Writing an API with Swift 3 on Linux

Rob Allen

akrabat.com ~ @akrabat ~ October 2016

Swift?

Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns.

swift.org

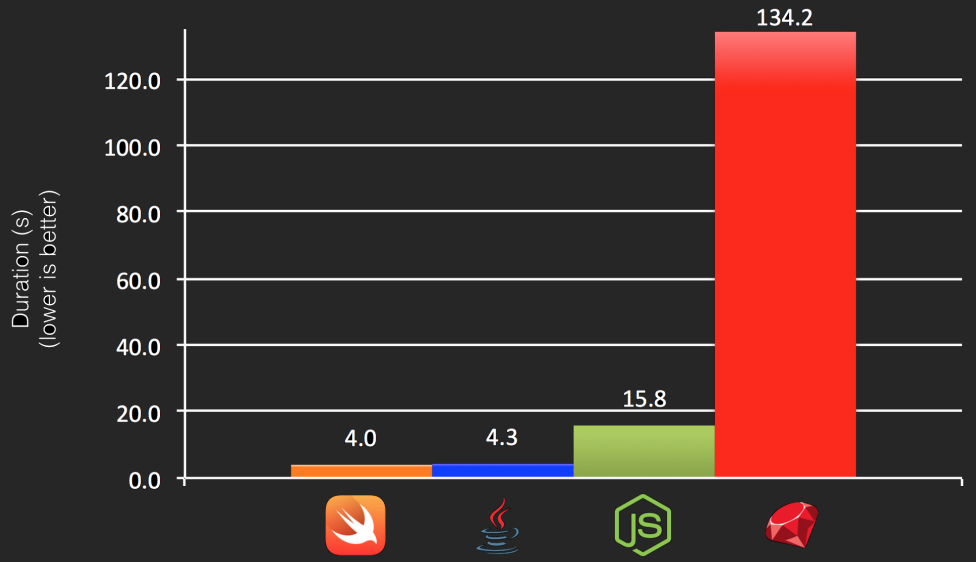
Open Source

- Created by Apple
- Apache 2 license
- Source code on GitHub
- Swift-evolution: open design of new features

Cross Platform

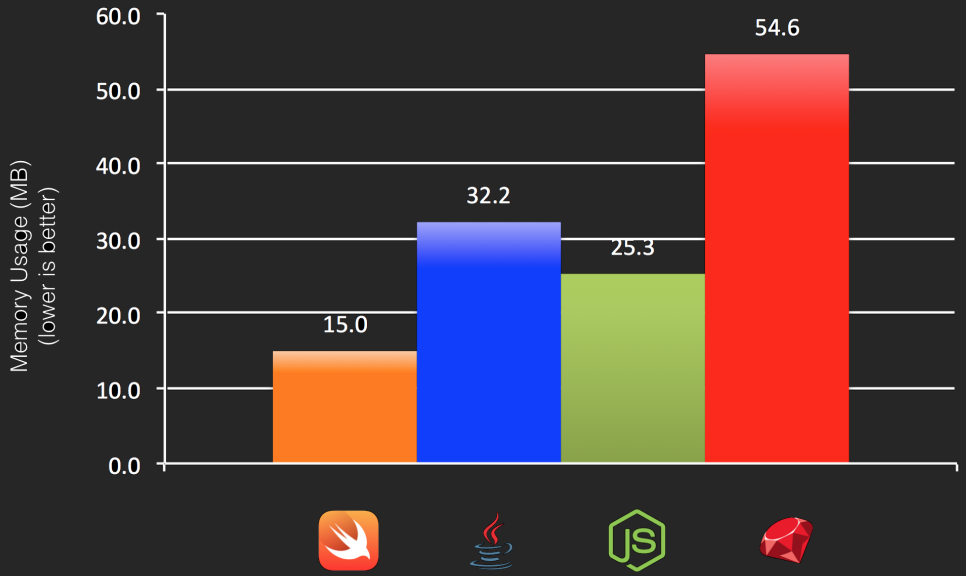
- Runs on Linux (x86) and all Apple OSs
- Ports in progress: Android, Linux(ARM), FreeBSD, Windows
- Cross-platform: Standard library, Foundation, Dispatch & XCTest

