Networking

REST, WebSocket, Concurrency

REST

Representational State Transfer (**REST**) is a software architectural style that defines a set of constraints to be used for creating Web services.

REST works over the Hypertext Transfer Protocol (HTTP), which is a protocol originally created to allow the transmission of web pages over the internet.

In HTTP a *session* is usually a sequence of network requests and responses, although when making calls from iOS apps, we usually use a single session for every call we make.

An HTTP request usually contains:

- **a URL** for the resource we want
- **an HTTP method** to state the action we want to perform
- **optional parameters** in the form of HTTP headers
- **some optional data** we want to send to the server

REST Client

The core of our REST client will be built on these following components:

Models: Classes that describe the data models of our application, reflecting the structure of data received from, or sent to, the backend servers.

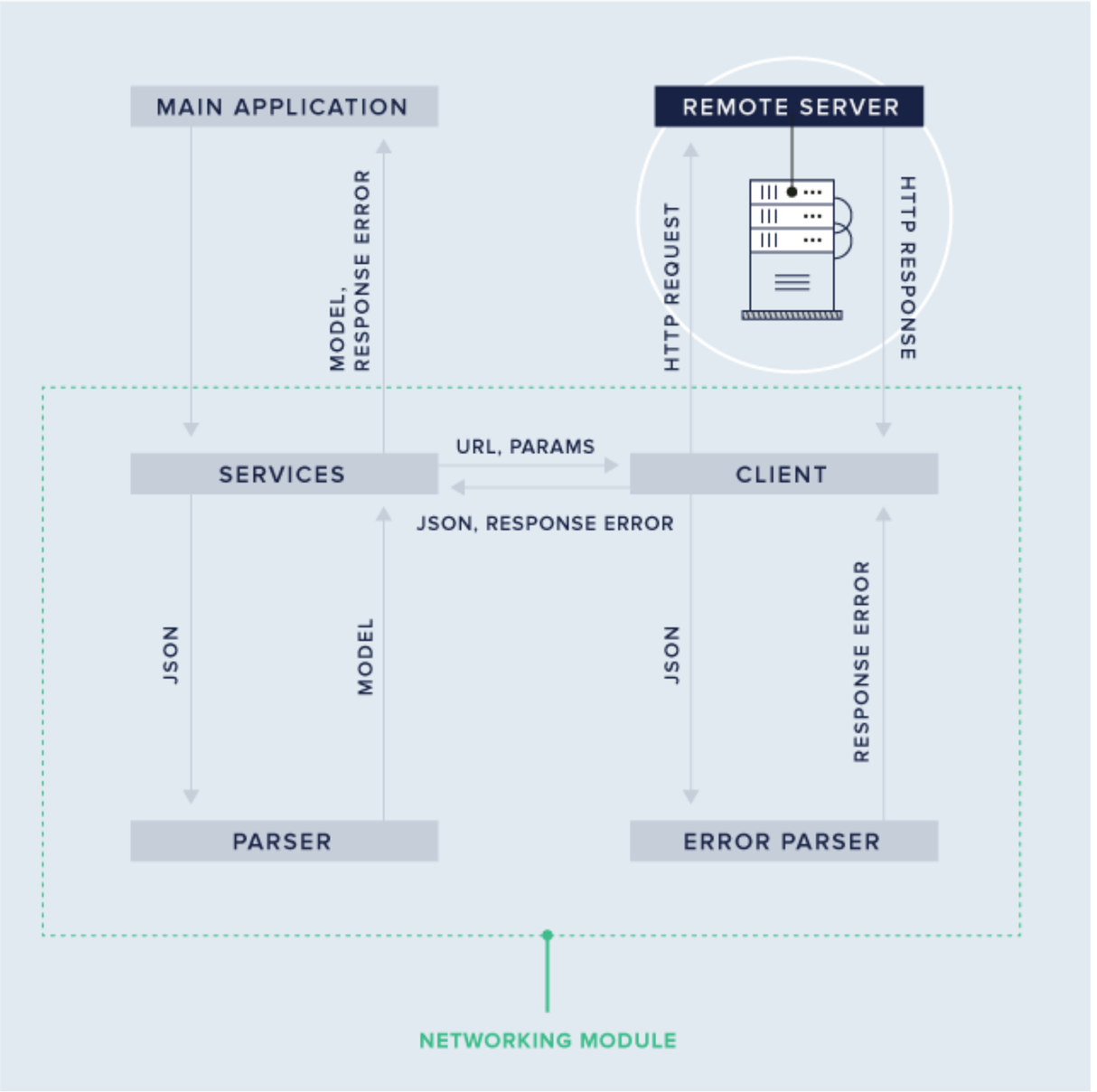
Parsers: Responsible for decoding server responses and producing model objects.

