

Macoun

Windige Abhängigkeiten

Enno Welbers, Benjamin Böcker

PALASTHOTEL



Ben Böcker

@benboecker

apps@benboecker.de

Enno Welbers

@mkernel

enno.welbers@palasthotel.de



Voraussetzungen

- Grundlegendes iOS-Verständnis
- Swift Protocols & Generics
- Grundkenntnisse Englisch

Ablauf

- Was ist ***Dependency Injection***?
- DI in Swift
- Apple Frameworks und DI

Was ist *Dependency Injection*?

Was ist *Dependency Injection*?

Ein paar Begrifflichkeiten

Komponente

Abhängigkeit

Auflösen

Was ist *Dependency Injection*?

Definition

- Entwurfsmuster der OOP
- Reglementiert Abhängigkeiten eines Objekts zur Laufzeit an einem *zentralen Ort*
- Komponenten benötigen keine Kenntnisse ihrer Umgebung

Was ist *Dependency Injection*?

**Vorteil: Separierung / Entkopplung
einzelner Komponenten**

👍 Wartbarkeit

👍 Einfachere Tests

👍 Wiederverwendbarkeit

Dependency Injection in Swift

Ein naiver Ansatz...

Dependency Injection in Swift




- Kennt alle Komponenten
- Liefert Abhängigkeiten
- Kontrolle über den Lebenszyklus
- Kapseln Funktionalität
- Werden vom Container verwaltet
- Verwalten Abhängigkeiten

Dependency Injection in Swift



```
protocol Component {  
    func resolveDependencies(using container: Container)  
}  
  
protocol Factory {  
    func create() → Component  
}
```

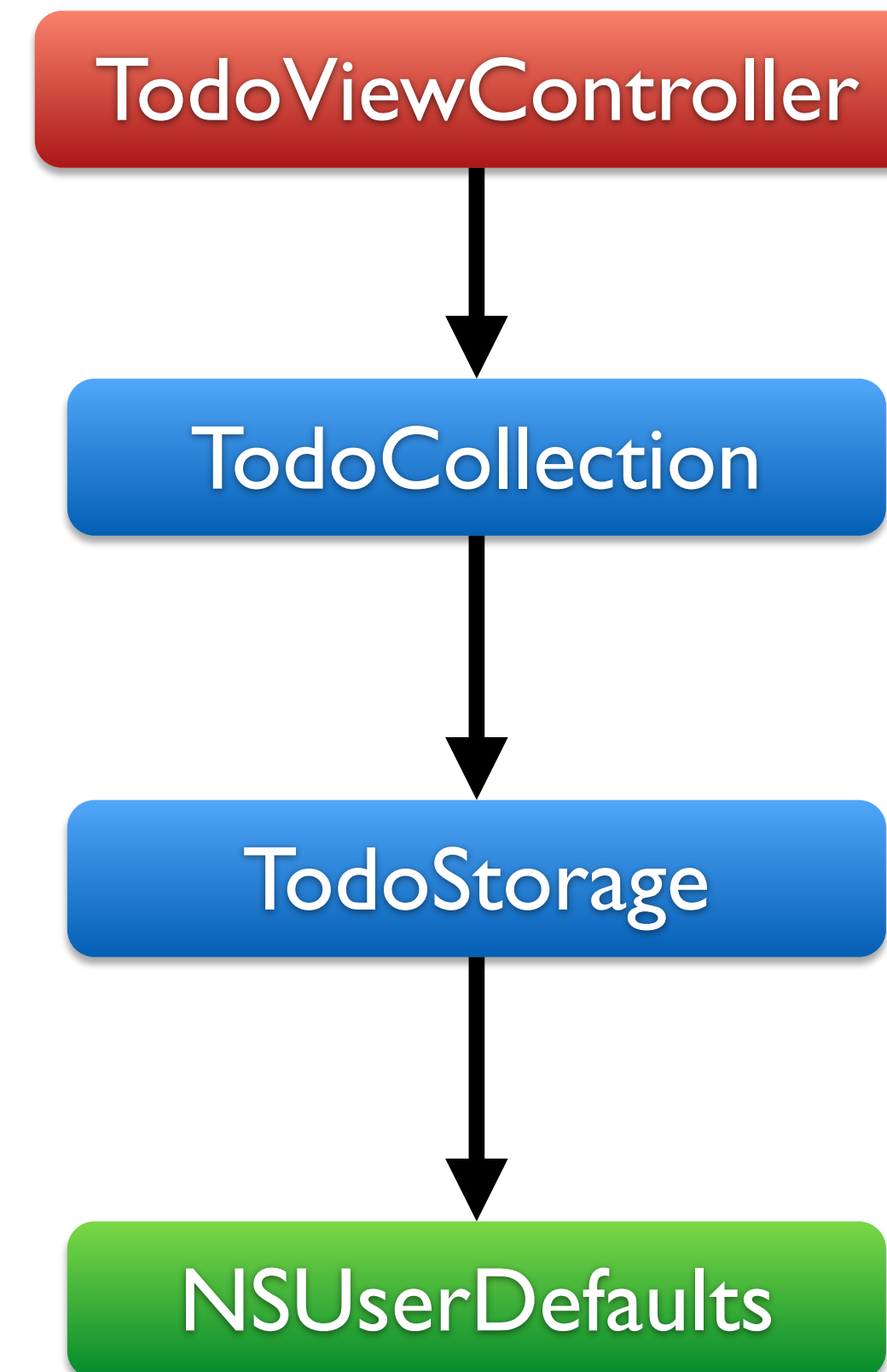


```
1 class Container {
2     var factories: [String: Factory] = [:]
3
4     func addFactory(_ factory: Factory, for service: String) -> Void {
5         factories[service] = factory
6     }
7
8     func resolve(service: String) -> Component? {
9         guard let factory = factories[service] else {
10             return nil
11         }
12
13         let component = factory.create()
14         component.resolveDependencies(using: self)
15         return component
16     }
17 }
```

