

# Swift Sync and Async Error Handling

iOS Conf Singapore - October 2015



– @Javi

# Agenda

- Traditional Asynchronous Code
- Problems with Traditional Error Handling
  - Result VS throws
- Modeling asynchronous computation: Future
- Making Result and throws play together

# Traditional asynchronous code

# Traditional asynchronous code

```
struct User { let avatarURL: NSURL }
func requestUserInfo(userID: String, completion: (User?, NSError?) -> ())
func downloadImage(URL: NSURL, completion: (UIImage?, NSError?) -> ())

func loadAvatar(userID: String, completion: (UIImage?, NSError?) -> ()) {
    requestUserInfo(userID) { user, error in
        if let user = user {
            downloadImage(user.avatarURL) { avatar, error in
                if let avatar = avatar { completion(avatar, nil) }
                else { completion(nil, error!) }
            }
        } else { completion(nil, error!) }
    }
}
```

# Traditional asynchronous code

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^a-zA-Z\d{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Passwords must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



# Traditional asynchronous code

```
func downloadImage(URL: NSURL, completion: (UIImage?, NSError?) -> ())
```

(.Some, .None)  
.None, .Some)  
**(.Some, .Some)**  
**(.None, .None)**

(.Some, .Some)  
(.None, .None)



# Error Handling

```
func downloadImage(URL: NSURL, completion: (UIImage?, NSError?) -> ())  
  
downloadImage(url) { imageOrNil, errorOrNil in  
    if error = errorOrNil {  
        // What if image is also not nil??  
    }  
    else if image = imageOrNil {  
    }  
}
```

# Error Handling

```
var error: NSError?  
let string = NSString(contentsOfFile:path encoding:NSUTF8StringEncoding error:&error)  
  
if string == nil {  
    // Oops:  
    if error.code == NSFileReadNoSuchFileError {  
        // ...  
    }  
    else if ...  
}
```

# Microsoft Visual Basic

Run-time error '6':

Overflow

Continue

End

Debug

Help

# Microsoft Money



An error has occurred but the error message cannot be retrieved due to another error.

