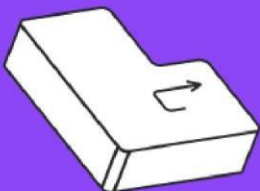


# Лекция 5.

## Работа с сетью

### В Objective-C.

Знакомство с работой с сетью в языке Objective – C. Изучим некоторые базовые понятия для работы с сетью. Поговорим про некоторые базовые классы, например, `NSURLSession` и другие, а также про форматы данных и передачу непосредственно на сервер, также научимся отлаживать нашу программу и отображать `WebView`.



# Оглавление

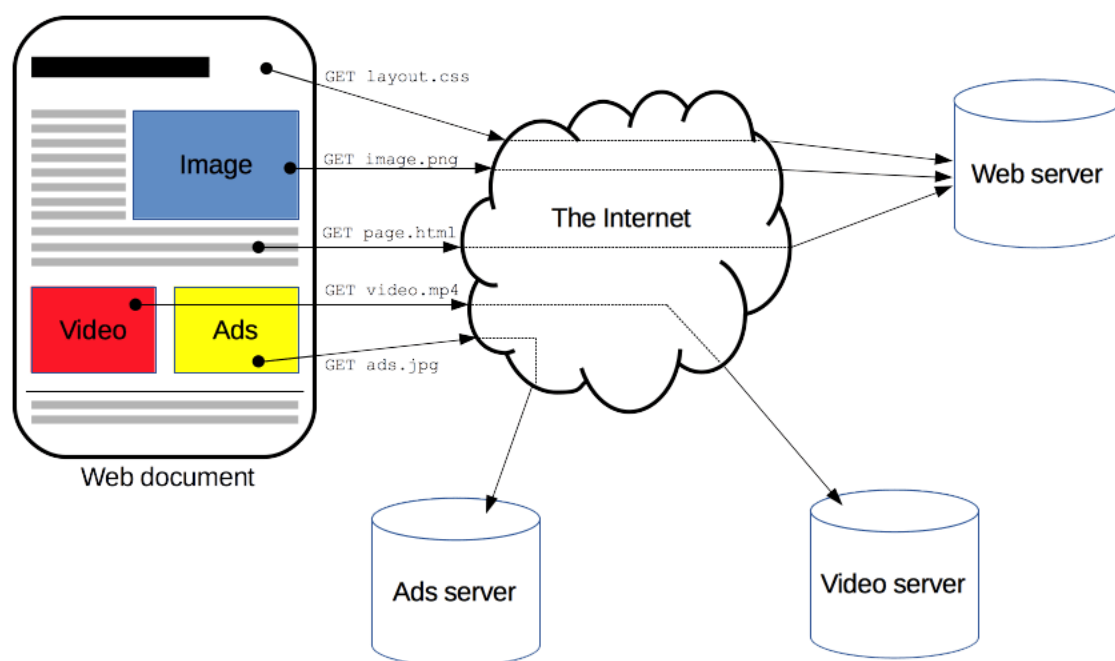
Обзор HTTP	3
Как работает HTTP	3
Компоненты систем на основе HTTP	4
Цепочка клиент-сервер	4
Клиент: пользовательский агент	4
Веб-сервер	5
Прокси	5
<b>Основные аспекты HTTP</b>	5
HTTP расширяемый	6
HTTP не имеет состояния, но не без сеанса	6
HTTP и соединения	6
HTTP-поток	6
<b>HTTP против HTTPS</b>	7
GET запрос	8
POST запрос	9
<b>О Сети</b>	10
С одного взгляда	11
Узнайте, почему работать в сети сложно	13
OS X и iOS предоставляют API на многих уровнях	13
Безопасное общение — ваша ответственность	14
Сеть должна быть динамичной и асинхронной	14
<b>Базовые классы в работе с сетью в рамках IOS SDK</b>	14
Примечание	15
Типы URL-сессий	15
Типы задач URL-сессии	16
Асинхронность и URL-сессии	16
<b>Введение Сетевые функции Objective – C</b>	17

# Базовые понятия

## Обзор HTTP

HTTP (протокол передачи гипертекста) — это набор правил для передачи файлов, таких как текст, изображения, звук, видео и другие мультимедийные файлы, через Интернет. Как только пользователь открывает свой веб-браузер, он косвенно использует HTTP. HTTP — это прикладной протокол, работающий поверх набора протоколов TCP/IP, который составляет основу Интернета.