Demo



Demo



Vor-/Nachteile

👍 Leicht verständlich

👎 String-Konstanten

👎 Unflexibel

Dependency Injection in Swift

Lass das Typensystem die Arbeit machen!

Abhängigkeiten

Wie modellieren wir eine Abhängigkeit ins Typensystem?

➡ *Mit leeren Protokollen!*



Lebenszyklus der Komponenten

Wie bilden wir den Lebenszyklus einer Komponente im Typensystem ab?

➡ Mit leeren Protokollen!

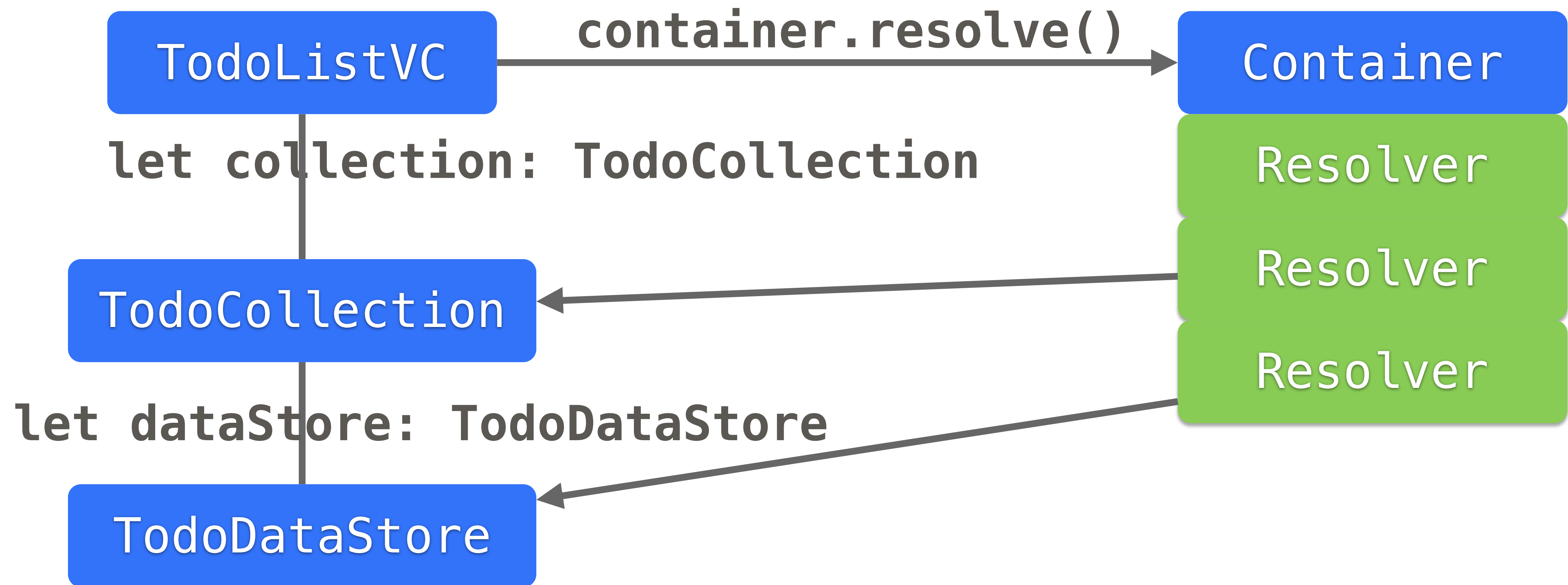


Abhängigkeiten auflösen

Wie lösen wir im Typensystem Abhängigkeiten auf?

