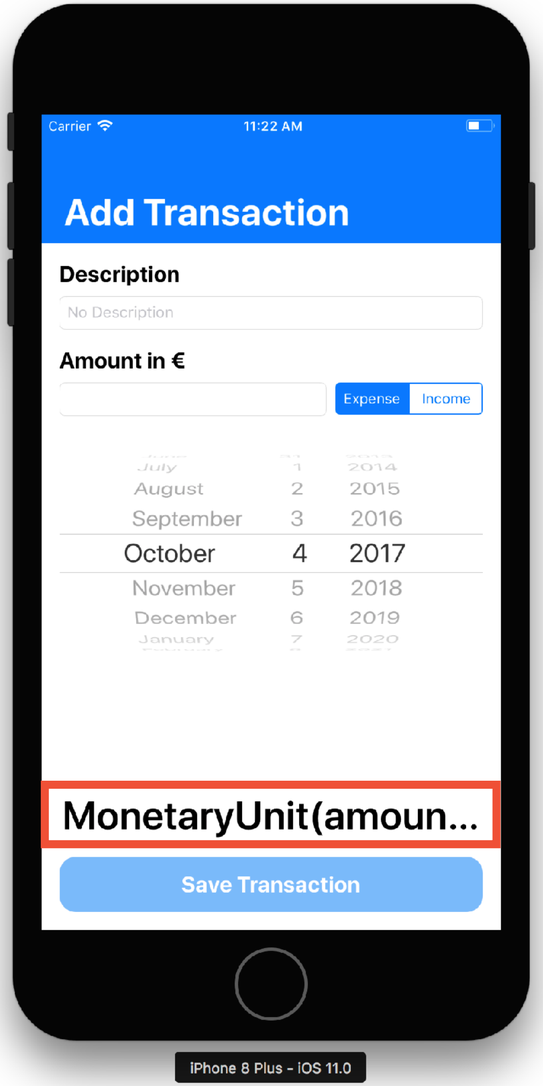
Finally, update the balance label.

Solution 2.3: Get the balance from the Account

- Run your app. Why is the label value behaving strangely?
- `String(describing: )` does not know how to create a String representation of our object! We need to specify this.
- Add the following to `MonetaryUnit`

More on this common protocol in **Swift 03**

```
extension MonetaryUnit: CustomStringConvertible {
    var description: String {
        return String(format: "%.2f", amount)
                    + currency.rawValue
    }
}
```



Overview

- Model-View-Controller
  - Motivation
  - Application in Cocoa Touch
- MVC implementation in the Xpense app
- **Persistent on-device data storage**
- UI Improvements
- Intermediate Interface Builder
  - Adding a tab bar and an additional scene
  - Examining the view hierarchy

Persistent on-device data storage

- Swift 4 has new functionality which enables you to easily code and decode your data types for compatibility with external representations like JSON
  - Useful for e.g. client-server communication or
  - Persistent storage of data on device
- To make a custom type support this, you have to declare conformance to the `Codable` protocol, which combines the `Encodable` and `Decodable` protocols
  - All of its properties have to also conform to Codable
  - Standard library types (String, Int, …) and Foundation types (Date, URL, …) conform to Codable
- You can then use `JSONDecoder` and `JSONEncoder` to decode and encode instances that conform to Codable to JSON

Exercise 3: Persist transactions on the device

Task 3.1: Make your custom types conform to Codable

Task 3.2: Set up a data handler class

Task 3.3: Read to and write from a JSON file

Task 3.4: Trigger reading and saving the data at the right places

Solution 3.1: Make your custom types conform to Codable

- Head to the `Transaction` struct.

```
struct Transaction {
    var id: Int
    var value: MonetaryUnit
    var description: String?
    var date: Date
    var accountId: Int

    // ...
}
```

- This type can now be encoded and decoded!

Solution 3.1: Make your custom types conform to Codable

- Head to the `Transaction` struct.

```
struct Transaction: Codable {
    var id: Int
    var value: MonetaryUnit
    var description: String?
    var date: Date
    var accountId: Int

    // ...
}
```

Not all properties are Codable; Let's convert MonetaryUnit as well!

- This type can now be encoded and decoded!

Solution 3.1: Make your custom types conform to Codable

- Open `MonetaryUnit.swift` and make both `Currency` and `MonetaryUnit` conform to the protocol as well:

```
enum Currency: String, Codable {
    // ...
}

struct MonetaryUnit: Codable {
    // ...
}
```

Since all properties are Codable, this is all we need to do.

- Finally, the `Account` class:

```
class Account: Codable {
    // ...
}
```

Solution 3.2: Set up a data handler class

- We want to have a separate class to handle reading from and writing to the JSON file
- Create a new Swift class called `DataHandler` in the Model folder
- We can now move the Account used in the app to this class
- In the `DataHandler` class add an type variable named accounts of type **Set** containing instances of `Account`

It's good practice

We also convert it to an set to support multiple accounts.

```
import Foundation

class DataHandler {
    static var accounts: Set<Account> = []
}
```

We define a type variable in the DataHandler

Solution 3.2: Set up a data handler class

- Remove the private variable `account` from your `ViewController`
- You should now get build errors (build using ⌘ + B) in the `saveTransaction` method of your ViewController.
- Add a line to the `guard` statement:

```
guard let transactionAmountString = transactionAmountField.text,
      let transactionDescriptionString = transactionDescriptionField.text,
      let transactionAmount = Double(transactionAmountString) else {
    return
}
```

Solution 3.2: Set up a data handler class

- Remove the private variable `account` from your `ViewController`
- You should now get build errors (build using ⌘ + B) in the `saveTransaction` method of your ViewController.
- Add a line to the `guard` statement:

```
guard let transactionAmountString = transactionAmountField.text,
      let transactionDescriptionString = transactionDescriptionField.text,
      let transactionAmount = Double(transactionAmountString),
      let account = DataHandler.accounts.first else {
    return
}
```

We access the account of the DataHandler

Solution 3.2: Set up a data handler class

- We want to store some basic settings in our data handler.
- Add a struct inside the class:

```
private struct Constants {

    static let fileName = "Account.json"

    static var localStorageURL: URL {
        let documentsDirectory = FileManager().urls(for: .documentDirectory,
                                                    in: .userDomainMask).first!
        return documentsDirectory.appendingPathComponent(Constants.fileName)
    }
}
```

We want to store our data in the standard directory. The document directory should always be present when running the App on iOS, therefor we may use force unwrap here.

Solution 3.2: Set up a data handler class

- We also need a method to generate a new Transaction ID
- Add a computed property inside the `DataHandler` class

```
static var nextTransactionID: Int {
    var largestId = 0
    for account in accounts {
        for transaction in account.transactions
                            where transaction.id > largestId {
                largestId = transaction.id
        }
    }
    return largestId + 1
}
```

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data

- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    let fileWrapper = FileWrapper(url: Constants.localStorageURL,
                                  options: .immediate)
}
```

You will get an error.

Call can throw, but it is not marked with 'try' and the error is not handled

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data

- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    let fileWrapper = try FileWrapper(url: Constants.localStorageURL,
                                      options: .immediate)
}
```

What? Another error?

