Разработка клиент-серверных приложений

HTTP (HyperText Transfer Protocol) и HTTPS (HyperText Transfer Protocol Secure) — это два протокола, которые лежат в основе обмена данными в интернете. Эти протоколы используются для передачи данных между клиентом (например, браузером или мобильным приложением) и сервером, где хранятся ресурсы.

HTTP

HTTP — это протокол, который позволяет клиентам (например, вашему браузеру) запрашивать данные с веб-серверов. Он является текстовым и работает по принципу запроса-ответа: клиент отправляет запрос на сервер, а сервер возвращает ответ. Этот протокол не обеспечивает шифрование данных, что значит, что информация может быть перехвачена третьими лицами. Это может быть проблемой при передаче конфиденциальных данных, таких как пароли или номера кредитных карт.

HTTPS

HTTPS — это защищённая версия HTTP. Когда вы видите “https://” в адресной строке браузера, это значит, что соединение между клиентом и сервером защищено с помощью SSL/TLS. Это шифрует данные, передаваемые между клиентом и сервером, что делает их нечитаемыми для посторонних. Важно отметить, что использование HTTPS стало стандартом, особенно для сайтов, где происходит обработка личной информации или финансовых операций. Благодаря SSL/TLS-шифрованию HTTPS значительно повышает уровень безопасности.

Принцип работы HTTP/HTTPS

Процесс общения через HTTP или HTTPS обычно выглядит так:

1. **Запрос от клиента:** Клиент, например, мобильное приложение или браузер, отправляет HTTP-запрос на сервер с просьбой о каком-либо ресурсе.
2. **Обработка запроса на сервере:** Сервер обрабатывает запрос, выполняет необходимые действия (например, извлекает данные из базы данных) и формирует ответ.
3. **Ответ клиента:** Сервер отправляет ответ в виде HTML-страницы, изображения или других данных.

Когда используется HTTPS, весь этот процесс проходит через защищённое соединение, что гарантирует, что данные не могут быть перехвачены злоумышленниками.

Основные различия между HTTP и HTTPS

1. **Шифрование:** HTTP не использует шифрование, а HTTPS применяет SSL/TLS для шифрования передаваемых данных.
2. **Порты:** HTTP обычно использует порт 80, тогда как HTTPS — порт 443.
3. **Безопасность:** HTTPS обеспечивает более высокий уровень безопасности, защищая данные от возможных атак, таких как MITM (man-in-the-middle), где злоумышленники могут перехватить или изменять передаваемую информацию.

HTTPS широко используется для сайтов, где важна конфиденциальность данных, а HTTP — для менее чувствительных запросов, хотя с развитием интернета многие сервисы уже перешли на HTTPS.

Если вы разрабатываете приложение, которое взаимодействует с сервером, важно понимать различие между этими протоколами и использовать HTTPS для защиты данных пользователей.

Виды запросов (GET, POST, PUT, DELETE)

Когда мы работаем с веб-сервисами, чаще всего взаимодействуем с ними через HTTP-запросы. Каждый запрос имеет свой тип, который определяет, как сервер должен обработать запрос. Рассмотрим четыре основные виды HTTP-запросов: **GET**, **POST**, **PUT** и **DELETE**. Эти запросы часто называются методами HTTP.

GET

Метод **GET** используется для получения данных с сервера. Это самый распространённый тип запроса, который выполняет браузер каждый раз, когда вы открываете веб-страницу. GET-запрос не изменяет данные на сервере, он только запрашивает их.
Пример использования: Когда вы открываете веб-страницу, ваш браузер отправляет GET-запрос на сервер, чтобы получить HTML-код этой страницы.
Особенности:

- Данные передаются в URL-запросе (в строке запроса), например, example.com/search?q=swift.
- GET-запросы не должны использоваться для изменения данных.
- Обычно такие запросы имеют ограничения по размеру (поскольку данные передаются в строке URL).