- Resolver: Factories aus naivem Ansatz "on steroids"
- Typsicher dank **typaliases** und **extensions**
- Es gibt drei Varianten je nach Anwendungsfall*

Use case: Komponente instanzieren



Where it all starts



```
public protocol Component: class {  
    func fill(dependency: Any.Type, with object: Component) → Void  
    func dependenciesFullfilled() → Void  
}
```



```
protocol DependsOnDataStorage { }

class TodoStorage: Singleton, IndirectResolver {
    typealias DependencyToken = DependsOnDataStorage
    typealias PublicInterface = DataStorage
    var dependencies: [String : [Component]] = [:]
}

class TodoCollection: DependsOnDataStorage { ... }
```

Mehr Komfort



```
public protocol AutomaticDependencyHandling: Component {  
    var dependencies: [String: [Component]] { get set }  
}  
  
public extension AutomaticDependencyHandling {  
    func fill(dependency: Any.Type, with: Component) → Void { ... }  
  
    func component<T>() → T! { ... }  
  
    func components<T>() → [T] { ... }  
}
```


Demo

Vor-/Nachteile

- 👍 Keine String Constants
- 👍 Kein Zugriff auf Container nötig
- 👍 Wenig Boilerplate
- 👎 Lernkurve

Dependency Injection in Apple Frameworks

DI in UIKit und Co.

- Keine Kontrolle über den Lebenszyklus
- Ansatz: Protokolle
 - **ForeignInstantiable**
 - **WeakDependencyAware**
- Innerhalb der Komponente: Manuelle Auflösung

UIKit



```
if let container = UIApplication.shared.container {  
    self.resolveMe(in: container)  
}
```

UIKit

- Neue Property **container** an UIStoryboard und UIApplication
- Für UIViewController: zusätzliche Klasse **StoryboardResolver**

Demo

NSManagedObject

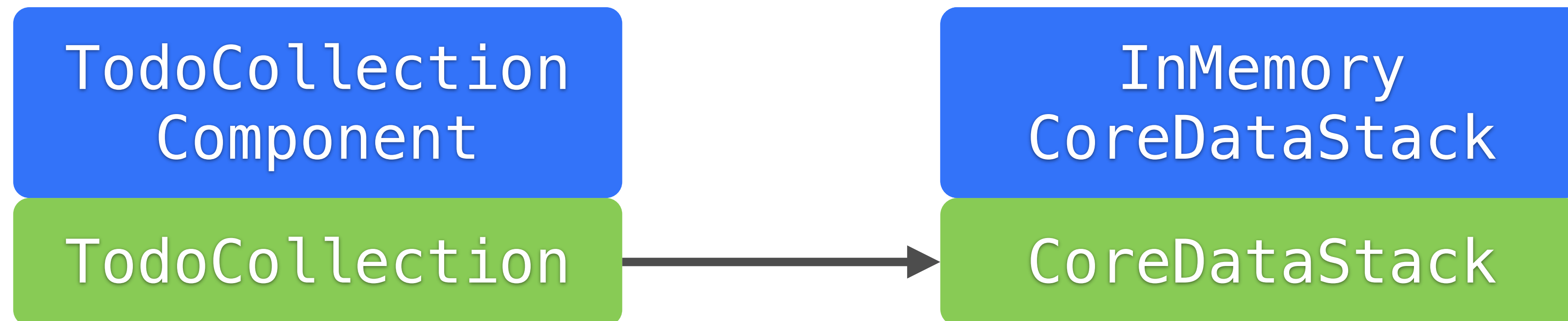
- Oberklasse **WindManagedObject** statt NSManagedObject
- Zusätzliche Property **container** an NSManagedObjectContext
- Bei Auflösung eines Faults werden Abhängigkeiten aufgelöst.

AppKit

- Storyboard-Support analog zu UIKit
 - ➡ Property **container** an `NSApplication` und `NSStoryboard`
- Zusätzliches Objekt für `NSViewController`
- in ~~`viewDidLoad`~~ **`dependenciesFulfilled()`**
alles vorhanden

XCTest

Dank DI nun einfaches Ersetzen
beliebiger Komponenten möglich



Demo

Dependency Injection in Swift

Wrapping things up

Vorzüge von DI

- 👍 Ebnet den Weg für Unit Tests
- 👍 Bildet Infrastruktur für Plugins
- 👍 Komponenten sind einfach zu schreiben
- 👍 Komponenten sind leichter wieder zu verwenden

Nachteile von DI

- 👎 Komplexer Code für den Container
- 👎 Nachvollziehbarkeit der Abhängigkeiten
- 👎 *Dependency Shopping*

Probiert's aus!

- DI Container + Protokolle

github.com/palasthotel/wind

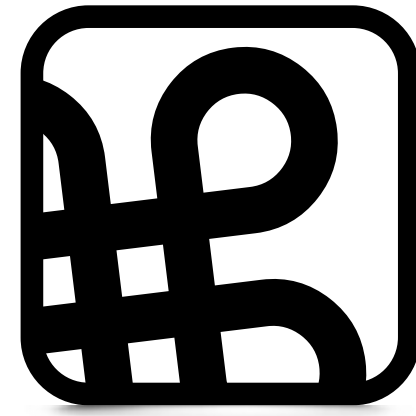
- Samples des Vortrags

github.com/palasthotel/talk-windy-dependencies

- Feedback ausdrücklich erwünscht!

Fragen?

Vielen Dank



Macoun