Errors: Objects to represent erroneous server responses.

Client: Sends requests to backend servers and receives responses.

Services: Manage logically linked operations (e.g. authentication, managing user related data, analytics, etc).

To make network requests in Swift we can use **Alamofire** or **URLSession**.


MAIN APPLICATION
REMOTE SERVER
MODEL, RESPONSE ERROR
HTTP REQUEST
HTTP RESPONSE
URL, PARAMS
SERVICES
CLIENT
JSON, RESPONSE ERROR
JSON
MODEL
JSON
RESPONSE ERROR
PARSER
ERROR PARSER
NETWORKING MODULE

URLSession

To fetch some data we need to setup the HTTP request with URLSession:

```
let urlString = "http://192.168.88.99:8180/countries"
guard let url = URL(string: urlString) else { return }
let session = URLSession.shared
```

Next step - create a data task with dataTask(with:completionHandler:) function of URLSession:

```
let task = session.dataTask(with: url) { data, response, error in
    print("Data: \(data)")
    print("Response: \(response)")
    print("Error: \(error)")
}
```

To start this request we should call next function:

```
task.resume()
```

Alamofire

To fetch some data we need to setup the Session Manager of Alamofire:

```
let sessionManager: Alamofire.SessionManager = {
    let configuration = URLSessionConfiguration.default
    configuration.timeoutIntervalForRequest = 15

    let sessionManager = Alamofire.SessionManager(configuration: configuration)
    return sessionManager
}()
```

Making request using Alamofire:

```
sessionManager.request(url).responseData { response in
    switch response.result {
    case .success:
        print("Data: \(response.data)")
    case .failure(let error):
        print("Error: \(error))")
    }
    print("Response: \(response.response))")
}
```

URLSession vs Alamofire

URLSession

Request type:

```
var request = URLRequest(url: url)
request.httpMethod = "POST" // "GET", "DELETE", "PUT"
```

Request parameters:

```
let parameterDictionary = ["username" : "Test", "password" : "123456"]
request.setValue("Application/json", forHTTPHeaderField: "Content-Type")
guard let httpBody = try?
    JSONSerialization.data(withJSONObject: parameterDictionary) else {
    return
}
request.httpBody = httpBody
```

Completion call:

```
DispatchQueue.main.async {
    competition(human)
}
```

Alamofire

```
sessionManager.request(url, method: .post) // .get, .put, .delete
```

```
let parameters: Parameters = ["username" : "Test", "password" : "123456"]
sessionManager.request(url, method: .post,
                       parameters: parameters,
                       encoding: JSONEncoding.default)
```

```
competition(human)
```

Mapping

Mapping is an operation where each element of a given set is associated with one or more elements of a second set.

We need to map our model class objects with the JSON data.

For parsing and mapping JSON in Swift we can use:
- Codable
- ObjectMapper
- Other libraries

Codable:
- More optimized and clean.
- Less code is required.
- It’s Swift native solution and removes third party dependency.

ObjectMapper:
- Faster and has better readability and support for Alamofire.
- 3-rd party framework.

ObjectMapper

ObjectMapper is a framework that can convert JSON to objects and back.

For using ObjectMapper:

- Our objects need to extend from **Mappable**
- Our objects need to implement the **mapping** function in which we will specify which properties of the JSON are assigned to which properties of the object.
- Properties must be declared as optional variables

Using ObjectMapper

```
class Human: Mappable {
    let firstName: String?
    let lastName: String?

    init?(map: Map) { }

    mutating func mapping(map: Map) {
        firstName  <- map["firstName"]
        lastName   <- map["lastName"]
    }
}
```

Convert a JSON data to a model object:

```
let jsonObject = try! JSONSerialization.jsonObject(with: jsonData)
let human = Mapper<Human>().map(JSONObject: jsonObject)
```

Convert a model object to a JSON string:

```
let jsonString = Mapper().toJSONString(human, prettyPrint: true)
```

Codable

Codable is a protocol introduced in Swift 4 Standard library.

It provides three types: **Encodable**(encoding), **Decodable**(decoding), **Codable**(both) protocols.

For using **Codable**:

- To encode and decode the custom types need to **adopt Codable** protocol.
- Custom type **must have** the **Codable type** properties.
- Codable types include data types like Int, Double, String, URL, Data.
- Other properties like array, dictionary are codable if they are comprised of codable types.