Клиенты и серверы общаются, обмениваясь отдельными сообщениями (в отличие от потока данных). Сообщения, отправляемые клиентом, обычно веб-браузером, называются запросами, а сообщения, отправляемые сервером в качестве ответа, называются ответами.



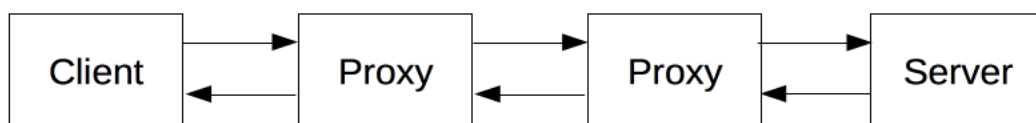
## Как работает HTTP

Через протокол HTTP происходит обмен ресурсами между клиентскими устройствами и серверами через Интернет. Клиентские устройства отправляют серверу запросы на ресурсы, необходимые для загрузки веб-страницы; серверы отправляют ответы обратно клиенту для выполнения запросов. Запросы и ответы совместно используют вложенные документы, такие как данные об изображениях, тексте, текстовом макете и т. д., которые объединяются клиентским веб-браузером для отображения полного файла веб-страницы. В дополнение к файлам веб-страниц, которые он может обслуживать, веб-сервер содержит демон HTTP, программу, которая ожидает HTTP-запросов и обрабатывает их, когда они поступают.

Веббраузер — это HTTP-клиент, который отправляет запросы на серверы. Когда пользователь браузера вводит запросы файлов, «открывая» веб-файл, вводя URL-адрес или щелкая гипертекстовую ссылку, браузер создает HTTP-запрос и отправляет его на адрес интернетпротокола (IP-адрес), указанный URL-адресом. Демон HTTP на целевом сервере получает запрос и отправляет обратно запрошенный файл или файлы, связанные с запросом.

## Компоненты систем на основе HTTP

HTTP — это клиент-серверный протокол: запросы отправляются одним объектом — агентом пользователя (или прокси-сервером от его имени). В большинстве случаев агентом пользователя является веб-браузер, но это может быть что угодно, например, робот, который просматривает Интернет для заполнения и поддержки индекса поисковой системы. Каждый отдельный запрос отправляется на сервер, который обрабатывает его и предоставляет ответ, называемый ответом. Между клиентом и сервером существует множество объектов, которые в совокупности называются прокси, которые выполняют различные операции и действуют, например, как шлюзы или кэши.



## Цепочка клиент-сервер

На самом деле между клиентом и сервером, обрабатывающим запрос, больше компьютеров: маршрутизаторы, модемы и прочее. Благодаря многоуровневой структуре высокоуровневых технологий они скрыты в сетевом и транспортном уровнях. HTTP находится сверху, на прикладном уровне. Хотя базовые уровни важны для диагностики сетевых проблем, они в основном не имеют отношения к описанию HTTP.

## Клиент: пользовательский агент

Пользовательский агент — это любой инструмент, который действует от имени пользователя. Эту роль в основном выполняет веб-браузер или мобильное приложение, но ее также могут выполнять программы, используемые инженерами и веб-разработчиками для отладки своих приложений (Postman). Клиент всегда является объектом, инициирующим запрос. Он никогда

не является сервером (хотя с годами были добавлены некоторые механизмы для имитации сообщений, инициированных сервером).

## Веб-сервер

На противоположной стороне канала связи находится сервер, который подает документ по запросу клиента. Сервер виртуально выглядит как одна машина; но на самом деле это может быть набор серверов, разделяющих нагрузку (балансировка нагрузки), или сложная часть программного обеспечения, опрашивающая другие компьютеры (например, кеш, сервер БД или серверы электронной коммерции), полностью или частично генерирующая документ по запросу.