Performance



http://benchmarksgame.alioth.debian.org/u64q/performance.php?test=spectralnorm

Memory



http://benchmarksgame.alioth.debian.org/u64q/performance.php?test=spectralnorm

Major features

Strong typing
Type inference
Closures
Optionals

Custom operators
Tuples
Generics
Interoperable

Safety

- Type safety
- Prefer constants over variables
- Variables are always initialized before use
- Optionals: variables can never be `nil`

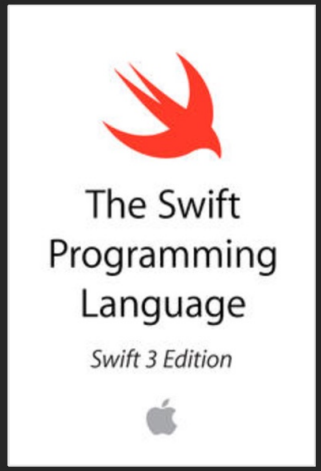
Rock-Paper-Scissors

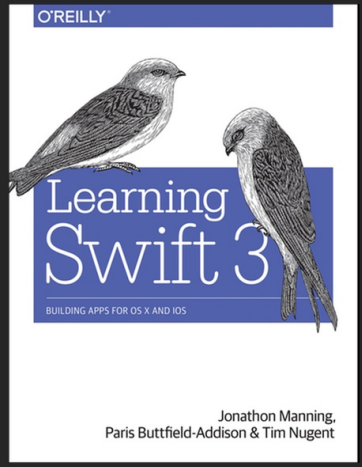
```
1  import Foundation
2
3  let shapes = ["rock", "paper", "scissors"]
4
5  for count in 1...3 {
6      print(count)
7      sleep(1)
8  }
9
10 srandom(UInt32(NSDate().timeIntervalSince1970))
11 let chosenShape = random() % shapes.count
12 print(player[chosenShape]);
```

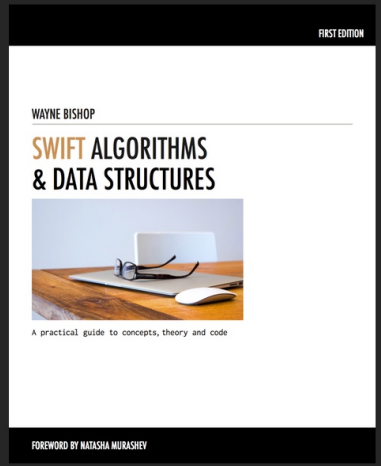
Result

```
$ swift rock-paper-scissors.swift
1
2
3
scissors
```

Learn the language







So what does it take to write Swift APIs on Linux?

Steps to Swift APIs

1. Environment
2. Framework
3. Write an API!
4. Deploy

Environment

Environment

- Ubuntu Linux 14.04, 15:10 (soon 16.04)
- Local development
    - Docker
    - Vagrant
    - Mac

Swiftenv version manager

- Simple install of releases and snapshots

```
$ swiftenv install 3.0
```

- Scoped to project

```
$ cd my-project
$ swiftenv local 3.0

# creates: .swift-version
```

Swift Package Manager

The Swift Package Manager is a tool for managing the distribution of Swift code. It's integrated with the Swift build system to automate the process of downloading, compiling, and linking dependencies.

swift.org

Swift Package Manager

- Dependency manager
    - Fetches & builds dependencies from Git
    - Controlled via `Package.swift`

Swift Package Manager

- Dependency manager
  - Fetches & builds dependencies from Git
  - Controlled via `Package.swift`
- Part of Swift cli:
  - `swift package`
  - `swift build`
  - `swift test`