Errors thrown from here are not handled

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data

- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    do {
        let fileWrapper = try FileWrapper(url: Constants.localStorageURL,
                                          options: .immediate)
    } catch _
        print                                    uses empty account")
    }
}
```

This is not a very elegant way to catch errors.
You we will learn more about this in detail in **Swift 03**.

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data
- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    do {
        let fileWrapper = try FileWrapper(url: Constants.localStorageURL,
                                          options: .immediate)
        guard let data = fileWrapper.regularFileContents else {
            throw NSError()
        }


    } catch _ {
        print("Could not load Account, DataHandler uses empty account")
    }
}
```

Check if the file is corrupt

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data
- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    do {
        let fileWrapper = try FileWrapper(url: Constants.localStorageURL,
                                          options: .immediate)
        guard let data = fileWrapper.regularFileContents else {
            throw NSError()
        }
        accounts = try JSONDecoder().decode(Set<Account>.self, from: data)

    } catch _ {
        print("Could not load Account, DataHandler uses empty account")
    }
}
```

Finally, the decoding!

Solution 3.3: Read to and write from a JSON file

- We need two methods: one to load data, and another to write it to a file and save it.
- Let's start with a method for loading the data
- Add the following method to the `DataHandler` class:

```
static func loadFromJSON() {
    do {
        let fileWrapper = try FileWrapper(url: Constants.localStorageURL,
                                          options: .immediate)
        guard let data = fileWrapper.regularFileContents else {
            throw NSError()
        }
        accounts = try JSONDecoder().decode(Set<A
        print("Decoded \(accounts.count) accounts.")
    } catch _ {
        print("Could not load Account, DataHandler uses empty account")
    }
}
```

Add a log statement for convenience

Solution 3.3: Read to and write from a JSON file

- Now, saving the data:

```
static func saveToJSON() {
    do {
        let data = try JSONEncoder().encode(accounts)
        let jsonFileWrapper = FileWrapper(regularFileWithContents: data)
        try jsonFileWrapper.write(to: Constants.localStorageURL,
                                  options: FileWrapper.WritingOptions.atomic,
                                  originalContentsURL: nil)

        print("Saved account.")
    } catch _ {
        print("Could not save Account")
    }
}
```

Encode your Account

Write to the file

Again, this is not how you should catch errors in a productive app 😉

See **Swift 03.**

Resulting code in DataHandler.swift - 01

```
import Foundation

class DataHandler {

    private struct Constants {
        static let fileName = "Account.json"
        static var localStorageURL: URL {
            let DocumentsDirectory = FileManager().urls(for: .documentDirectory, in: .userDomainMask).first!
            return DocumentsDirectory.appendingPathComponent(Constants.fileName)
        }
    }

    static var account: Account = Account()

    static var nextTransactionID: Int {
        var largestId = 0
        for account in accounts {
            for transaction in account.transactions where transaction.id > largestId {
                largestId = transaction.id
            }
        }
        return largestId + 1
    }

   // ...
```

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Resulting code in DataHandler.swift - 02

```
    // ...

    static func loadFromJSON() {
        do {
            let fileWrapper = try FileWrapper(url: Constants.localStorageURL, options: .immediate)
            guard let data = fileWrapper.regularFileContents else {
                throw NSError()
            }
            accounts = try JSONDecoder().decode(Set<Account>.self, from: data)
            print("Decoded \(accounts.count) accounts.")
        } catch _ {
            print("Could not load Account, DataHandler uses empty account")
        }
    }

    static func saveToJSON() {
        do {
            let data = try JSONEncoder().encode(accounts)
            let jsonFileWrapper = FileWrapper(regularFileWithContents: data)
            try jsonFileWrapper.write(to: Constants.localStorageURL,
                                      options: FileWrapper.WritingOptions.atomic,
                                      originalContentsURL: nil)
            print("Saved account.")
        } catch _ {
            print("Could not save Account")
        }
    }
}
```

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Solution 3.4: Trigger reading and saving the data at the right places

- We want to load the data when the app is launched. The common way to do that is in the `didFinishLaunchingWithOptions` method of your `AppDelegate`:

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

    DataHandler.loadFromJSON()
    if DataHandler.accounts.isEmpty {
        DataHandler.accounts.insert(Account(id: 1, name: "Wallet"))
    }

    return true
}
```

Add these lines

Create the default account on start if none exists

Solution 3.4: Trigger reading and saving the data at the right places

- We can now also properly set the Transaction ID. Go to the initializer of `Transaction` and change the line where the `id` is set to use the computed property of `DataHandler`:

```
self.id = DataHandler.nextTransactionID
```

- Finally, trigger a save of the data whenever a Transaction is saved. Add the following line to the end of the `saveTransaction` method in `ViewController`:

```
DataHandler.saveToJSON()
```

Save the data when the transactions change

Solution 3.4: Trigger reading and saving the data at the right places

- The last thing we need to do is set the `balanceLabel` in the `ViewController` to the current value after we reopen the app as we load the account when the app launches
- To achieve this add the following lines in the `viewDidLoad` method in the `ViewController` above calling the `resetViewController()` method:

```
if let account = DataHandler.accounts.first {
    balanceLabel.text = String(describing: account.balance())
}

resetViewController()
```

More advanced JSON parsing

- You now know how to save objects to JSON and how to load them back into your app.
- But what about client-server communication? Possible problems can be:
  - Key names on the server do not match those on the client
  - Complex data formats like dates, images
  - Complex object structure (inheritance, nested objects, …)
- You will need at least some of this in the iPraktikum!
- A lot of these are explained in this extensive tutorial or this playground provided by Apple.
- More information about client-server communication in **Server Side Swift**

Overview

- Model-View-Controller
  - Motivation
  - Application in Cocoa Touch
- MVC implementation in the Xpense app
- Persistent on-device data storage
- **UI Improvements**
- Intermediate Interface Builder
  - Adding a tab bar and an additional scene
  - Examining the view hierarchy

UI Improvements for better user feedback

- A nicely designed user interface should react to user input. In particular, it should tell the user when their input is wrong.
- In our case, we do this with the following changes:
  - Color the amount field red if the input is wrong
  - Deactivate the Save button if the user provided wrong or insufficient data

