TodoList Tutorial

Creating a TodoList

- → Creating a project
- → Add dependencies
- → Simple Routing
- → Error Handling
- → Promises
- → Testing

Create a Project

```
mkdir TodoList
cd TodoList
swift package init
swift package generate-xcodeproj
```

Add Dependencies

```swift
import PackageDescription

let package = Package(
    name: "MyTodoList",
 dependencies: [
        .Package(url: "https://github.com/IBM-Swift/Kitura",          majorVersion: 0, minor: 32),
        .Package(url: "https://github.com/IBM-Swift/HeliumLogger",    majorVersion: 0, minor: 17)
        ]
)
```

Simple Route

```
let router = Router()

router.get("/") { request, response, next in
    response.status(.OK).send("Hello World!")
}
```

Simple Server

```
Kitura.addHTTPServer(onPort: 8090, with: todoListController.router)

Kitura.run()
```

Simple Logger

```swift
import HeliumLogger
HeliumLogger.use()

Log.info("Hello world!")
```

Using multiple targets

```
targets: [
        Target(name: "Server", dependencies: [.Target(name: "TodoList")]),
        Target(name: "TodoList")
    ],
```

Separation project to:

- `Sources/TodoList/TodoList.swift`
- `Sources/Server/main.swift`

Create a Controller

```
public final class TodoListController {

    public let router = Router()
    public init() {
        router.get("/v1/tasks", handler: handleGetTasks)
        router.post("/v1/tasks", handler: handlerAddTask)
    }
}
```

Add routes

```swift
func handleGetTasks(request: RouterRequest,
                    response: RouterResponse,
                    next: @escaping () -> Void) throws {


}

func handleAddTask(request: RouterRequest,
                    response: RouterResponse,
                    next: @escaping () -> Void) throws {

}
```

Add basic collection to Controller

```
let tasks: [String] = []
```

Get tasks

```swift
func handleGetTasks(request: RouterRequest,
                    response: RouterResponse,
                    next: @escaping () -> Void) throws {

      response.status(.OK).send(json: JSON( tasks ))

}
```

Add ability to add tasks

Add a Body Parser

```
router.all("*", middleware: BodyParser())
```

Simplify getting the JSON back

```swift
extension RouterRequest {

    var json: JSON? {
        guard let body = self.body else {
            return nil
        }

        guard case let .json(json) = body else {
            return nil
        }

        return json
    }

}
```

Get the description back

```swift
func handleAddTask(request: RouterRequest,
                   response: RouterResponse,
                   next: @escaping () -> Void) throws {

    if let json = request.json else {
        response.status(.badRequest)
        next()
        return
    }

    let description = json["description"].stringValue

    tasks.append(description)

}
```

Protect your array

```
let queue = DispatchQueue(label: "com.example.tasklist")

queue.sync {

}
```

Create a more rich Task

```swift
struct Task {

    let id:          UUID
    let description: String
    let createdAt:   Date
    let isCompleted: Bool

}
```

Make it Equatible

```
extension Task: Equatable { }

func == (lhs: Task, rhs: Task) -> Bool {
    if  lhs.id          == rhs.id,
        lhs.description == rhs.description,
        lhs.createdAt   == rhs.createdAt,
        lhs.isCompleted == rhs.isCompleted
    {
        return true
    }
    return false

}
```

Make things tranformable to Dictionary

```
typealias StringValuePair = [String: Any]

protocol StringValuePairConvertible {
    var stringValuePairs: StringValuePair {get}
}
```

Make collections also tranformable to Dictionary

```
extension Array where Element : StringValuePairConvertible {

    var stringValuePairs: [StringValuePair] {
        return self.map { $0.stringValuePairs }
    }
}
```

Make Task a StringValuePairConvertible

```
extension Task: StringValuePairConvertible {

    var stringValuePairs: StringValuePair {
        return [
            "id":            "\(self.id)",
            "description":   self.description,
            "createdAt":     self.createdAt.timeIntervalSinceReferenceDate,
            "isCompleted":   self.isCompleted
        ]
    }

}
```

Change [String] to [Task]

```
private var tasks: [Task] = []

response.status(.OK).send(json: JSON(task.stringValuePairs))
```

Add task with Tasks

```
task.append(Task(id: UUID(),
                 description: "Do the dishes",
                 createdAt: Date(),
                 isCompleted: false))
```

Factor out the Database

```
final class TaskDatabase {

    private var storage: [Task] = []

    let queue = DispatchQueue(label: "com.example.tasklist")

    func addTask(oncompletion: (Task) -> Void) {

            queue.sync {
                self.storage.append(task)
                oncompletion(task)
            }
    }

    func getTasks(oncompletion: ([Task]) -> Void) {

            queue.sync {
                oncompletion(self.storage)
            }
    }
}
```