Using Codable

```
class Human: Codable {
    public let firstName: String?
    public let lastName: String?
}
```

Convert a JSON data to a model object:

```
let human = try! JSONDecoder().decode(Human.self, from: jsonData)
```

Convert a model object to a JSON data:

```
let data = try! JSONEncoder().encode(human)
let jsonData = try! JSONSerialization.jsonObject(with: data, options: [])
```

Native vs 3-rd party

URLSession + Codable

```
func getHuman(competition: @escaping (Human?, Error?) -> Void) {
    let urlString = "http://192.168.88.99:8180/human"
    guard let url = URL(string: urlString) else { return }
    let session = URLSession.shared

    let task = session.dataTask(with: url) { data, _, error in
        if let data = data {
            let human = try! JSONDecoder().decode(Human.self, from: data)
            competition(human, nil)
        }
        if let error = error {
            competition(nil, error)
        }
    }

    task.resume()
}
```

Alamofire + ObjectMapper

```
func getHuman(completionHandler: @escaping (Human?, Error?) -> Void) {
    let urlString = "http://192.168.88.99:8180/human"
    guard let url = URL(string: urlString) else { return }

    sessionManager.request(url).responseData { response in
        switch response.result {
        case .success:
            guard let jsonObject = try? JSONSerialization.jsonObject(with: value),
                let result = Mapper<T>().map(JSONObject: jsonObject) else {
                    print("Error Serialization JSON")
                    return
            }
            completionHandler(result, nil)
        case .failure(let error):
            completionHandler(nil, error)
        }
    }
}
```

SocketManager and SocketIOClient

```
let configuration = SocketIOClientConfiguration(arrayLiteral: .log(false),
                                                .forceWebsockets(true),
                                                .secure(true),
                                                .reconnectAttempts(2),
                                                .forceNew(true))

let url = "https://your.url"

let manager = SocketManager(socketURL: url, config: configuration)
```

Enable logging

Set transport only by WebSockets

Secure transports

Always create new engine by client

```
private var socket: SocketIOClient? {
    return manager?.defaultSocket
}
```

Socket methods

SocketManager.connect() - connects the underlying transport and the default namespace socket.

Most times this function is called after user login or in case of socket reconnection attempts.

SocketManager.disconnect() - disconnects the manager and all associated sockets.

Most times this function is called if app moves to background.

Socket event handlers

Use SocketIOClient.**on()** to add handler for event

```
socket?.on(clientEvent: "greeting", callback: {[weak self] (data, ack) in
    print("greeting")
})
```

Use SocketIOClient.**once()** to add a single-use handler for an event.

```
socket?.once(clientEvent: "greeting", callback: {[weak self] (data, ack) in
    print("greeting")
})
```

Use SocketIOClient.**off()** to remove handler for event. To remove handlers for all events, call function SocketIOClient.**removeAllHandlers()**

```
func removeGreetingEventHandler() {
    socket.off(clientEvent: "greeting")
}
```