Exercise 4: Improve the UI to react to user input

Task 4.1: Color the amount field red on incorrect input

Task 4.2: Deactivate the save button on incorrect data

Solution 4.1: Color the amount field red on incorrect input

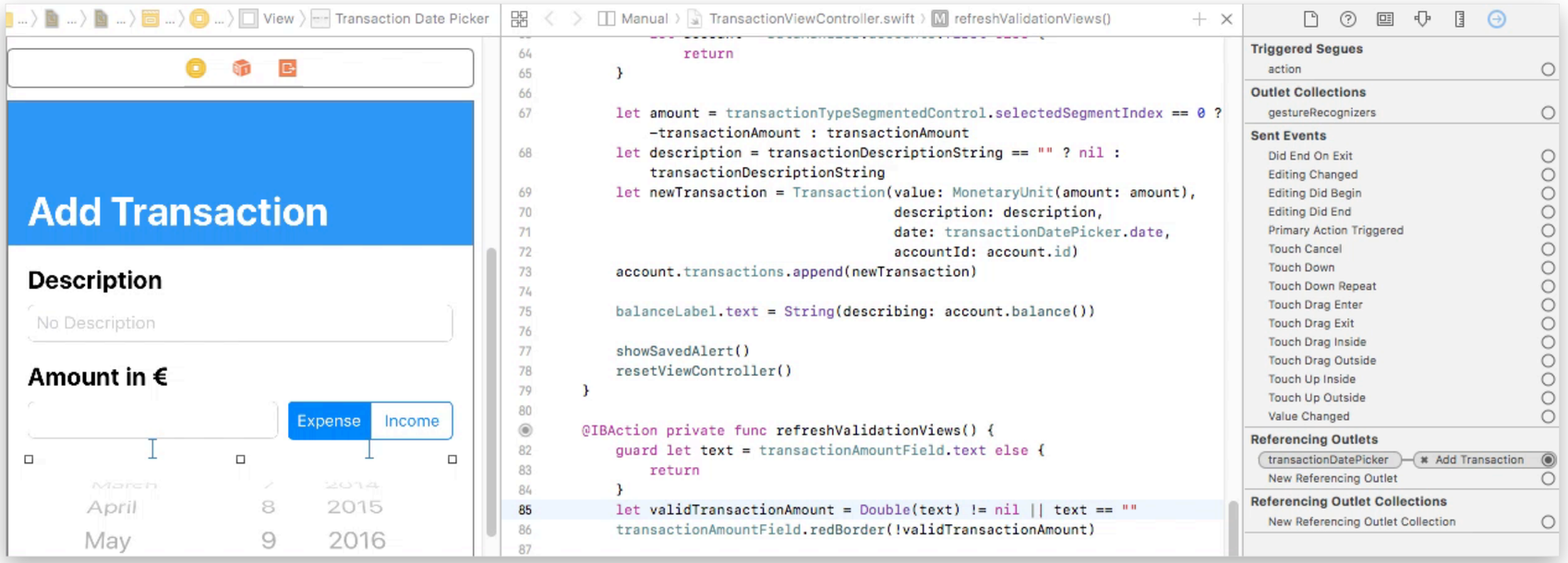
- We provided you with an Extension to `UITextField` which contains a method to color its border red based on a boolean value (check `UITextFieldExtension.swift` to see the implementation)
- We will define an IBAction which will be called every time the amount field value has changed
- Add the following at the end of `ViewController` (before the first extension):

```
@IBAction private func refreshValidationViews() {
    guard let text = transactionAmountField.text else {
        return
    }
    let validTransactionAmount = Double(text) != nil || text == ""
    transactionAmountField.redBorder(!validTransactionAmount)
}
```

Check if the input is a number (or empty).

Solution 4.1: Color the amount field red on incorrect input

- We want to execute this action every time the field's value changes
- An easy way to achieve this is to connect it to a sent event in Interface Builder
  - Open `Main.Storyboard` and `ViewController.swift` side-by-side in the Assistant Editor
  - Select the `Transaction Amount Field` and switch to the **Connections Inspector** in the right sidebar
  - Connect the **Editing Changed** event under **Sent Events** to your action in the code



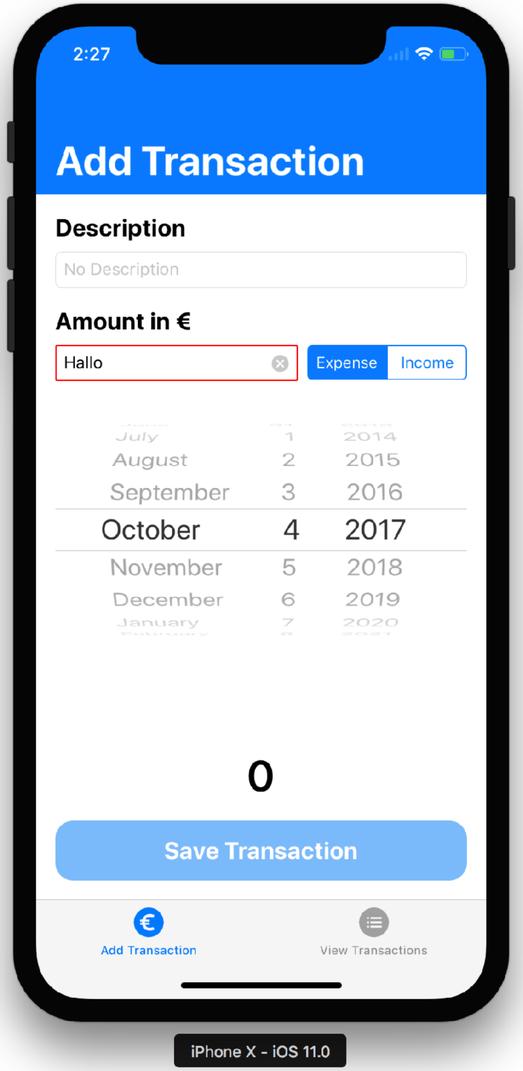
Solution 4.1: Color the amount field red on incorrect input

- We also want to call this action in `resetViewController`:

```
private func resetViewController() {
    transactionDescriptionField.text = ""
    transactionAmountField.text = ""

    refreshValidationViews()
}
```

- Your text field is now colored in red if the user types in text. It is reset to its normal state once the value is corrected (e.g. the user empties the field).
- As the iPhone keyboard blocks this kind of entry, this only applies in the simulator, or if the user pastes e.g. letters from the clipboard into the field. But you can use the same principle in your app to implement more relevant UI feedback!



Solution 4.2: Deactivate the save button on incorrect data

- A transaction should only be saved if the user has entered an amount, and this amount is valid. Otherwise, we want to deactivate the Save button
- In order to signal this to the user, we will also make it half-translucent
- We want to save this value as a constant

Why? It's good practice.

- Extend the `struct` named `Constants` in `ViewController`:

Add this line. The value means 50% translucence.

```
struct Constants {
    static let saveButtonDisabledAlpha = CGFloat(0.5)
    static let hideSaveTransactionButtonLevel = CGFloat(230)
    static let hidetransactionDatePickerLevel
                                    = hideSaveTransactionButtonLevel + 150
}
```

Solution 4.2: Deactivate the save button on incorrect data

- Locate the refreshValidationViews method we added in the previous step and add the necessary checks to enable/disable the save button:

```
@IBAction private func refreshValidationViews() {
    guard let text = transactionAmountField.text else {
        return
    }
    let validTransactionAmount = Double(text) != nil || text == ""
    transactionAmountField.redBorder(!validTransactionAmount)

    let enableSave = validTransactionAmount && text != ""
    saveTransactionButton.isEnabled = enableSave
    saveTransactionButton.alpha = enableSave
                                ? 1.0 : Constants.saveButtonDisabledAlpha
}
```

If disabled, the button will not respond to touch events.

We signal this to the user by making the button translucent.

Solution 4.2: Deactivate the save button on incorrect data

- While the user can't use the button once it's deactivated, they could still save the transaction by hitting the return key on the keyboard.
- Locate the `textFieldShouldReturn` method (in the `UITextFieldDelegate` extension) and add a check to see whether the transaction should be saved:

```
extension ViewController: UITextFieldDelegate {
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        textField.resignFirstResponder()
        if saveTransactionButton.isEnabled {
            saveTransaction()
        }
        return true
    }
}
```

This line was there before!