Use asynchronous callbacks

Error Handling

TaskListError

```
enum TaskListError : LocalizedError {

    case descriptionTooShort(String)
    case descriptionTooLong(String)
    case noJSON
```

Error Description

```
var errorDescription: String? {
        switch self {
        case .descriptionTooShort(let string): return "\(string) is too short"
        case .descriptionTooLong(let string): return "\(string) is too long"
        case .noJSON: return "No JSON in payload"
        }
    }
```

Make it convertible to JSON

```
extension TaskListError: StringValuePairConvertible {

    var stringValuePairs: StringValuePair {
        return ["error": self.errorDescription ?? ""]
    }
}
```

Validating the Request

```swift
let maximumLength = 40
let minimumLength = 3

struct AddTaskRequest {
    let description: String
}
```

Validate the request

```
func validateRequest(request: RouterRequest) throws -> AddTaskRequest {

    guard let json = request.json else {
        throw TaskListError.noJSON
    }

    let description = json["description"].stringValue

    if description.characters.count > maximumLength {
        throw TaskListError.descriptionTooLong(description)
    }

    if description.characters.count < minimumLength {
        throw TaskListError.descriptionTooShort(description)
    }

    return AddTaskRequest(description: description)

}
```

Use Promises

Add MiniPromiseKit

```
.Package(url: "https://github.com/davidungar/miniPromiseKit", majorVersion: 4, minor:  1),
```

Create Promises

```
final class TaskDatabase {

    private var storage: [Task] = []

    let queue = DispatchQueue(label: "com.example.tasklist")

    func addTask(task: Task) -> Promise<Task> {

        return Promise{ fulfill, reject in
            queue.sync {
                self.storage.append(task)
                fulfill(task)
            }
        }
    }

    func getTasks() -> Promise<[Task]> {

        return Promise{ fulfill, reject in
            queue.sync {
                fulfill(self.storage)
            }
        }
    }
}
```

Use the Promises

```
_ = firstly {
            taskDatabase.getTasks()
            }.then (on: self.queue) { tasks in
                response.status(.OK).send(json: JSON(tasks.stringValuePairs))
            }
            .catch (on: self.queue) { error in
                if let err = error as? TaskListError {
                    response.status(.badRequest).send(json: JSON(err.stringValuePairs))
                }
            }
            .always(on: self.queue) {
                next()
        }
```

Use the Promises

```
_ = firstly { () throws -> Promise<Task> in

            let addRequest = try validateRequest(request: request)
            let task = Task(with: addRequest)

            return taskDatabase.addTask(task: task)
            }
            .then (on: self.queue) { task -> Void in
                response.status(.OK).send(json: JSON(task.stringValuePairs))

            }
            .catch (on: self.queue) { error in
                if let err = error as? TaskListError {
                    response.status(.badRequest).send(json: JSON(err.stringValuePairs))
                }

            }
            .always(on: self.queue) {
                next()
        }
```

Testing

Set up Kitura framework

```
private let queue = DispatchQueue(label: "Kitura runloop", qos: .userInitiated, attributes: .concurrent)

    public let defaultSession = URLSession(configuration: .default)

    private let todoListController = TodoListController()

    override func setUp() {
        super.setUp()

        Kitura.addHTTPServer(onPort: 8090, with: todoListController.router)

        queue.async {
            Kitura.run()
        }
    }
```

Add a test

```swift
func testGetTodos() {
        let expectation1 = expectation(description: "Get Todos")

        var url: URLRequest = URLRequest(url: URL(string: "http://localhost:8090/v1/tasks")!)
        url.addValue("application/json", forHTTPHeaderField: "Content-Type")
        url.httpMethod = "GET"
        url.cachePolicy = URLRequest.CachePolicy.reloadIgnoringCacheData

        let dataTask = defaultSession.dataTask(with: url) {
            data, response, error in
            XCTAssertNil(error)

            switch (response as? HTTPURLResponse)?.statusCode {
            case 200?:
                guard let data = data else {
                    XCTAssert(false)
                    return
                }

                let json = JSON(data: data)

                print(json)
                expectation1.fulfill()

            case nil:       XCTFail("response not HTTPURLResponse")
            case let code?: XCTFail("bad status: \(code)")
            }
        }

        dataTask.resume()
        waitForExpectations(timeout: 10, handler: { _ in  })
    }
```

Enable Code coverage

```
swift package generate-xcodeproj --enable-code-coverage
```