Сервер не обязательно представляет собой одну машину, но на одной машине может быть размещено несколько экземпляров серверного программного обеспечения. С HTTP/1.1 и заголовком Host они могут даже иметь один и тот же IP-адрес.

## Прокси

Между клиентом и сервером многочисленные компьютеры и машины передают HTTP-сообщения. Из-за многоуровневой структуры веб-стека большинство из них работают на транспортном, сетевом или физическом уровнях, становясь прозрачными на уровне HTTP и потенциально оказывая значительное влияние на производительность. Те, которые работают на прикладных уровнях, обычно называются прокси. Они могут быть прозрачными, пересылая запросы, которые они получают, не изменяя их каким-либо образом, или непрозрачными, и в этом случае они каким-то образом изменяют запрос, прежде чем передать его на сервер.

Прокси могут выполнять множество функций:

- кэширование (кэш может быть общедоступным или частным, как кэш клиента)
- фильтрация (например, антивирусное сканирование или родительский контроль)
- балансировка нагрузки (чтобы несколько серверов могли обслуживать разные запросы)
- аутентификация (для управления доступом к различным ресурсам)
- ведение журнала (позволяющее хранить историческую информацию)

## Основные аспекты HTTP

HTTP, как правило, разработан таким образом, чтобы быть простым и понятным для человека, даже с добавленной сложностью, представленной в HTTP/2 за счет инкапсуляции сообщений HTTP в фреймы. Сообщения HTTP могут быть прочитаны и поняты людьми, что упрощает тестирование для разработчиков и снижает сложность для новичков.

### ➤ HTTP расширяемый

Представленные в HTTP/1.0 заголовки HTTP упрощают расширение этого протокола и экспериментирование с ним. Новая функциональность может быть введена даже простым соглашением между клиентом и сервером о семантике нового заголовка.

### ➤ HTTP не имеет состояния, но не без сеанса

HTTP не имеет состояния: нет связи между двумя запросами, последовательно выполняемыми по одному и тому же соединению. Это немедленно создает проблемы для пользователей, пытающихся последовательно взаимодействовать с определенными страницами, например, используя корзину для покупок в электронной коммерции. Но хотя ядро самого HTTP не имеет состояния, файлы cookie HTTP позволяют использовать сеансы с отслеживанием состояния. С помощью расширяемости заголовков в рабочий процесс добавляются файлы cookie HTTP, что позволяет создавать сеансы для каждого HTTP-запроса для совместного использования одного и того же контекста или одного и того же состояния.

### ➤ HTTP и соединения

Соединение контролируется на транспортном уровне и, следовательно, принципиально выходит за рамки HTTP. HTTP не требует, чтобы базовый транспортный протокол был основан на соединении; требуется только, чтобы он был надежным или не терял сообщения (как минимум, в таких случаях выдавал ошибку). Среди двух наиболее распространенных транспортных протоколов в Интернете TCP является надежным, а UDP — нет. Таким образом, HTTP опирается на стандарт TCP, основанный на соединении. Прежде чем клиент и сервер смогут обменяться парой HTTP-запрос/ответ, они должны установить TCP-соединение, процесс, для которого требуется несколько круговых обходов. По умолчанию

HTTP/1.0 открывает отдельное TCP-соединение для каждой пары HTTP-запрос/ответ. Это менее эффективно, чем совместное использование одного TCP-соединения, когда несколько запросов отправляются друг за другом. Чтобы смягчить этот недостаток, в HTTP/1.1 была введена конвейерная обработка (которую оказалось трудно реализовать) и постоянные соединения: базовое TCP-соединение можно частично контролировать с помощью заголовка Connection. HTTP/2 пошел еще дальше, мультиплексируя сообщения по одному соединению, помогая сохранять соединение теплым и более эффективным. В настоящее время проводятся эксперименты по разработке лучшего транспортного протокола, более подходящего для HTTP. Например, Google экспериментирует с QUIC, который основан на UDP, чтобы обеспечить более надежный и эффективный транспортный протокол.