Overview

- Model-View-Controller
  - Motivation
  - Application in Cocoa Touch
- MVC implementation in the Xpense app
- Persistent on-device data storage
- UI Improvements
- **Intermediate Interface Builder**
  - **Adding a tab bar and an additional scene**
  - Examining the view hierarchy

Exercise 5: Add a Tab Bar to the application

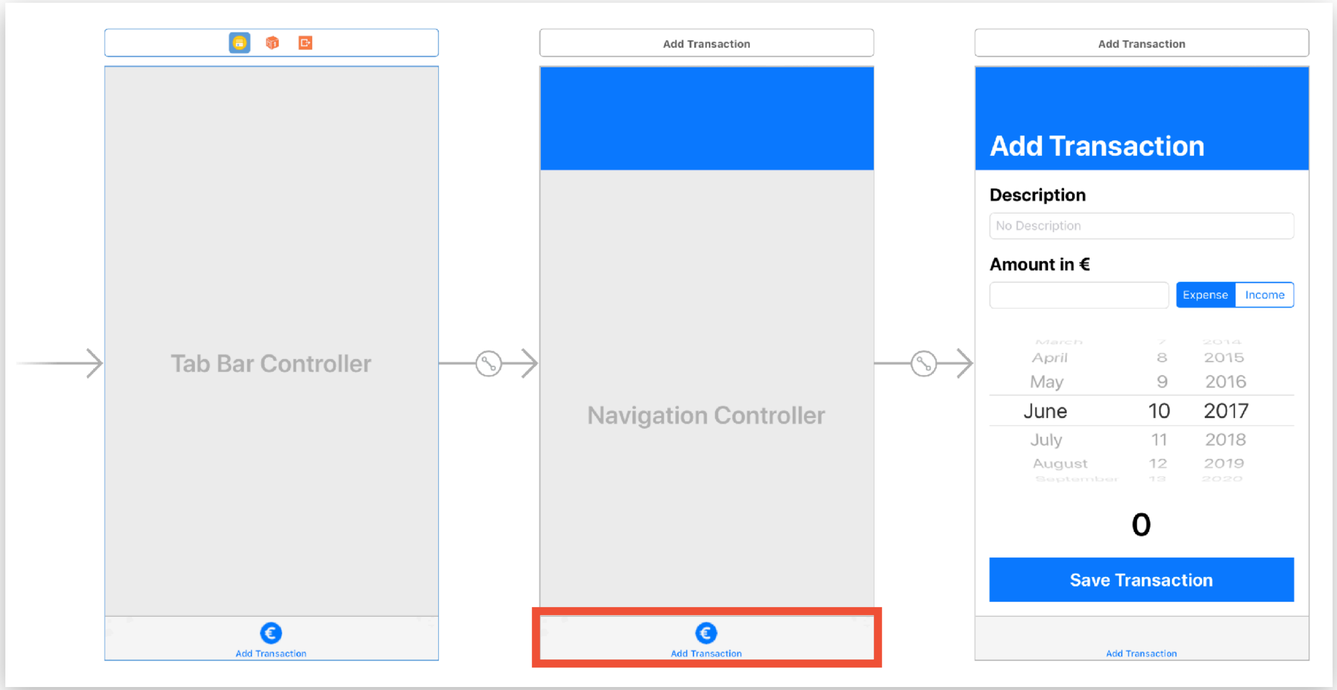
Task 5.1: Embed the existing Scenes into a Tab Bar Controller

Task 5.2: Change the Tab Bar icons

Task 5.3: Rename your existing ViewController

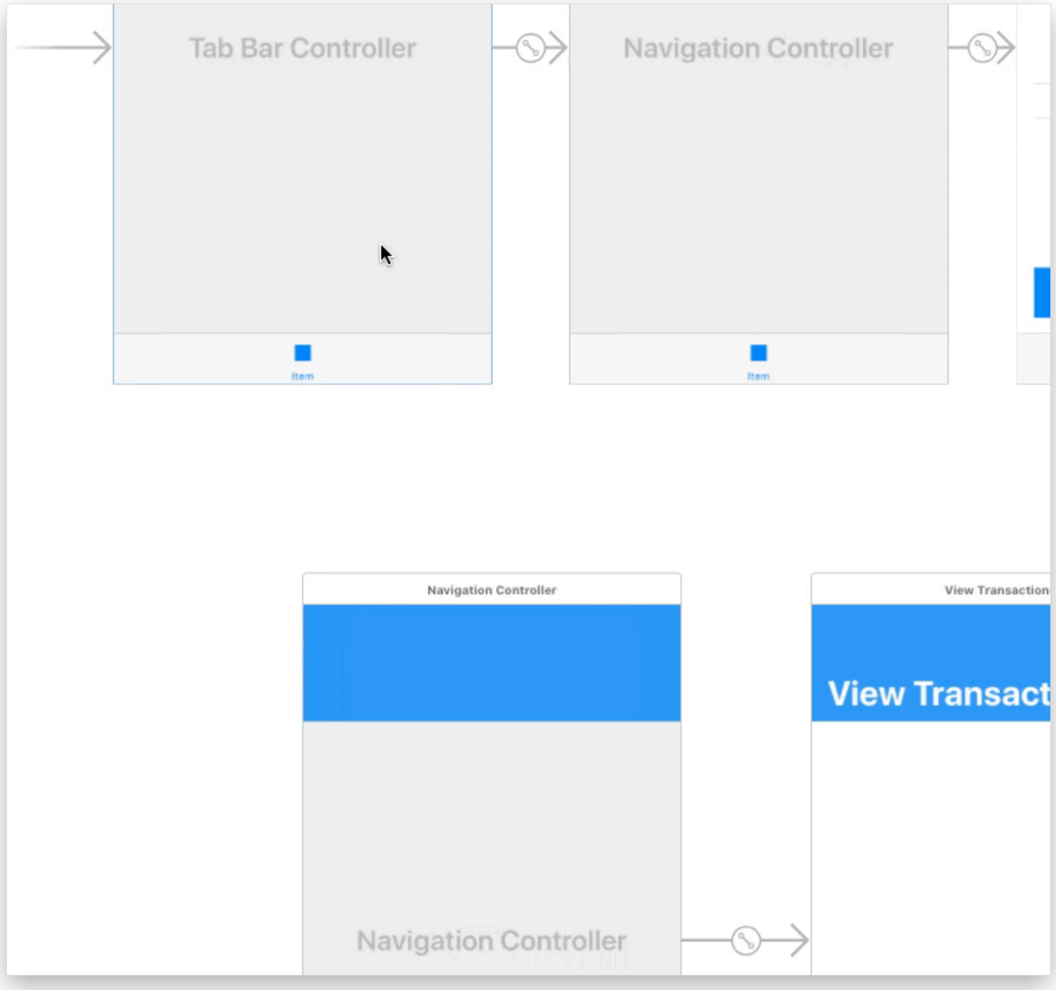
Solution 5.1: Embed the existing Scenes into a Tab Bar Controller

- Open your Storyboard and select the top Navigation Controller
- Choose Editor > Embed In > Tab Bar Controller
- You now have a Tab Bar with one Tab Bar Item



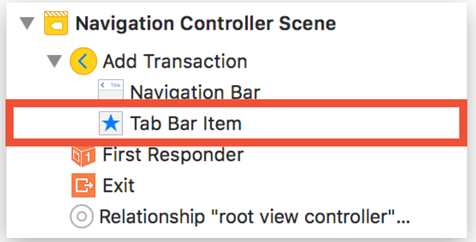
Solution 5.1: Embed the existing Scenes into a Tab Bar Controller