POST

Метод **POST** используется для отправки данных на сервер с целью их обработки. POST-запросы обычно применяются для отправки форм на веб-сайтах, например, при регистрации пользователя или при оформлении заказа.
Пример использования: Когда вы отправляете форму регистрации на сайте, браузер отправляет POST-запрос, который включает введённые вами данные, такие как имя, email и пароль.
Особенности:

- Данные отправляются в теле запроса, а не в URL, что позволяет передавать большие объемы данных.
- POST-запросы могут изменять состояние сервера (например, добавлять новые записи в базу данных).

PUT

Метод **PUT** используется для отправки данных на сервер с целью замены существующих данных. Этот метод часто используется в API для обновления ресурсов на сервере. Когда вы отправляете PUT-запрос, вы как бы говорите серверу: “Замените текущие данные новыми”.
Пример использования: Представьте, что у вас есть API для редактирования профиля пользователя. Когда вы отправляете PUT-запрос с обновлёнными данными (например, новым email), сервер заменяет старые данные на новые.**Особенности:**

- PUT-запросы обычно применяются для замены целого ресурса, а не для частичных изменений.

• Они также могут быть идемпотентными, что означает, что если вы повторно отправите одинаковый запрос, результат не изменится.

DELETE

Метод **DELETE** используется для удаления данных с сервера. Этот метод позволяет серверу удалять ресурсы, которые больше не нужны.
Пример использования: Если вы хотите удалить свой аккаунт на сайте, сайт может отправить DELETE-запрос на сервер для удаления данных о вашем аккаунте.
Особенности:
• DELETE-запросы могут быть использованы для удаления записей или других данных.
• Обычно сервер возвращает подтверждение удаления или сообщение об ошибке, если удаление не удалось.

Когда и как использовать эти методы
1. **GET** — используйте для запросов, которые не изменяют данные на сервере (например, получение списка пользователей или информации о товаре).
2. **POST** — используйте для создания новых ресурсов на сервере или отправки данных, которые должны быть обработаны (например, отправка формы).
3. **PUT** — используйте для обновления существующих данных на сервере (например, изменение информации о пользователе).
4. **DELETE** — используйте для удаления ресурсов с сервера (например, удаление записи о пользователе).

Важным моментом является то, что каждый из этих методов имеет свою специфику в контексте взаимодействия с сервером. Правильное использование методов HTTP поможет сделать взаимодействие с сервером более понятным и логичным, а также улучшит безопасность вашего приложения.

Создание моделей данных (Decodable, Encodable, Codable)

Когда мы разрабатываем клиент-серверные приложения, важно правильно работать с данными, которые приходят от сервера и которые мы отправляем на сервер. Для этого в Swift есть удобные механизмы для сериализации и десериализации данных, такие как **Decodable**, **Encodable** и **Codable**. Давайте разберемся, что это такое и как они используются.

Decodable
Decodable — это протокол, который позволяет преобразовать данные, полученные от сервера, в объекты Swift. Например, когда мы получаем JSON с серверного API, мы хотим преобразовать этот JSON в объект, с которым нам будет проще работать в коде.

Предположим, что сервер возвращает нам JSON с информацией о пользователе. Структура может быть такой:

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

Чтобы десериализовать эти данные в объект Swift, мы можем создать модель, которая будет соответствовать этому JSON:

```
struct User: Decodable {
    let id: Int
    let name: String
    let email: String
}
```

Теперь с помощью JSONDecoder мы можем преобразовать полученные данные в объект:

```
let jsonData = // данные, полученные от сервера
do {
    let user = try JSONDecoder().decode(User.self, from: jsonData)
    print(user.name)
} catch {
    print("Ошибка при декодировании: \(error)")
}
```

Таким образом, **Decodable** помогает нам получать данные с сервера и преобразовывать их в удобные для работы объекты.

Encodable

Encodable — это протокол, который позволяет преобразовать объекты Swift в формат, подходящий для отправки на сервер. Например, когда мы хотим отправить информацию о новом пользователе на сервер, мы можем преобразовать наш объект в JSON.