### ➤ HTTP-поток

Когда клиент хочет связаться с сервером, конечным сервером или промежуточным прокси, он выполняет следующие шаги:

Открывает TCP-соединение: TCP-соединение используется для отправки запроса или нескольких и получения ответа. Клиент может открыть новое соединение, повторно использовать существующее соединение или открыть несколько TCP-соединений с серверами.

Отправить HTTP-сообщение: HTTP-сообщения (до HTTP/2) удобочитаемы. В HTTP/2 эти простые сообщения инкапсулируются во фреймы, что делает невозможным их прямое чтение, но принцип остается прежним. Например:

```
GET / HTTP/1.1
```

```
Host: developer.mozilla.org
```

```
Accept-Language: fr
```

Прочитайте ответ, отправленный сервером. Например:

```
HTTP/1.1 200 OK
```

```
Date: Sat, 09 Oct 2010 14:28:02 GMT
```

```
Server: Apache
```

```
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
```

```
ETag: "51142bc1-7449-479b075b2891b"
```

```
Accept-Ranges: bytes
```

```
Content-Length: 29769
```

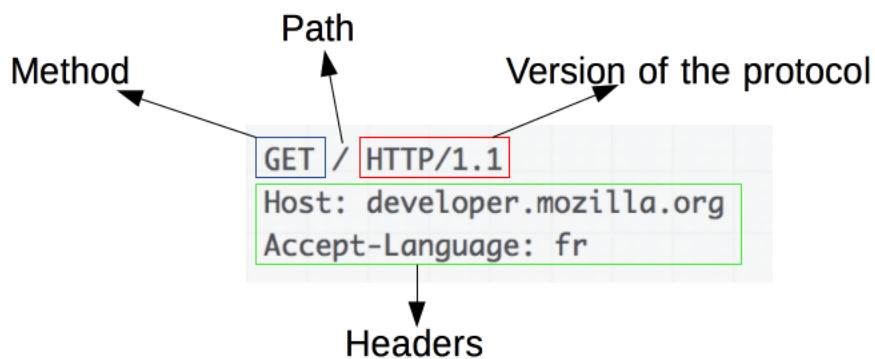
```
Content-Type: text/html
```

Закройте или повторно используйте соединение для дальнейших запросов.

## HTTP против HTTPS

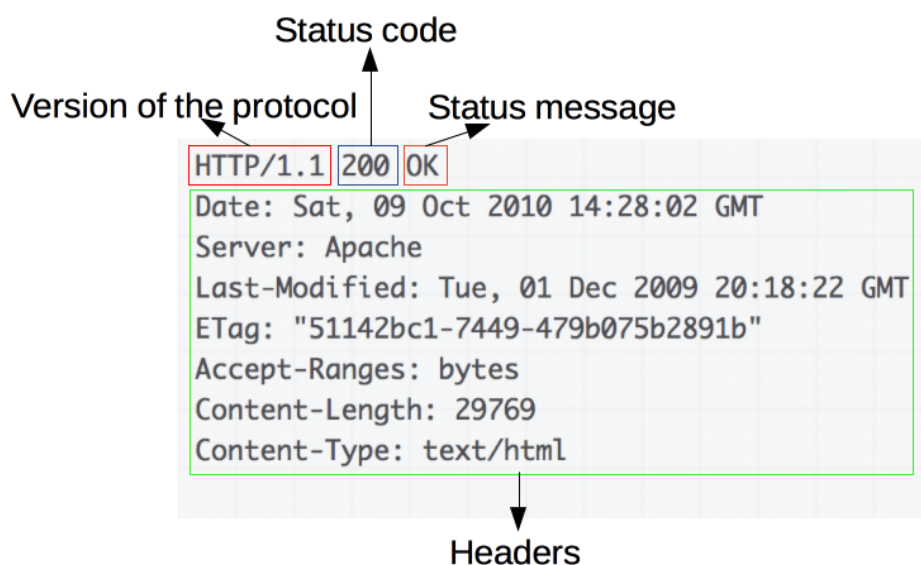
HTTPS — это использование Secure Sockets Layer (SSL) или Transport Layer Security (TLS) в качестве подуровня при обычном уровне приложений HTTP. HTTPS шифрует и расшифровывает запросы пользовательских HTTP-страниц, а также страницы, возвращаемые веб-сервером. Он также защищает от подслушивания и атак «человек посередине» (MitM). HTTPS был разработан компанией Netscape. Переход с HTTP на HTTPS считается выгодным, поскольку он предлагает дополнительный уровень безопасности и доверия.