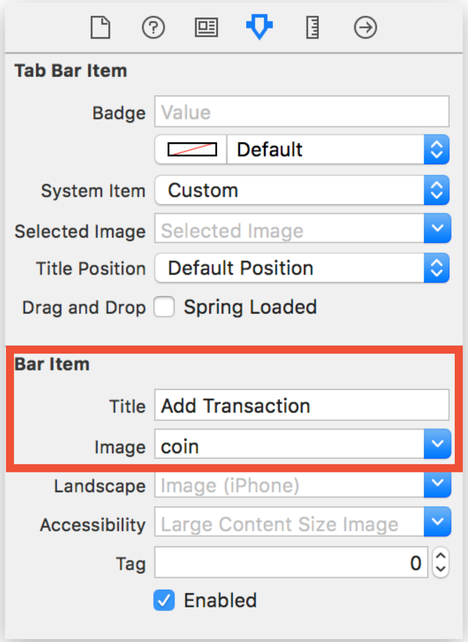
- Right click on the Tab Bar Controller and drag to the bottom Navigation Controller
- Choose Relationship Segue > view controllers
- Notice that a second Tab Bar Item just appeared

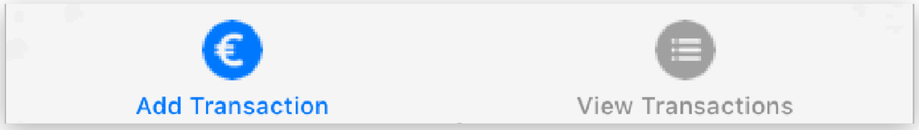


Solution 5.2: Change the Tab Bar icons

- In your Document Outline, you now have two Scenes named `Item Scene`
- Select the Tab Bar Item named `Item` in the first scene. Check which Navigation Controller is selected.
- In the Attributes Inspector, you need to set the **Title** and **Image** of the Bar Item.
  - Values for the top View Controller: `Add Transaction` and `coin`
  - Values for the bottom View Controller: `View Transactions` and `list`







These are **Assets** that we prepared for you.

Solution 5.3: Rename your existing ViewController

- We will soon have a second View Controller, so we should give our existing one a more descriptive name.
- Head to `ViewController.swift` and right click on the class name `ViewController`. Choose **Refactor > Rename...**
- Rename it to `TransactionViewController` and hit **Enter**

File Name: TransactionViewController.swift

```
1   //
2   //  TransactionViewController.swift
3   //  Xpense
10
11  // MARK: - TransactionViewController
12  class TransactionViewController: UIViewController {
13      // MARK: Constants
112 // MARK: - Extension: UITextFieldDelegate
113 extension TransactionViewController: UITextFieldDelegate {
114     func textFieldShouldReturn(_ textField: UITextField) -> Bool {
124 // MARK: - Extension: React to Keyboard
125 extension TransactionViewController {
126     private func keyboardParameterFrom(notification: NSNotification) -> (keyboardHeight: CGFloat, duration:
            Double, animatorOption: UIViewAnimationOptions)? {
```

Main.storyboard

Transaction View Controller

Class Name: TransactionViewController

The file name will be changed automatically.

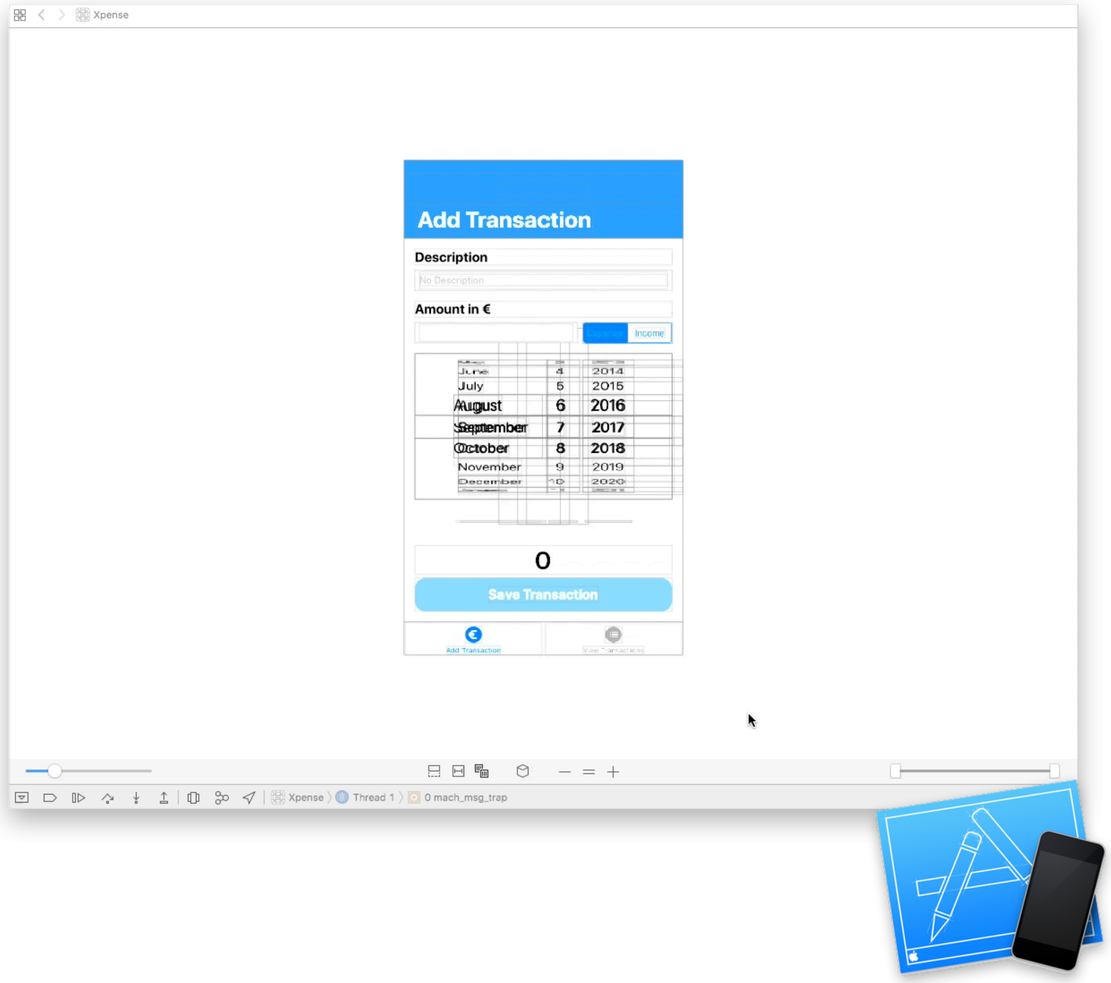
Xcode even renames the Storyboard reference for you!