OK

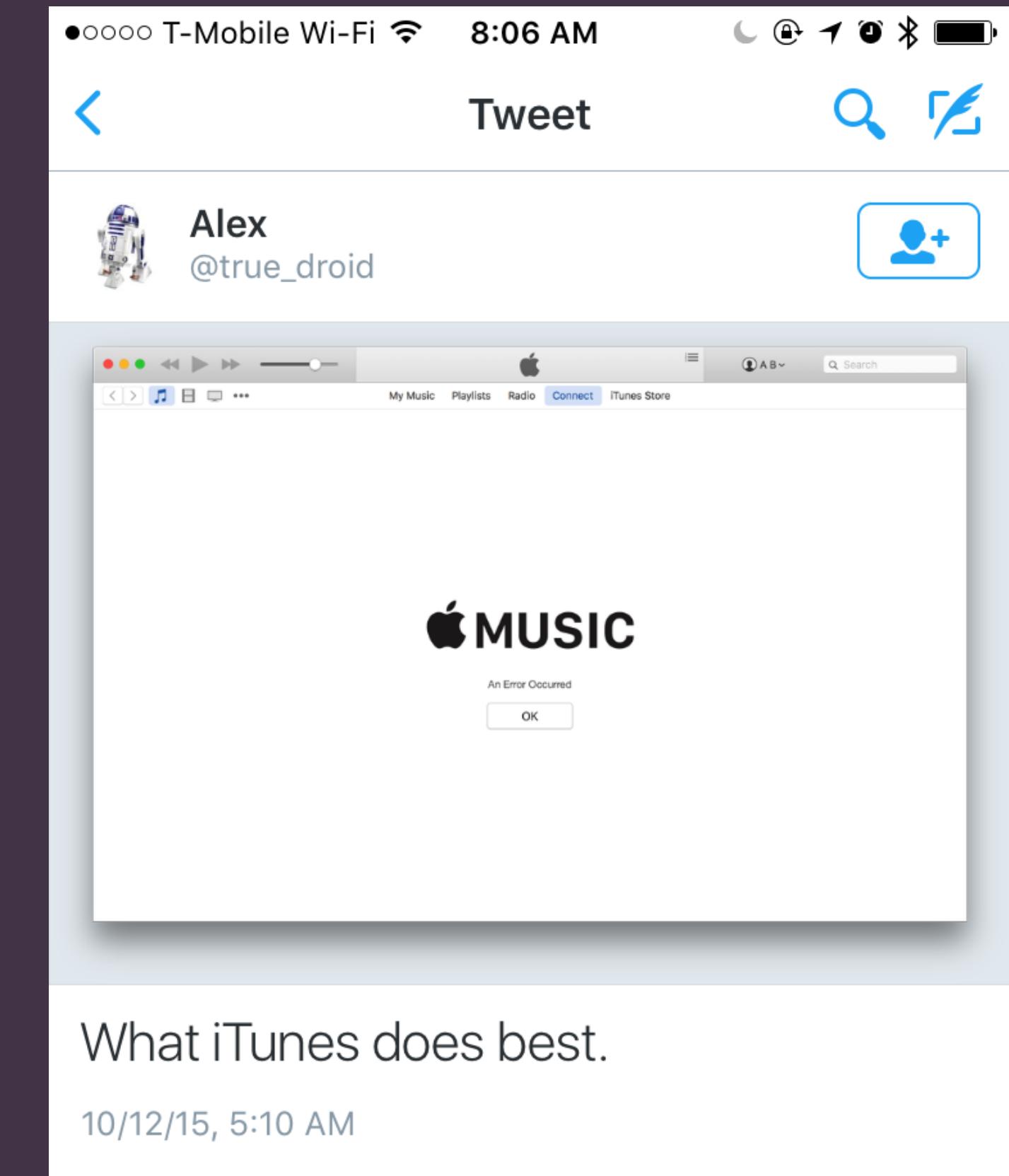
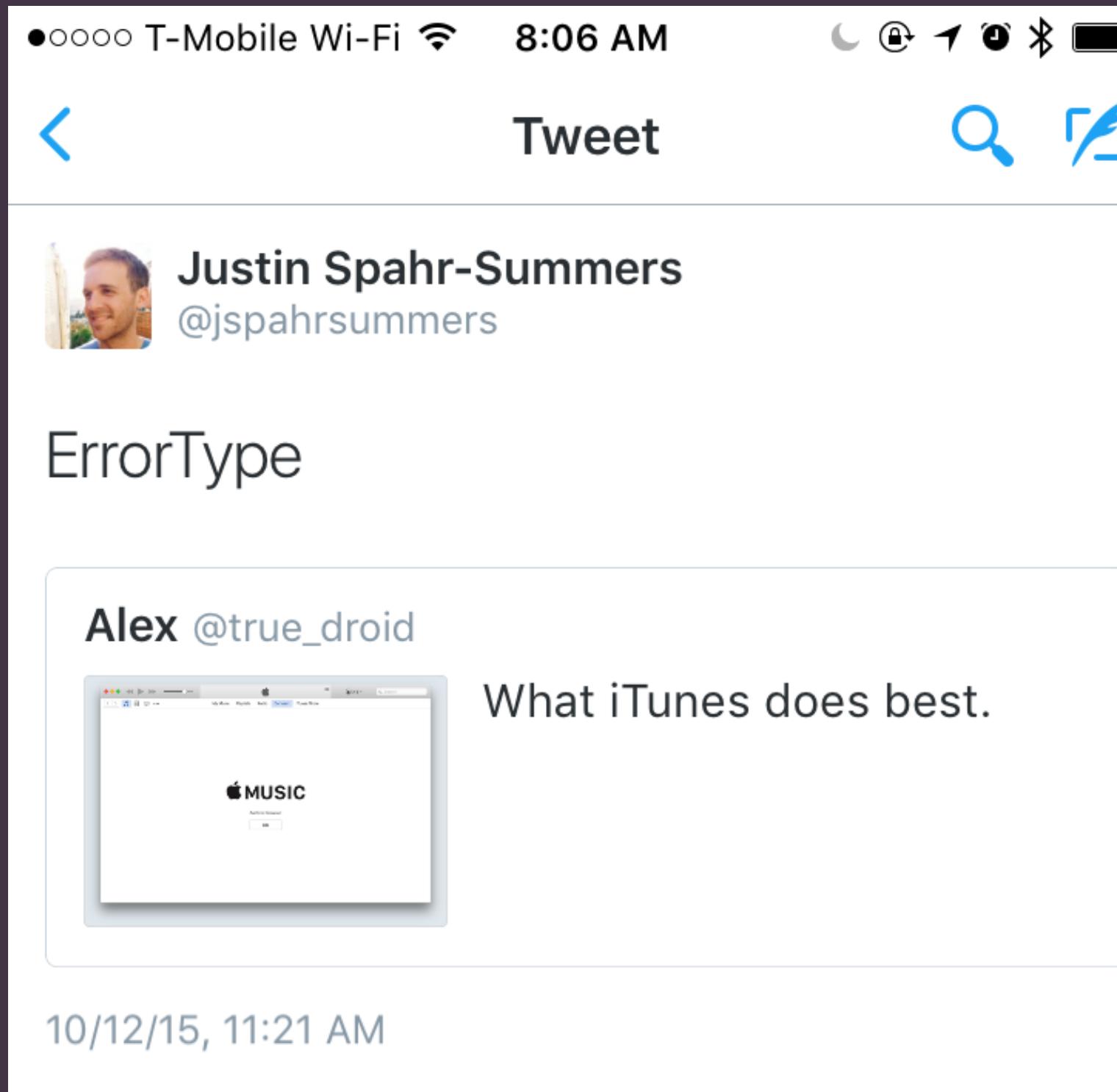
# Error

