// I've used protocol because they have common property
// removed optional type from isbn and title

```
protocol Book: Decodable {
    var isbn: String { get }
    var title: String { get }
    var author: String? { get }
}

struct Item: Book {
    var isbn: String // https://en.wikipedia.org/wiki/ISBN
    var title: String
    var author: String?
}

struct ComicBook: Book {
    var isbn: String
    var title: String
    var author: String?
    var marvelUniverse: Bool?
}
```

// For network layer I've removed the singleton class because with future implementations it could
// be fall in big network manager and also it violate a single responsability principle.

// Created a protocol to manage requests
// At first i've created a Protocol request to manage all HTTP requests
// We could create un Enum conform to Request and add every case we needed.
//
// In this case Item and ComicBook could have same host and different path
// we can use one enum implementation for all request or crete each implementation for different
// host

```
protocol Request {
    var host: string
    var path: string
    var headers: [String: String]
    var parameters: [String: Any]
    var type: String

    func getUrlRequest() -> URLRequest
}
```

// To made code reusable I've created generic networking protocol which take in input a request
and return raw data

```
protocol NetworkingProtocol {
    func fetch(request: Request) -> Data
}
```

// A parser that take in input raw data from network e return decodable type
// I used generics in order to manage all decodable type with one func

```
protocol Parser {
    func parse<T: decodable>(data: Data) throws -> T
}
```

```
// Generic database protocol in order to manage different types
// Added missing crud operations
protocol DatabaseProtocol {

    func fetchItems<T: DBProtocol>() -> [T]
    func fetchItem<T: DBProtocol>(query: String) -> T
    func save<T: DBProtocol>(items: [T])
    func save<T: DBProtocol>(item: T)
    func delete<T: DBProtocol>(item: T)
    func deleteAll<T: DBProtocol>()
    func update<T: DBProtocol>(item: T)
}


protocol FooViewModelProtocol {
    var networking: NetworkingProtocol
    var parser: Parser
    var database: DatabaseProtocol
    func fetchItem(completion: @escaping ()->())
    func save()
    var items: [Item]

}

//I've decoupled all business logic from VC to VM
class FooViewModel: FooViewModelProtocol {
   // View Model depends on abstraction in order to test
   // Also we can change networking/database implementation easily
    var networking: NetworkingProtocol
    var parser: Parser
    var database: DatabaseProtocol

    var items: [Item] = []

   // Used DI
   // We can easily change dependency to test
    init(networking: NetworkingProtocol, parser: Parser, database:
DatabaseProtocol) {
        self.networking = networking
        self.database = database
    }

    func fetchItem(completion: @escaping ()->()) {
      // check data in DB calling fetch<Item>...
      // if data is expired or empty perform network request
      // parsing data
      // Saving new data in DB
      // update items
      // completion
    }

    func save() {
        // saving obj to db
```

```
        database.save(items: )
    }
}
```

```
class FooViewController {
    // Removed old code to avoid massive view controller
    // All business logic is moved to vm
    // Removed instantiation inside class and used Dependency injection
    // With DI we can test several vm implementation

    var vm: FooViewModelProtocol

    init(vm: FooViewModelProtocol) {
        self.vm = vm
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        //
        if vm.items.isEmpty {
            vm.fetch(completion: {
                // reload or show error
            })
        }

    // FooViewController Declaration
    }

    func reloadTableView() {
        // reload data
    }
}
```