Create a new project

```
$ cd my-project
$ swift package init --type executable
Creating executable package: swiftpm
Creating Package.swift
Creating .gitignore
Creating Sources/
Creating Sources/main.swift
Creating Tests/
```

Package.swift

```
1 import PackageDescription
2
3 let package = Package(
4     name: "my-project"
5 )
```

Add dependencies

```
1  import PackageDescription
2
3  let package = Package(
4      name: "my-project",
5      dependencies: [
6          .Package(url: "https://github.com/jatoben/CommandLine",
7              majorVersion: 2, minor: 0),
8          .Package(url: "https://github.com/stormpath/Turnstile",
9              majorVersion: 1),
10     ]
11 )
```

Choose a web framework

Web frameworks

Pick the one that fits you best

(We'll use Kitura)

Kitura

```
1 import PackageDescription
2
3 let package = Package(
4     name: "HelloWorld",
5     dependencies: [
6         .Package(url: "https://github.com/IBM-Swift/Kitura.git",
7             majorVersion: 1, minor: 0),
8     ]
9 )
```

Hello world

```
1  import Kitura
2  import SwiftyJSON
3
4  let router = Router()
5  router.get("/hello") { request, response, next in
6      response.status(.OK).send(json: JSON(["hello" : "world"]))
7      next()
8  }
9
10 Kitura.addHTTPServer(onPort: 8090, with: router)
11 Kitura.run()
```

Hello world

```
$ curl -i -X GET  http://localhost:8090/hello
HTTP/1.1 200 OK
Date: Sun, 09 Oct 2016 08:26:34 GMT
Content-Length: 22
Content-Type: application/json
Connection: Keep-Alive
Keep-Alive: timeout=60, max=99

{
  "hello": "world"
}
```

Writing an API

Primary aspects of a RESTful API

- URI for *each* resource: `https://localhost:8090/books/14`
- HTTP methods are the set of operations allowed for the resource
- Correct use of status codes
- Media type used for representations of the resource
- The API must be hypertext driven

HTTP methods & URIs

HTTP methods

Method	Used for	Idempotent?
GET	Retrieve data	Yes
PUT	Change data	Yes
DELETE	Delete data	Yes
POST	Change data	No
PATCH	Update data	No

Send `405 Method Not Allowed` or `404 Not Found` if cannot honour

A URI for each resource

```
1 router.get("/books", handler: listBooksHandler)
2 router.post("/books", handler: createBookHandler)
3
4 router.get("/books/:id", handler: getBookHandler)
5 router.put("/books/:id", handler: replaceBookHandler)
6 router.patch("/books/:id", handler: updateBookHandler)
7 router.delete("/books/:id", handler: deleteBookHandler)
```

Dynamic routes

```
1  router.get("/books/:id") { request, response, next in
2
3      guard let id: String = request.parameters["id"] else {
4          response.status(.notFound)
5              .send(json: JSON(["error": "Not Found"]))
6          next()
7      }
8
9      // id is now valid
10     // do things and set up response
11     next()
12 }
```

Reading data

Query parameters:

```
1 router.get("/books") { request, response, next in
2
3   // e.g. http://localhost/books?filter=new
4   let filter = request.quaryParameters["filter"] ?? ""
```

Body data:

```
1 router.post("/books") { request, response, next in
2
3   let data = try request.readString() ?? ""
4   // decode data appropriately
```

JSON & form body data

Add BodyParser middleware:

```
1 router.all("/name", middleware: BodyParser())
```

Retrieve parsed body in callback:

```
1 guard let parsedBody = request.body else { next(); return }
2
3 switch(parsedBody) {
4     case .json(let jsonBody):
5         // do something with jsonBody
6     case .multipart(let parts):
7         // do something with form data
8     default: break
9 }
```