Как и говорилось выше, существует два типа HTTP-сообщений: запросы и ответы, каждый из которых имеет собственный формат. Запросы. Пример HTTP-запроса:



### Запросы состоят из следующих элементов:

Метод HTTP, обычно глагол, такой как GET, POST, или существительное, такое как OPTIONS или HEAD, который определяет операцию, которую хочет выполнить клиент. Как правило, клиент хочет получить ресурс (используя GET) или опубликовать значение используя POST, хотя в других случаях может потребоваться больше операций (PUT, DELETE). **Ответы. Пример ответа:**



### Ответы состоят из следующих элементов:

- Версия протокола HTTP, которому они следуют.
- Код состояния, указывающий, был ли запрос успешным или нет, и почему.
- Сообщение о состоянии, неавторитетное краткое описание кода состояния.
- Заголовки HTTP, например, для запросов.
- Необязательно тело, содержащее выбранный ресурс.

#### ➤ GET запрос

GET – это простейший метод HTTP запроса, и именно его использует браузер для загрузки вебстраниц. Он используется для запроса содержимого, расположенного по определенному



URL. Содержимое может быть, например, веб-страницей, рисунком или аудиофайлом. По соглашению, GET запросы осуществляют только чтение и в соответствии с W3C стандартом не должны быть использованы в операциях, изменяющих серверную сторону. Например, мы не будем использовать GET запрос для отсылания формы или пересылки фотографии, потому что эти операции требуют некоторых изменений на серверной стороне (мы будем использовать в этих случаях POST).

### ➤ POST запрос

POST посылает данные для дальнейшей обработки на URL. Параметры включены в тело запроса, использующего тот же формат, что и GET. Например, если мы хотим запостить форму, содержащую два поля, имя и возраст, то мы пошлем что-то похожее на `name=Martin&age=29` в теле запроса. Такой способ пересылки параметров широко используется в веб-страницах. Наиболее популярные случаи – это формы. Когда мы заполняем форму на сайте и кликаем Submit, вероятнее всего запрос будет POST.

Данные POST запроса могут быть структурированы с использованием разных форматов. Параметры обычно отформатированы в соответствии со стандартами `form-url`-кодирования (в соответствии с W3C HTML стандартом). Это формат по умолчанию и широко используется во многих браузерах. Наш метод принимает словарь Dictionary в качестве аргумента, но мы не можем послать по HTTP соединению словарь Dictionary, потому что это внутренний тип Swift. Для пересылки по HTTP-соединению нам надо создать распознаваемое представление словаря. Это как общение с иностранцем. Мы переводим наше сообщение на универсальный язык, а он переводит с универсального языка уже на свой родной. Универсальный язык в HTTP — это W3C стандарт, наш язык — это Swift, язык получателя нам неизвестен.

## WebView

С помощью средств языка программирования мы можем посылать запросы на удаленный сервер. Именно это делают браузеры перед отображением веб-страницы. Отличие только в содержимом ответа. Веб-страницы форматированны с помощью HTML стандарта, который определяет ряд правил на то, как графически определить различные теги разметки. Эти правила кажутся простыми, но отображение целой страницы, следующей W3C стандарту – это сложная задача. К счастью, в iOS есть встроенный компонент `UIWebView`, который использует хорошо известный движок `WebKit`, и интерпретирует HTML/CSS/JavaScript, и отображает целые веб-страницы внутри `UIView`.