X



The operation completed successfully

OK



# NSError Alternative

```
protocol ErrorType { }

enum UserInfoErrorDomain: ErrorType {
    case UserDoesNotExist
    case UserRequestFailure(reason: String)
    case NetworkRequestFailure(underlyingError: NSError)
}

extension NSError: ErrorType { }
```

# NSError Alternative

```
enum NoError: ErrorType { }
```

```
let error = NoError(?)
```

# Result

```
enum Result<T, E: ErrorType> {  
    case Success(T)  
    case Failure(E)  
}
```

Result by Rob Rix:  
<https://github.com/antitypical/Result>

# Swift's throws

# Swift's throws

```
func readFile(path: String) throws -> NSData
```

# Swift's throws

```
func readFile(path: String) throws -> NSData  
// throws... what exactly?
```

## **Swift's throws**

```
func readFile(path: String) throws -> NSData
```

**Functionally equivalent to**

```
func readFile(path: String) -> Result<NSData, ErrorType>
```

# Swift's throws

```
func readFile(path: String) throws -> NSData  
  
do {  
    let contents = try readFile("path/to/my/file")  
}  
catch(SomeTypeOfError) {  
    // How can I know which errors I should match..?  
}  
catch {  
    // catch-all block.  
}
```

# Swift's throws

```
func readFile(path: String) throws -> NSData
```

```
let contents: NSData? = try? readFile("path/to/my/file")
```

# Swift's throws

```
func readFile(path: String) -> Result<NSData, FileIOError>

let file = readFile("path/to/my/file")

switch file {
    case let .Success(value):
        // ...
    case let .Failure(error):
        // This `error` variable has type information!
}
```

# Swift's throws

***Hopefully in a future version of Swift ...***

```
func readFile(path: String) throws FileIOError -> NSData
```

# Error handling in Asynchronous APIs

Future<T, E>

# Futures

- Encapsulate a deferred computation.
- Treat values that incur a delay to be retrieved as if they were regular values.
- Allow us to treat errors as first class citizens.
  - Easily composable.

# Future

```
struct Future<T, E: ErrorType> {  
    typealias ResultType = Result<T, E>  
    typealias Completion = ResultType -> ()  
    typealias AsyncOperation = Completion -> ()  
  
    private let operation: AsyncOperation  
}
```

# Future

```
struct Future<T, E: ErrorType> {
    init(operation: AsyncOperation) {
        self.operation = operation
    }

    func start(completion: Completion) {
        self.operation() { result in
            completion(result)
        }
    }
}
```

# Future.map(): transforming the computed value

# Future.map(): transforming the computed value

```
struct User { let avatarURL: NSURL }

func requestUserInfo(userID: String) -> Future<User, ErrorDomain>

func requestUserAvatarURL(userID: String) -> Future<NSURL, ErrorDomain> {
    return requestUserInfo(userID)
        .map { $0.avatarURL }
}
```

# Future.map(): transforming the computed value

```
struct Future<T, E: ErrorType> {  
    func map<U>(f: T -> U) -> Future<U, E>  
}
```

# map() in other types

```
struct Array<T> {  
    func map<U>(f: T -> U) -> [U]  
}
```

```
enum Optional<T> {  
    func map<U>(f: T -> U) -> U?  
}
```

```
struct Future<T, E: ErrorType> {  
    func map<U>(f: T -> U) -> Future<U, E>  
}
```

# Future.map(): transforming the computed value

```
func map<U>(f: T -> U) -> Future<U, E> {  
    // Return a new Future w/ a new operation...  
    return Future<U, E>(operation: { completion in  
        })  
}
```

# Future.map(): transforming the computed value

```
func map<U>(f: T -> U) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
        // Retrieve the value from self...  
        self.start { result in  
  
        }  
    })  
}
```

# Future.map(): transforming the computed value

```
func map<U>(f: T -> U) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
        self.start { result in  
            // Consider .Success and .Failure...  
            switch result {  
  
                case .Success(let value):  
                    let transformed = f(value)  
                    completion(.Success(transformed))  
                case .Failure(let error):  
                    completion(.Failure(error))  
            }  
        }  
    })  
}
```

# Future.map(): transforming the computed value

```
case let .Success(value):  
    // Call completion with the transformed value  
    completion(Result.Success(f(value)))
```