Overview

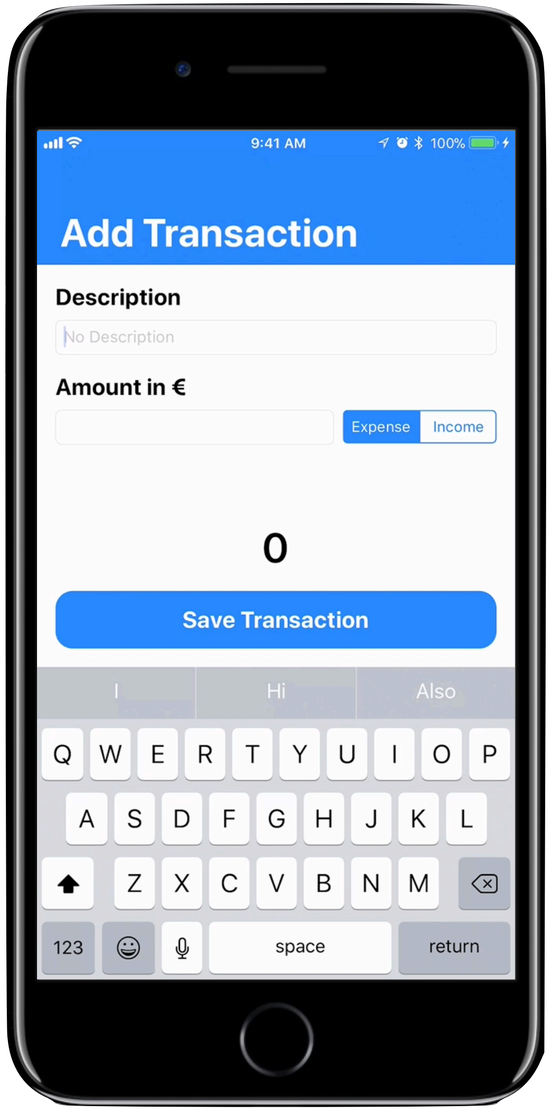
- Model-View-Controller
  - Motivation
  - Application in Cocoa Touch
- MVC implementation in the Xpense app
- Persistent on-device data storage
- UI Improvements
- Intermediate Interface Builder
  - Adding a tab bar and an additional scene
  - **Examining the view hierarchy**

Examining the view hierarchy in Xcode

- While your app is running, click the  button above the Debug Area to inspect the current Scene and its elements
- You can rotate the rendering (click & drag), play with the spacing of the views, set which views are visible, reveal constraints, ….
- Note that the app will be paused while you do this! Click on  to continue.



Why does the Save Button move above the keyboard?



Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

```
override func viewDidLoad() {
    super.viewDidLoad()
    transactionDescriptionField.de
    transactionAmountField.delegat

    NotificationCenter.default.addObserver(self,
                                           selector: #selector(keyboardWillShow(notification:)),
                                           name: .UIKeyboardWillShow,
                                           object: nil)
    NotificationCenter.default.addObserver(self,
                                           selector: #selector(keyboardWillHide(notification:)),
                                           name: .UIKeyboardWillHide,
                                           object: nil)

    rese
}
```

The NotificationCenter is a central notification dispatch mechanism that enables the broadcast of information to registered observers.

Each running Cocoa program has a default notification center which offers the possibility to get notified about certain events such as the keyboard appearing/disappearing

Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

- What does `#selector` mean? A selector is the name used to **select a method to execute** for an object, or the unique identifier that replaces the name when the source code is compiled
- The only thing that makes the selector method name different from a plain string is that the compiler makes sure that selectors are unique
- So `#selector(keyboardWillShow(notification:))` in the `addObserver` functions means, that a method `keyboardWillShow` will be called when the keyboard will appear
- As the selector syntax is a relict from Objective-C till here for compatibility, we need to mark the `keyboardWillShow` function with the `@objc` keyword

Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

```
@objc func keyboardWillShow(notification: NSNotification) {
    if let parameter = keyboardParameterFrom(notification: notification) {
        UIView.animate(withDuration: parameter.duration,
                       delay: 0.0,
                       options: parameter.animatorO
                       animations: {
            self.additionalSafeAreaInsets.bottom =
            self.additionalSafeAreaInsets.bottom = parameter.keyboardHeight
                       - self.view.safeAreaInsets.bottom
            let remainingHeight = self.view.frame.size.height
                       - self.view.safeAreaInsets.bottom - self.view.safeAreaInsets.top
            self.saveTransactionButton.isHidden =
                       remainingHeight < Constants.hideSaveTransactionButtonLevel
            self.transactionDatePicker.isHidden =
                       remainingHeight < Constants.hidetransactionDatePickerLevel
            self.view.layoutIfNeeded()
        })
    }
}
```

The `keyboardWillShow` method calls the helper method `keyboardParameterFrom`

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

```
private func keyboardParameterFrom(notification: NSNotification) -> (keyboardHeight: CGFloat,
                                                                     duration: Double,
                                                                     animatorOption: UIViewAnimationOptions)? {
    guard let keyboardSize = notification.userInfo?[UIKeyboardFrameEndUserInfoKey]
                                                                      as? CGRect,
          let animationTime = notification.userInfo?[UIKeyboardAnimationDurationUserInfoKey]
                                                                     as? NSNumber,
          let animationCurve = notification.userInfo?[UIKeyboardAnimationCurveUserInfoKey]
                                                                     as? NSNumber else {
        return nil
    }
    let animationOptions = UIViewAnimationOptions(rawValue: UInt(animationCurve.intValue<<16))
    return (keyboardSize.height, animationTime.doubleValue, animationOptions)
}
```

- This helper function extracts the `keyboardSize`, the `animationTime` of the keyboard animation as well as the `animationCurve` of the animation and returns them as a tuple

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

```
@objc func keyboardWillShow(notification: NSNotification) {
    if let parameter = keyboardParameterFrom(notification: notification) {
        UIView.animate(withDuration: parameter.duration,
                       delay: 0.0,
                       options: parameter.animatorOption ,
                       animations: {
            self.additionalSafeAreaInsets.bottom = 0.0
            self.additionalSafeAreaInsets.bottom = parameter.keyboardHeight
                                                   ottom
            let remaini                            ize.height
                                                   ottom - self.view.safeAreaInsets.top
            self.saveTransactionButton.isHidden =
                remainingHeight < Constants.hideSaveTransactionButtonLevel
            self.transactionDatePicker.isHidden =
                remainingHeight < Constants.hidetransactionDatePickerLevel
            self.view.layoutIfNeeded()
        })
    }
}
```

Usage of the closure syntax, more on this in **Swift 03**

We call the type method of UIView that animates the changes done in the animations block

Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

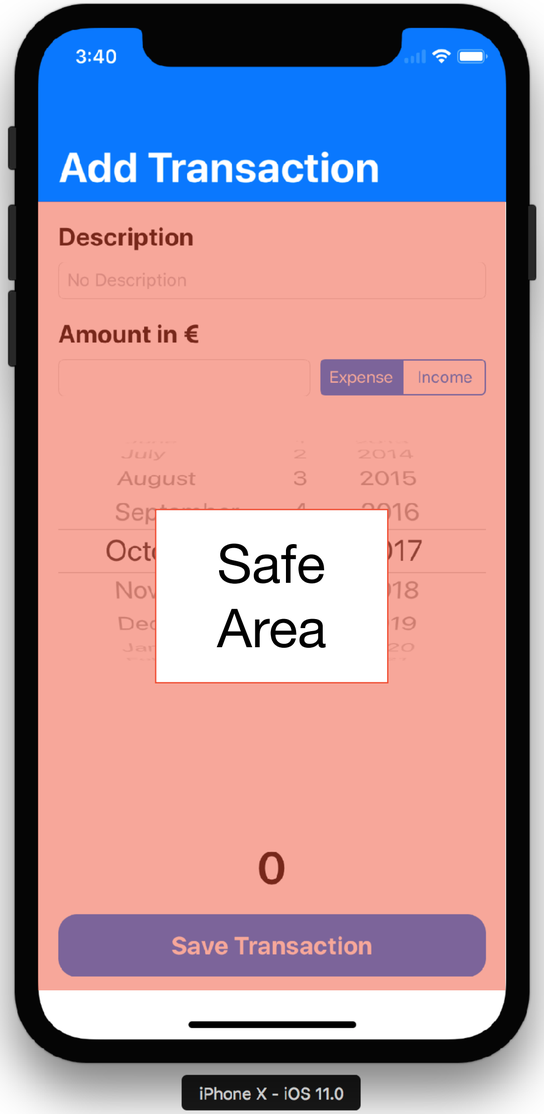
```
@objc func keyboardWillShow(notification: NSNotification) {
    if let parameter = keyboardParameterFrom(notification: notification) {
        UIView.animate(withDuration: parameter.duration,
                       delay: 0.0,
                       options: parameter.animatorOption ,
                       animations: {
            self.additionalSafeAreaInsets.bottom = 0.0
            self.additionalSafeAreaInsets.bottom = parameter.keyboardHeight
                       - self.view.safeAreaInsets.bottom
            let remainingHeight = self.view.frame.size.height
                       - s                                    AreaInsets.top
            self.saveTrans
                       re                                     onLevel
            self.transacti
                       remainingHeight < Constants.hidetransactionDatePickerLevel
            self.view.layoutIfNeeded()
        })
    }
}
```