Есть несколько случаев, когда мы хотим контролировать поток навигации. Например, мы хотим знать, когда определенный контент или определенный URL загружен.

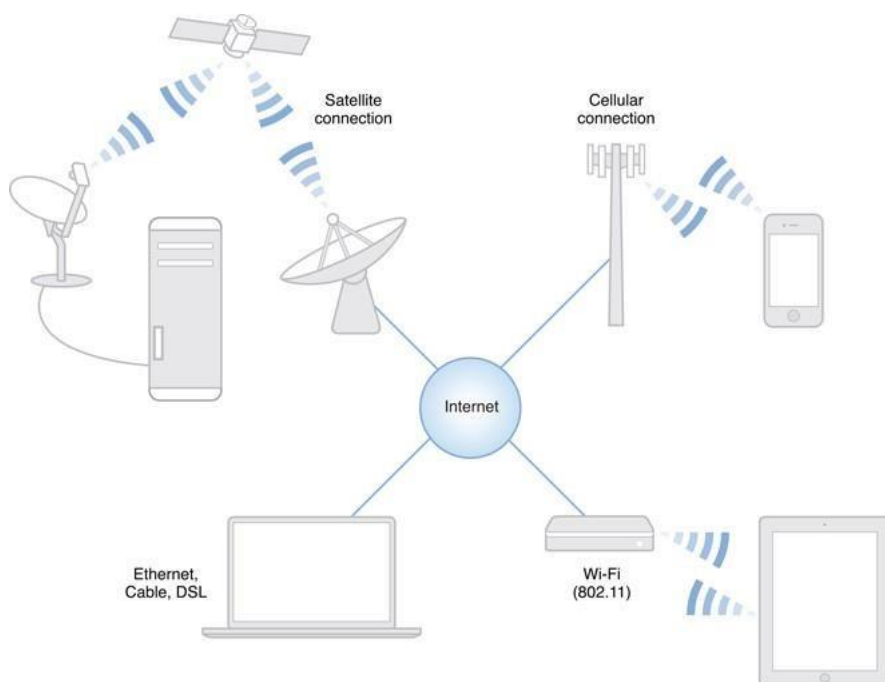
Или возможно мы реализуем безопасный браузер для детей, мы хотим заблокировать пользователя от загрузки страниц, попадающих под определенные критерии, как секс или наркотики. Для всех таких типов кастомизации мы создаем экземпляр класса, который реализует `UIWebViewDelegate` протокол в качестве делегата `UIWebView`. Мы можем реализовать следующие методы:

```
webView:shouldStartLoadWithRequest:navigationType:  
webViewDidStartLoad:      webViewDidFinishLoad:  
webView:didFailLoadWithError:
```

С помощью первого метода мы можем контролировать поток навигации, разрешая или блокируя специфические запросы. Остальные три метода – информационные события (имена методов дают хорошее представление о событии).

## О Сети

Мир сетей сложен. Пользователи могут подключаться к Интернету, используя широкий спектр технологий — кабельные модемы, DSL, Wi-Fi, сотовые соединения, спутниковые восходящие каналы, Ethernet и даже традиционные акустические модемы. Каждое из этих подключений имеет различные характеристики, в том числе различия в пропускной способности, задержке, потере пакетов и надежности.



Еще больше усложняет то, что подключение пользователя к Интернету не дает полной картины. На пути от пользователя к интернет-серверу сетевые данные пользователя проходят через от одного до десятков физических межсоединений, любое из которых может быть высокоскоростной линией OC-768 (почти 40 миллиардов бит в секунду), скудный модем на 300 бод (300 бит в секунду) или что-то среднее. Хуже того, в любой момент скорость подключения пользователя к серверу может резко измениться — кто-то может включить микроволновую печь, которая мешает связи пользователя по Wi-Fi, пользователь может уйти или выехать за пределы сотовой связи, кто-то на другой конец света может начать скачивать большой фильм с сервера, к которому пытается получить доступ пользователь, и так далее. Как разработчик сетевого программного обеспечения, ваш код должен уметь адаптироваться к изменяющимся условиям сети, включая производительность, доступность и надежность.