# Future.map(): transforming the computed value

```
case let .Failure(error):  
    // We didn't get a value: no transformation  
    completion(Result.Failure(error))
```

# Future.map(): transforming the computed value

```
func map<U>(f: T -> U) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
        self.start { result in  
            switch result {  
                case let .Success(value):  
                    completion(Result.Success(f(value)))  
                case let .Failure(error):  
                    completion(Result.Failure(error))  
            }  
        }  
    })  
}
```

# Future.andThen(): concatenating async work

# Future.andThen(): concatenating async work

```
func requestUserAvatarURL(userID: String) -> Future<NSURL, ErrorDomain>

func downloadImage(URL: NSURL) -> Future<UIImage, ErrorDomain>

func downloadUserAvatar(userID: String) -> Future<UIImage, ErrorDomain> {
    return requestUserAvatarURL(userID)
        .andThen(downloadImage)
}
```

# Future.andThen(): concatenating async work

```
struct Future<T, E: ErrorType> {  
    func andThen<U>(f: T -> Future<U, E>) -> Future<U, E>  
}
```

# Future.andThen(): concatenating async work

```
func andThen<U>(f: T -> Future<U, E>) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
  
    })  
}
```

# Future.andThen(): concatenating async work

```
func andThen<U>(f: T -> Future<U, E>) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
        self.start { firstFutureResult in  
  
            }  
    })  
}
```

# Future.andThen(): concatenating async work

```
func andThen<U>(f: T -> Future<U, E>) -> Future<U, E> {  
    return Future<U, E>(operation: { completion in  
        self.start { firstFutureResult in  
            switch firstFutureResult {  
                case let .Success(value): // ...  
                case let .Failure(error): // ...  
            }  
        }  
    })  
}
```

# Future.andThen(): concatenating async work

```
case let .Success(value):  
    let nextFuture = f(value)
```

# Future.andThen(): concatenating async work

```
case let .Success(value):  
    let nextFuture = f(value)  
  
    nextFuture.start { finalResult in  
        completion(finalResult)  
    }
```

# Future.andThen(): concatenating async work

```
case let .Failure(error):  
    completion(Result.Failure(error))
```

# Future.andThen(): concatenating async work

```
func andThen<U>(f: T -> Future<U, E>) -> Future<U, E> {
    return Future<U, E>(operation: { completion in
        self.start { firstFutureResult in
            switch firstFutureResult {
                case let .Success(value): f(value).start(completion)
                case let .Failure(error): completion(Result.Failure(error))
            }
        }
    } )
}
```

# We can go from this...

```
func loadAvatar(userID: String, completion: (UIImage?, NSError?) -> ()) {  
    requestUserInfo(userID) { user, error in  
        if let user = user {  
            downloadImage(user.avatarURL) { avatar, error in  
                if let avatar = avatar {  
                    completion(avatar, nil)  
                } else {  
                    completion(nil, error)  
                }  
            }  
        }  
        else { completion(nil, error) }  
    }  
}
```

## ... To this

```
func requestUserInfo(userID: String) -> Future<User, UserInfoErrorDomain>

func downloadImage(URL: NSURL) -> Future<UIImage, UserInfoErrorDomain>

func loadAvatar(userID: String) -> Future<UIImage, UserInfoErrorDomain> {
    return requestUserInfo(userID)
        .map { $0.avatarURL }
        .andThen(downloadImage)
}
```

# Mixing it all up

`Result<T, E>` and `throws`

## Result<T, E> **and** throws

throws => Result

```
extension Result {  
    public init(@autoclosure _ f: () throws -> T) {  
        do {  
            self = .Success(try f())  
        } catch {  
            self = .Failure(error as! Error)  
        }  
    }  
}
```

# Result<T, E> **and** throws

Result => throws

```
extension Result {  
    public func dematerialize() throws -> T {  
        switch self {  
            case .Success(value):  
                return value  
            case let .Failure(error):  
                throw error  
        }  
    }  
}
```

## Result<T, E> **and** throws

Future => throws

```
let avatarFuture = loadAvatar("4815162342")
```

```
avatarFuture.start { result in
    do {
        let avatar = try result.materialize()
    }
    catch {
        // handle `error`
    }
}
```

# Limitations of Futures

- Only represent computation to retrieve one value.
  - Can't encapsulate streams of values.
- Only consumer (pull) driven, not producer (push) driven.

# ReactiveCocoa

## Signals > Futures

# Follow-up Resources

- ReactiveCocoa: <https://github.com/ReactiveCocoa/ReactiveCocoa>
- Railway Oriented Programming: <http://fsharpforfunandprofit.com/posts/recipe-part2/>

# Thanks!

## *Questions?*