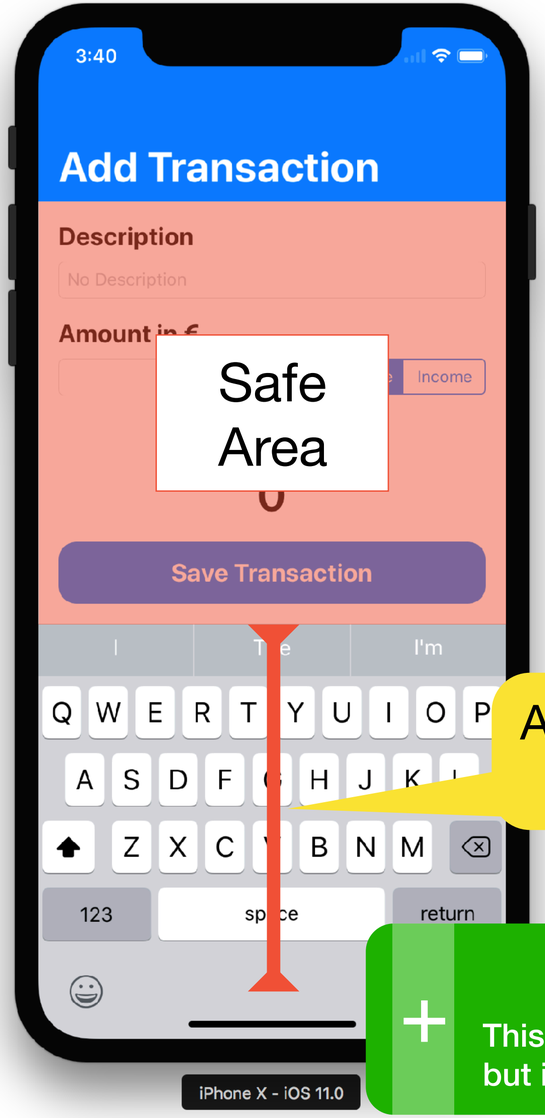
We remove the current `additionalSafeAreaInsets` at the bottom and add an inset at the size of the keyboard minus all fixed `safeAreaInsets` at the bottom

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?





Why does the Save Button move above the keyboard?

```
@objc func keyboardWillShow(notification: NSNotification) {
    if let parameter = keyboardParameterFrom(notification: notification) {
        UIView.animate(withDuration: parameter.duration,
                       delay: 0.0,
                       options: parameter.animatorOption ,
                       animations: {
            self.additionalSafeAreaInsets.bottom = 0.0
            self.additionalSafeAreaInsets.bottom = parameter.keyboardHeight
                       - self.view.safeAreaInsets.bottom
            let remainingHeight = self.view.frame.size.height
                       - self.view.safeAreaInsets.bottom - self.view.safeAreaInsets.top
            self.saveTransactionButton.isHidden =
                       remainingHeight < Constants.hideSaveTransactionButtonLevel
            self.transactionDatePicker.isHidden =
                       remainingHeight < Constants.hidetransactionDatePickerLevel
            self.view.layoutIfNeeded()
        })
    }
}
```

Depending on how much space is left, we hide the `saveTransactionButton` or `transactionDatePicker`

Additional slide

This slide is not covered in the course, but included for your future reference.

Why does the Save Button move above the keyboard?

```
@objc func keyboardWillHide(notification: NSNotification) {
    if let parameter = keyboardParameterFrom(notification: notification) {
        UIView.animate(withDuration: parameter.duration,
                       delay: 0.0,
                       options: parameter.animatorOption ,
                       animations: {
            self.additionalSafeAreaInsets.bottom = 0.0
            self.saveTransactionButton.isHidden = false
            self.transactionDatePicker.isHidden = false
            self.view.layoutIfNeeded()
        })
    }
}
```

In the keyboardWillHide we set the `additionalSafeAreaInsets` at the bottom to 0.0 and unhide all potentially hidden elements

Additional slide

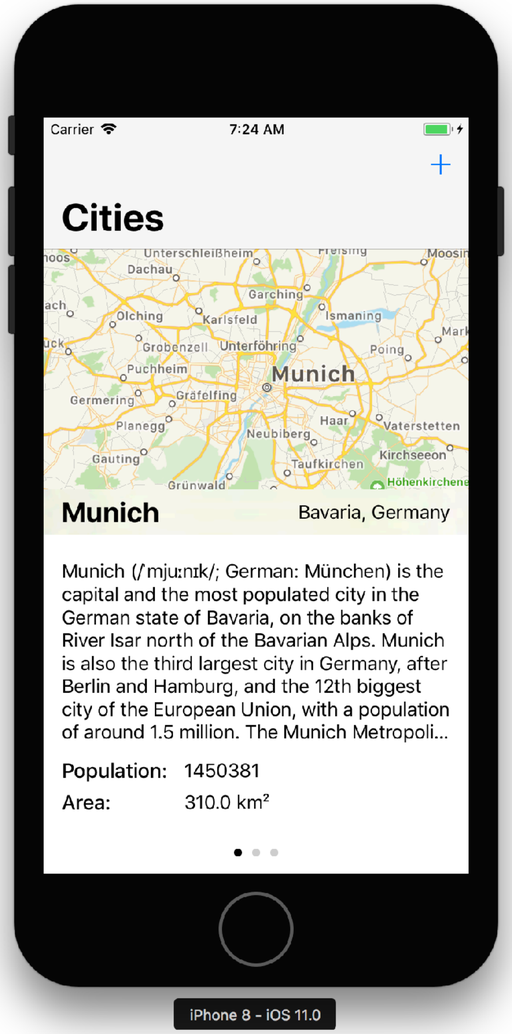
This slide is not covered in the course, but included for your future reference.