Content-type handling

Content negotiation

Correctly parse the request

- Read the `Content-Type` header
- Decode body appropriately

Correctly create the response

- Read the `Accept` header
- Set the `Content-Type` header

Raise `415 Unsupported media type` status if unsupported

Reading headers in Kitura

```
1  router.get("/books/:id") { request, response, next in
2
3      let contentType = request.headers["Accept"] ?? ""
4
5      if contentType.range(of: "json") == nil {
6          // send 415
7          Log.info("Invalid Media Type: \(contentType)")
8          response.status(.unsupportedMediaType)
9          response.send(json: JSON(["error": "Unsupported Media Type"]))
10         next()
11     }
12
13     // Okay to continue
```

Hypermedia

- Media type used for a representation
- The link relations between representations and/or states
- Important for discoverability

Paginate collections

Mobile devices don't *that* much memory!

- Hypermedia link relations: `first`, `last`, `next` & `prev` relations
- Include count of items sent
- Include total count of items too

Media types matter

With `application/json` you abdicate responsibility.

A more structured media type:

- Tells the client how to interpret the data
- Enforces structure of the payload
- Informs on what the payload data means

Hal+Json

- Resources
  - State (standard data)
  - Links (to URIs)
  - Embedded resources (within this resource)
- Links
  - Target (URI)
  - Relation (i.e. the name of the link)

Hal+Json

```
Content-Type: application/hal+json

{
  "title": "The Hunger Games",
  "isbn": "9780439023528"
}
```

Hal+Json

```
Content-Type: application/hal+json

{
  "title": "The Hunger Games",
  "isbn": "9780439023528",
  "_links": {
    "self": { "href": "http://localhost:8090/books/1" }
  }
}
```

Hal+Json

```
Content-Type: application/hal+json

{
  "title": "The Hunger Games",
  "isbn": "9780439023528",
  "_links": {
    "self": { "href": "http://localhost:8090/books/1" }
  },
  "_embedded": {
    "author": {
      "name": "Suzanne Collins",
      "_links": {
        "self": { "href": "http://localhost:8090/authors/1" }
      }
    }
```

In Kitura

```
1  var json = book.toJSON()
2
3  json["_links"] = JSON(["self": ["href": baseURL + "/books/" + book.id]])
4
5  json["_embedded"]["author"] = book.getAuthor().toJSON()
6  json["_embedded"]["author"]["_links"] = JSON(["self":
7      ["href": baseURL + "/authors/" + book.getAuthor().id]])
8
9  response.status(.OK).send(json: json)
10 response.headers["Content-Type"] = "application/hal+json"
11 next()
```

(or use a library!)

Error handling

Error handling

- Error representations are first class citizens
  - Correct content-type
  - Uses correct HTTP status code

Error handling

- Error representations are first class citizens
  - Correct content-type
  - Uses correct HTTP status code
- Provides application error code & human readable message
  - End user needs a short, descriptive message
  - Client developer needs detailed information
  - Client application needs an error code

HTTP Problem (RFC 7807)

```
HTTP/1.1 503 Service Unavailable
Content-Type: application/problem+json
Content-Language: en
```

```
{
   "status": 503,
   "type": "https://example.com/service-unavailable",
   "title": "Could not authorise user due to an internal problem - try late
   "detail": "The authentication service is down for maintenance.",
   "instance": "https://example.com/maintenance-schedule/2016-08-30",
   "error_code": "AUTHSERVICE_UNAVAILABLE"
}
```

Testing

XCTest

- Supplied as part of the Swift toolchain
- Create module subdirectory in `Tests` directory
- Create Test files
- Create `Tests/LinuxMain.swift`

Testcase

```
1  import XCTest
2  @testable import BookshelfAPI
3
4  class BookTests: XCTestCase {
5    static var allTests : [(String, (BookTests) -> () throws -> Void)] {
6        return [ ("testBookToJson", testBookToJson) ]
7    }
8    func testBookToJson() {
9        let book = Book(id: "1", title: "2", author: "3", isbn: "4")
10
11       XCTAssertEqual(book.id, "1", "Book id incorrect")
12       XCTAssertEqual(book.title, "1", "Book title is incorrect")
13   }
14 }
```