Предположим, что мы создаем нового пользователя. У нас есть объект типа User, и нам нужно отправить его данные в JSON-формате:

```
struct User: Encodable {
    let id: Int
    let name: String
    let email: String
}
```

Для отправки этого объекта на сервер в формате JSON мы используем JSONEncoder:

```
let user = User(id: 1, name: "John Doe", email: "john.doe@example.com")
do {
    let jsonData = try JSONEncoder().encode(user)
    // теперь jsonData готово для отправки на сервер
} catch {
    print("Ошибка при кодировании: \(error)")
}
```

Encodable используется, когда нам нужно отправить объект на сервер в нужном формате (например, JSON).

Codable

Codable — это объединение протоколов **Decodable** и **Encodable**, и если вам нужно и декодировать, и кодировать данные, вы можете использовать этот протокол. Таким образом, **Codable** позволяет работать как с входящими, так и с исходящими данными.

Когда вы создаете модель, которая должна поддерживать и кодирование, и декодирование, достаточно применить протокол **Codable**:

```
struct User: Codable {
   let id: Int
   let name: String
   let email: String
}
```

Теперь вы можете и декодировать, и кодировать этот объект:

```
let user = User(id: 1, name: "John Doe", email: "john.doe@example.com")
do {
   let jsonData = try JSONEncoder().encode(user)  // кодируем объект в JSON
   let decodedUser = try JSONDecoder().decode(User.self, from: jsonData)  // декодируем обратно
   print(decodedUser.name)
} catch {
   print("Ошибка при кодировании или декодировании: \(error)")
}
```

Когда использовать?

1. **Decodable**: если вы получаете данные с сервера и хотите преобразовать их в объекты Swift.
2. **Encodable**: если вам нужно отправить данные на сервер в виде JSON.
3. **Codable**: если вам нужно и декодировать, и кодировать данные (в большинстве случаев).

Каждый из этих протоколов помогает вам легко и удобно работать с данными, особенно когда приложение взаимодействует с внешними API. Они обеспечивают мощный и гибкий инструмент для сериализации и десериализации данных в Swift.

Response. JSON. Mock-данные

Когда мы начинаем разрабатывать клиент-серверные приложения, неизбежно сталкиваемся с ответами от сервера, различными форматами данных (например, JSON) и необходимостью работы с тестовыми данными, если сервер еще не готов. Давайте разберемся, как все это работает в реальной разработке, и как мы можем это эффективно использовать.

Что такое Response?

Response — это, по сути, ответ сервера на наш запрос. Когда мы посылаем запрос на сервер, он всегда возвращает какой-то отклик: это могут быть как сами данные, так и метаинформация о запросе. В этом отклике обычно присутствует статусный код (например, 200 OK для успешного запроса), заголовки, и, конечно, тело ответа.

Представим, что мы сделали запрос на получение данных о пользователе, и вот пришел ответ от сервера. Мы можем получить эту информацию с помощью URLSession в iOS.

```swift
let url = URL(string: "https://api.example.com/user")!
let task = URLSession.shared.dataTask(with: url) { data, response, error in
    guard let data = data, error == nil else {
        print("Ошибка: \(String(describing: error))")
        return
    }

    if let httpResponse = response as? HTTPURLResponse, httpResponse.statusCode == 200 {
        print("Успешный запрос!")
        // Далее обрабатываем полученные данные
    }
}
task.resume()
```

Здесь в переменной data мы получаем информацию от сервера, а response может содержать информацию о статусе запроса.

Что такое JSON?

Теперь, как только сервер вернул нам ответ, скорее всего, данные будут в формате **JSON**. Это один из самых популярных форматов передачи данных в веб-разработке. Что важно, JSON отлично читается как людьми, так и машинами, что делает его удобным для обмена данными между клиентом и сервером.

Когда мы получаем JSON, он, как правило, представляет собой текст, который мы должны преобразовать в объекты, с которыми можем работать в коде. Для этого в iOS есть замечательные инструменты — JSONDecoder и JSONEncoder.