Summary

- You learned about the MVC architectural style and how it is applied in iOS
- You wrote Model classes and learned how to use them in your app
- You learned how to apply Codable and JSONEncoder and Decoder to persistently save your Model on the device
- You got to know advanced UI and Interface Builder features
  - Giving UI feedback to the user in case of invalid input
  - Adding a Tab Bar to your app

Homework

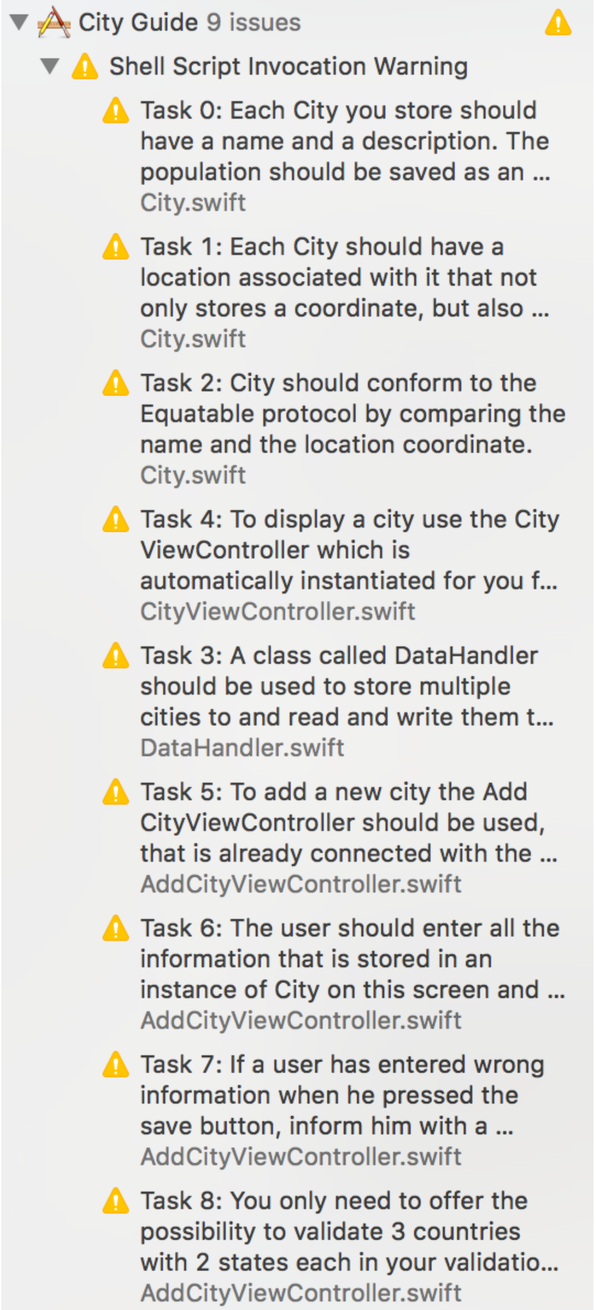
- We will create an application to keep track of the different cities you want to visit. We have already provided you with part of the UI in a project called **“City Guide”.**
- The “City Guide” project already has a basic UI structure centered around a `UIPageViewController` that will be used to display the different cities by swiping left and right.
- The `CitiesViewController` is a subclass of `UIPageViewController` and handles all the heavy work for you. You can take a look at this code but there is no need to change it! 😉
- Your task is to create a model to store and load cities from JSON using the `Codable` protocol, improve the UI to display a city and create a view to add new cities as well as connect the UI with the model.
- To get to know the code in the City Guide project, structure it using the Model-View-Controller folder structure.



Homework

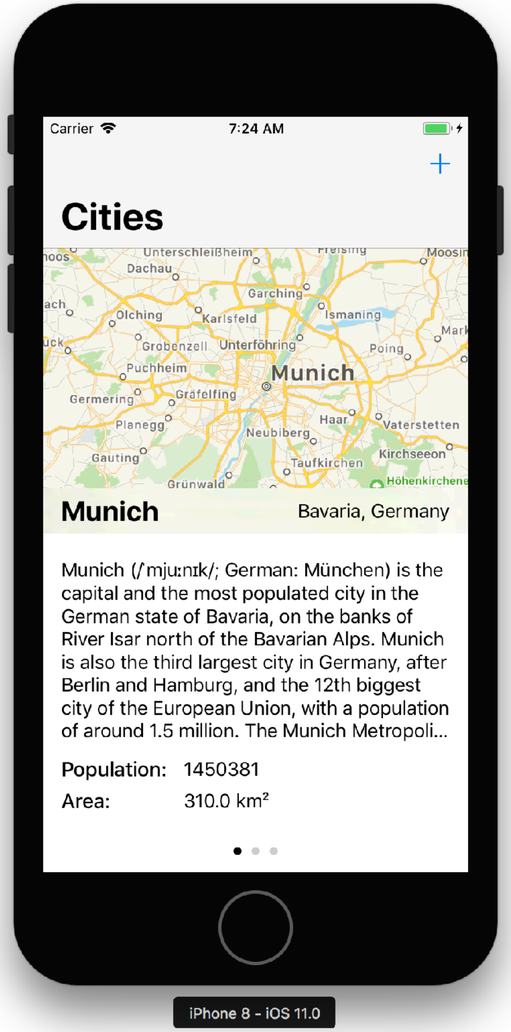
As soon as you restructured the project, use the WARNINGS displayed in the project to complete the homework.

If you can’t find the warnings, use ⌘ + B to rebuild the project



Homework

- Each `City` you store should have a `name` and a `description`. The `population` should be saved as an Integer value and the `area` of the city using a floating point number that stores the area in km².
- Each City should have a `location` associated with it that not only stores a `coordinate`, but also the `country` and `state` the `City` is located in. `Location` and `Coordinate` should be structures, `Country` and `State` should be enumerations.
- City should conform to the `Equatable` protocol by comparing the `name` and the `location coordinate`.
- A class called `DataHandler` should be used to store multiple cities to and read and write them to JSON.
- To display a city use the `CityViewController` which is automatically instantiated for you for every city by the `CitiesViewController`. The layout should be similar as in the screenshot and the layout should adapt the to different display sizes. If there is not enough space on a device, truncate the description. The map should b centered on the city using the method provided in `MKMapViewExtension`.



Additional slide

This slide is not covered in the course, but included for your future reference.

- To add a new city the `AddCityViewController` should be used, that is already connected with the other elements of the UI using Bar Button Items. Create a UI using labels, text fields (and slider, segmented controls or picker if you feel comfortable with it).
- The user should enter all the information that is stored in an instance of `City` on this screen and you should validate his input so that only correct data is entered.
  - A state should be located in the country that is entered
  - When a number is expected only a number should be accepted and when a string is accepted only a string should be stored in the model
- If a user has entered wrong information when he pressed the save button, inform him with a `UIAlertView` describing the wrong input.
- You only need to offer the possibility to validate 3 countries with 2 states each in your validation logic. Use computed properties and extensions when implementing the validation. Unknown countries, states and parties can be denied. (Normally not a good practice but acceptable for this example 🤓)

+ Additional slide

This slide is not covered in the course, but included for your future reference.

Submission

- Create a **branch** using the from the button in your JIRA Task called "Solution S04"
- **Check out** the branch in SourceTree
- Create a new folder "Solution S04" in your repository using the Finder
- **Develop** your solution inside this folder
- **Commit** and **push** your solution
- Create a **Pull Request** (right click / Ctrl+click on the branch in SourceTree)
- Add your **tutor** as a **reviewer**
- Implement your tutor's **feedback** and commit again
- **Merge** the Pull Request
- **Delete** the branch in your local repository

Further Material

- MVC in iOS: a modern approach - Ray Wenderlich
- What’s new in Foundation - WWDC 2017
- Encoding and Decoding Custom Types - Apple
- Ultimate Guide to JSON Parsing With Swift 4 - SwiftJSON