LinuxMain.swift

```
1 import XCTest
2 @testable import BookshelfTests
3
4 XCTMain([
5     testCase(BookTests.allTests),
6 ])
```

Run tests

```
$ swift test
[compiles and links]

Test Suite 'EntityTests' started at 2016-10-09 14:44:49.510
Test Case '-[BookshelfTests.EntityTests testAuthorToJson]' started.
Test Case '-[BookshelfTests.EntityTests testAuthorToJson]' passed (0.002 seconds)
Test Case '-[BookshelfTests.EntityTests testBookToJson]' started.
./Tests/BookshelfTests/EntityTests.swift:14: error:
  -[BookshelfTests.EntityTests testBookToJson] :
  XCTAssertEqual failed: ("2") is not equal to ("1") - Book title is incorrect
Test Case '-[BookshelfTests.EntityTests testBookToJson]' failed (0.003 seconds).
Test Suite 'EntityTests' failed at 2016-10-09 14:44:49.515.
   Executed 2 tests, with 1 failure (0 unexpected) in 0.005 (0.005) seconds
```

Deployment

Deployment

- Any Ubuntu Linux server
- Heroku / Bluemix / AWS
- Openwhisk (serverless)

Heroku

Use Kyle Fuller's buildpack:

```
$ heroku create --buildpack https://github.com/kylef/heroku-buildpack-swift
Creating app... done, ▫ intense-scrubland-10670
Setting buildpack to https://github.com/kylef/heroku-buildpack-swift.git...
https://intense-scrubland-10670.herokuapp.com/ |
  https://git.heroku.com/intense-scrubland-10670.git
```

Create `.swift-version`

```
3.0
```

Create `Procfile`:

```
web: HelloWorld --workers 1 --bind 0.0.0.0:$PORT
```

Environment variables

You will be running on `$PORT`, not `8090`:

Add `DotEnv` dependency:

```
1 .Package(url: "https://github.com/SwiftOnTheServer/SwiftDotEnv.git",
2   majorVersion: 1),
```

Update to use `$PORT`

```
1 let env = DotEnv()
2 let port = env.getAsInt("PORT") ?? 8090
3 Kitura.addHTTPServer(onPort: port, with: router)
```

Deploy

```
$ git push heroku master
Counting objects: 19, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (19/19), 3.12 KiB | 0 bytes/s, done.
Total 19 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:

[... install Swift & run swift build ...]

remote: ---> Launching...
remote:      Released v6
remote:      https://intense-scrubland-10670.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
```

Run

```
$ curl -i https://intense-scrubland-10670.herokuapp.com/hello
HTTP/1.1 200 OK
Server: Cowboy
Connection: keep-alive
Content-Type: application/json
Date: Sun, 09 Oct 2016 15:04:34 GMT
Content-Length: 22
Via: 1.1 vegur

{
  "hello": "world"
}
```

OpenWhisk

- Serverless
- Uses `wsk` command line tool
- Each action is a simple Swift application with a `main` function
- Event driven: send a POST to trigger

An OpenWhisk action

```swift
// HelloWorld.swift
func main(args: [String:Any]) -> [String:Any] {
    return [ "hello" : "world" ]
}
```

Add to OpenWhisk:

```
$ wsk action create hello_world HelloWorld.swift
```

Run:

```
$ wsk action invoke --blocking --result hello_world
{
    "hello": "world"
}
```

To sum up

Resources

Websites:

- https://swift.org
- http://www.kitura.io
- https://akrabat.com/kitura-tutorial
- https://swift.org/community/#mailing-lists

Newsletters:

- https://www.serverswift.tech
- https://swiftnews.curated.co

Thank you!

Rob Allen ~ akrabat.com ~ @akrabat