Предположим, что сервер отправил нам такой JSON-ответ:

```
{
   "id": 1,
   "name": "John Doe",
   "email": "john.doe@example.com"
}
```

Чтобы удобно работать с этими данными в Swift, мы используем модель:

```swift
struct User: Decodable {
    let id: Int
    let name: String
    let email: String
}

func fetchUserData() {
    let url = URL(string: "https://api.example.com/user")!
    URLSession.shared.dataTask(with: url) { data, response, error in
        guard let data = data, error == nil else {
            print("Ошибка: \(String(describing: error))")
            return
        }

        do {
            let user = try JSONDecoder().decode(User.self, from: data)
            print("Имя пользователя: \(user.name)")
        } catch {
            print("Ошибка при декодировании: \(error)")
        }
    }.resume()
}
```

Теперь у нас есть объект user, с которым мы можем работать как с обычной структурой.

Мок-данные

Когда сервер еще не готов или когда нам нужно тестировать UI, бывает полезно использовать **Mock-данные**. Это такие данные, которые мы создаем вручную, чтобы заменить реальное подключение к серверу. В таком случае мы можем сэкономить время на тестирование и проверку интерфейса.

Как это работает? Пример с мок-данными:

```
func mockUserData() -> User {
   return User(id: 1, name: "Mock User", email: "mock.user@example.com")
}

func fetchMockUserData() {
   let user = mockUserData()
   print("Полученные данные: \(user.name), \(user.email)")
}
```

Тут мы не подключаемся к серверу, а просто возвращаем заранее определенные данные. Так можно легко и быстро тестировать интерфейс приложения, не завися от бэкенда.

Когда использовать Mock-данные?

Использование мок-данных необходимо в следующих случаях:

1. **Отсутствие реального сервера** — если сервер не готов или не доступен, можно работать с мок-данными.
2. **Тестирование интерфейса** — можно тестировать внешний вид и взаимодействие с UI, не ожидая, пока сервер не предоставит данные.
3. **Нагрузочное тестирование** — генерируем фальшивые данные для имитации различных сценариев и тестируем производительность приложения.

Понимание работы с **Response**, **JSON** и **Mock-данными** критично для разработки. Мы можем эффективно получать и обрабатывать данные от сервера с помощью декодирования JSON в объекты, а также работать с мок-данными для тестирования и отладки нашего приложения.

Обработка запросов в приложении

Давайте посмотрим, как обработка запросов может быть организована внутри **UIViewController**, чтобы вы могли увидеть весь процесс на конкретном примере. Мы будем работать с запросом, который получаем через **URLSession**, и обрабатывать ответ внутри контроллера, при этом не забывая обновлять интерфейс.

1. Подготовка

Предположим, у нас есть экран с лейблом, на котором мы хотим отобразить имя пользователя, полученное с сервера. Начнем с создания интерфейса:

```swift
import Foundation
import UIKit

struct User: Decodable {
    let id: Int
    let name: String
    let email: String
}

class UserViewController: UIViewController {

    // Лейбл для отображения имени пользователя
    var nameLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Инициализация лейбла
        nameLabel = UILabel()
        nameLabel.frame = CGRect(x: 20, y: 100, width: 300, height: 40)
        nameLabel.textAlignment = .center
        self.view.addSubview(nameLabel)

        // Запрос на сервер
        fetchData()
    }

    // Функция для получения данных с сервера
    func fetchData() {
        let url = URL(string: "https://api.example.com/user")!
        let task = URLSession.shared.dataTask(with: url) { [weak self] (data, response, error) in
            // Проверка ошибок
            if let error = error {
                self?.showError(error.localizedDescription)
                return
            }

            guard let data = data else {
                self?.showError("Нет данных от сервера")
                return
            }

            // Парсим данные
            do {
                let user = try JSONDecoder().decode(User.self, from: data)
                DispatchQueue.main.async {
                    self?.updateUI(with: user)
                }
            } catch {
                self?.showError("Ошибка при обработке данных")
            }
        }
        task.resume()
    }

    // Обновление интерфейса
    func updateUI(with user: User) {
        nameLabel.text = user.name
    }

    // Показ ошибки
    func showError(_ message: String) {
        let alert = UIAlertController(title: "Ошибка", message: message, preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .default))
        present(alert, animated: true)
    }
}
```

2. Что здесь происходит?