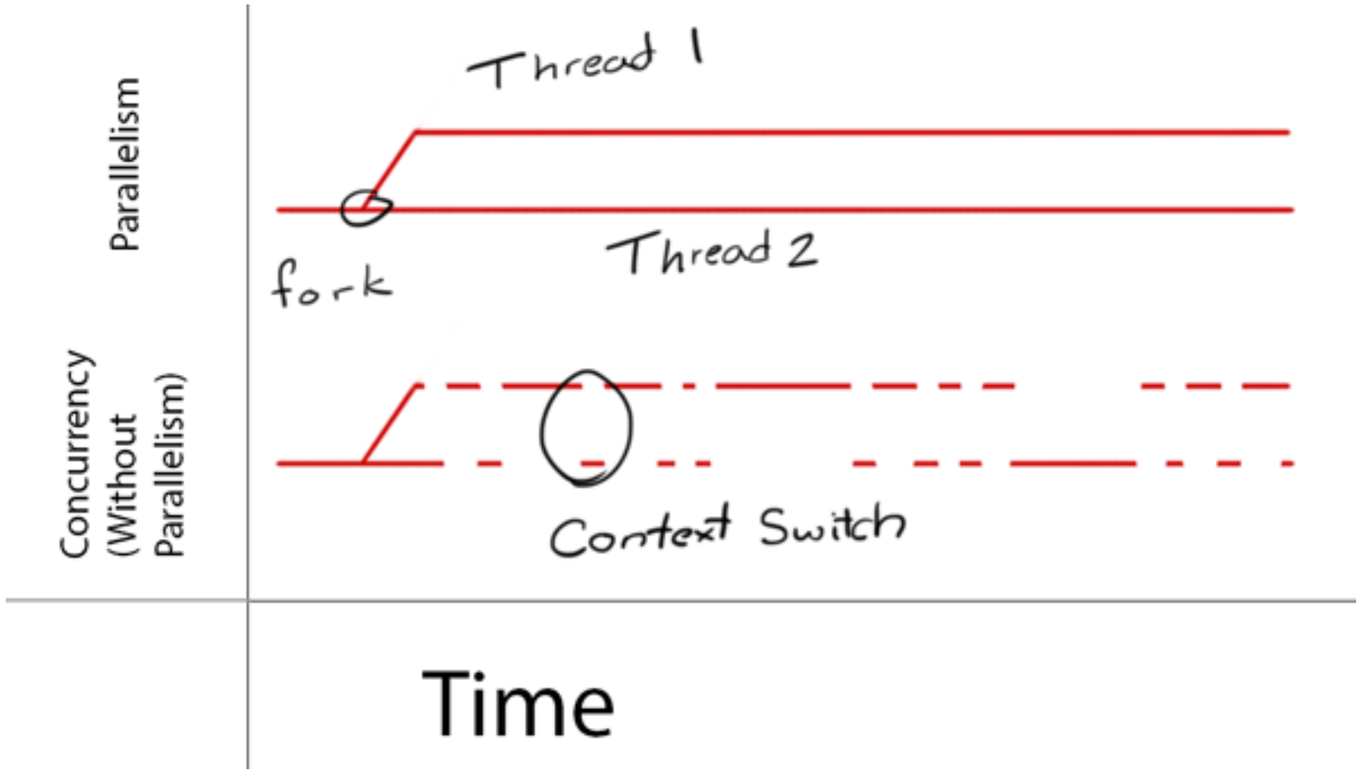
Emit

Use SocketIOClient.**emit()** to send events by socket to server.

```
func sendHello() {
    guard socket.status == .connected else { return } //check if socket is conneted

    let event = "message"                             //key for socket event
    socket.emit(event, "Hello!")
}
```

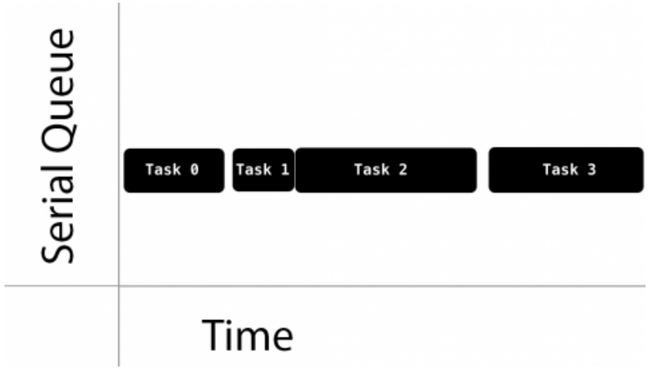
Concurrency & Threading

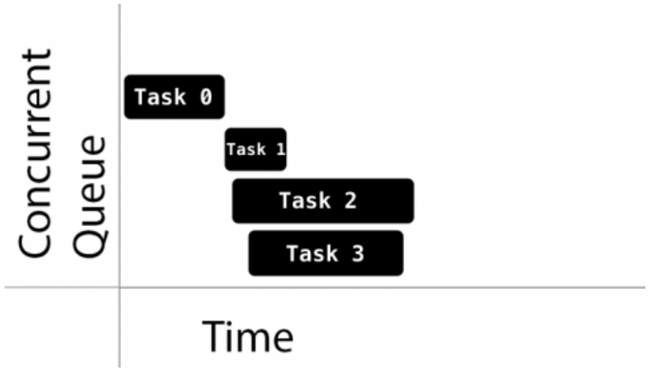


Grand Central Dispatch

Queues:

- **DispatchQueue** class
- **FIFO** order
- Thread-safe
- Can be either **serial** or **concurrent**





Types of Queues

- **Main queue**: runs on the main thread and is a serial queue.
- **Global queues**: concurrent queues that are shared by the whole system.
- **Custom queues**: queues that you create which can be serial or concurrent.

The **QoS** classes:

- **User-interactive**: Main thread.
- **User-initiated**: High priority global queue.
- **Utility**: Low priority global queue.
- **Background**: Background priority global queue.

Synchronous & Asynchronous:

- `DispatchQueue.sync(execute:)`
- `DispatchQueue.async(execute:)`

Example

```
DispatchQueue.global(qos: .userInitiated).async { [weak self] in

    let newImage = self?.getImage() // heavy image task
    let imageView = UIImageView(image: newImage)

    DispatchQueue.main.sync {
        self?.view = imageView // set on view in main thread
    }

}
```