1. **Интерфейс:**

Мы создаем лейбл nameLabel на экране, который будет отображать имя пользователя. Он добавляется в иерархию в viewDidLoad.

2. **Запрос:**

В методе fetchData() мы создаем запрос к серверу, используя **URLSession**. После того, как данные получены, мы проверяем на наличие ошибок, парсим данные и передаем их в основной поток для обновления UI.

3. **Обработка данных:**

Сервер возвращает данные в формате JSON, которые мы парсим с помощью JSONDecoder. Если все прошло успешно, обновляем текст лейбла на главном потоке с помощью DispatchQueue.main.async.

4. **Ошибки:**

Если произошла ошибка на любом этапе (неудачный запрос, ошибка при парсинге), мы показываем ошибку с помощью **UIAlertController**.

3. Что важного?

• **Асинхронность:** Запрос к серверу выполняется асинхронно, то есть приложение не блокирует основной поток. Все действия, связанные с обновлением интерфейса, выполняются на главном потоке с помощью DispatchQueue.main.async.

• **Обработка ошибок:** Мы тщательно обрабатываем ошибки, как на уровне сети, так и при декодировании данных. Это важно, чтобы пользователь получил обратную связь о проблемах.

• **Обновление интерфейса:** Все изменения в интерфейсе происходят в главном потоке, что является обязательным для правильного отображения UI в iOS-приложениях.

4. Как все это работает на практике?

Представьте, что вы работаете над проектом и столкнулись с задачей обработать запросы в приложении. Зачастую, вы будете выполнять HTTP-запросы для получения данных с сервера или отправки данных на сервер.

Когда вы хотите запросить данные с сервера, например, получить информацию о пользователе, вы начинаете с того, что создаете URL, по которому сервер должен вернуть данные. Весь процесс начинается с того, что вы создаете метод, например, fetchUserData(). Внутри этого метода вы создаете объект URLRequest с нужным вам URL-адресом и конфигурируете его (например, добавляете HTTP-метод). Затем, используя URLSession, вы отправляете запрос и получаете ответ.

После того как ответ пришел, нужно его обработать. Например, если сервер вернул вам JSON-данные, вы преобразуете их в объекты с помощью Codable моделей, используя JSONDecoder. Всё это, конечно же, происходит в фоне, и в конце, после того как данные успешно получены и декодированы, вы можете обновить интерфейс пользователя, отобразив эти данные в UILabel, UITableView или другом элементе на экране.

OpenAPI

OpenAPI (ранее известный как Swagger) — это спецификация для описания RESTful API. Она предоставляет стандартный формат для документирования API, который упрощает понимание того, как взаимодействовать с сервером. По сути, OpenAPI — это средство, которое позволяет разработчикам и системам лучше понимать и тестировать API, без необходимости вручную искать информацию о каждом эндпоинте, методах, параметрах и типах данных.

OpenAPI часто используется для автоматической генерации документации, создания клиентских библиотек, а также для тестирования API. В интернете существует множество

открытых API, которые предоставляют различные данные (например, погоду, новости, изображения и т.д.) и позволяют работать с ними для практики.

С помощью OpenAPI можно описывать структуру API в виде файла (обычно в формате YAML или JSON), который содержит описание доступных запросов, их параметров, возвращаемых данных и кода состояния. Это позволяет как пользователям API, так и инструментам автоматически генерировать код для работы с API, тестировать его и поддерживать актуальность документации.

В интернете вы найдете множество открытых (public) API, которые можно использовать для практики, таких как:

- **JSONPlaceholder** — фиктивные данные для тестирования.
- **OpenWeatherMap** — данные о погоде.
- **The Dog API** — изображения и информация о собаках.
- **NewsAPI** — новости и статьи.

Эти API часто имеют документацию в формате OpenAPI, что делает их удобными для разработчиков, позволяя быстро ознакомиться с доступными возможностями и методами работы с ними.