## С одного взгляда

Сети по своей природе ненадежны, а сотовые сети вдвойне ненадежны. В результате хороший сетевой код имеет тенденцию быть несколько сложным. Среди прочего ваше программное обеспечение должно:

Передавайте ровно столько данных, сколько необходимо для выполнения задачи. Минимизация объема отправляемых и получаемых данных продлевает срок службы батареи и может снизить затраты пользователей на лимитные подключения к Интернету, которые оплачиваются по мегабайтам. По возможности избегайте тайм-аутов. Вы, вероятно, не хотите,

чтобы что – то перестала загружаться только потому, что процесс загрузки занял слишком много времени. Вместо этого предоставьте пользователю возможность отменить операцию.

В некоторых редких случаях данные становятся неактуальными, если они существенно задерживаются. В этих ситуациях может иметь смысл использовать протокол, который не передает пакеты повторно. Например, если вы пишете многопользовательскую игру в реальном времени, которая отправляет крошечные сообщения о состоянии на другое устройство через локальную сеть (LAN) или Bluetooth, часто лучше пропустить сообщение и сделать предположения о том, что происходит на другом устройстве. Устройство, чем позволить операционной системе ставить эти пакеты в очередь и доставлять их все сразу. Однако в большинстве случаев, если вам не нужно поддерживать совместимость с существующими протоколами, вам обычно следует использовать TCP.

Разрабатывайте пользовательские интерфейсы, которые позволяют пользователю легко отменять транзакции, выполнение которых занимает слишком много времени. Если ваше приложение выполняет загрузку потенциально больших файлов, вы также должны предоставить способ приостановить эти загрузки и возобновить их позже.

Относитесь к неудачам изящно. Соединение может завершиться неудачно по ряду причин — сеть может быть недоступна, имя хоста, может быть, не разрешено успешно и т. д. При возникновении сбоев ваша программа должна продолжать функционировать в максимально возможной степени в автономном режиме. Чтобы еще больше усложнить ситуацию, иногда пользователь может иметь доступ к ресурсам только в определенных сетях. Например, AirPlay может подключаться к Apple TV только в той же сети. Доступ к корпоративным сетевым ресурсам возможен только во время работы или через виртуальную частную сеть (VPN). Визуальная голосовая почта может быть доступна только через сеть оператора сотовой связи (в зависимости от оператора). И так далее.

В частности, вам следует избегать интерфейсов, требующих от пользователя присматривать за вашей программой, когда сеть работает со сбоями.

Не отображайте модальные диалоговые окна, чтобы сообщить пользователю, что сеть не работает. Повторите попытку автоматически, когда сеть снова заработает. Не предупреждайте пользователя о сбоях подключения, которые не были инициированы пользователем.

Изящно снижайте производительность, когда производительность сети низкая. Поскольку пропускная способность между устройством пользователя и его интернет-провайдером ограничена, ваше приложение может получить доступ к другим устройствам в домашней сети пользователя гораздо быстрее, чем к серверам на другом конце света. Эта разница становится

еще больше, когда кто-то еще в локальной сети начинает использовать эту ограниченную полосу пропускания для других целей.

Выберите API, которые подходят для задачи. Если существует высокоуровневый API, который может удовлетворить ваши потребности, используйте его, а не внедряйте собственную реализацию с использованием низкоуровневых API. Если для того, чем вы занимаетесь, существует API, специфичный для вашей задачи (например, игровой API), используйте его.

Используя API самого высокого уровня, вы предоставляете операционной системе больше информации о том, что вы на самом деле пытаетесь выполнить, чтобы она могла более оптимально обработать ваш запрос. Эти высокоуровневые API-интерфейсы также решают многие из самых сложных сетевых проблем — кэширование, прокси-серверы, выбор хоста из нескольких IP-адресов и т. д. Если вы пишете свой собственный низкоуровневый код для выполнения тех же задач, вам придется самостоятельно справляться с этой сложностью (а также отлаживать и поддерживать рассматриваемый код).

Тщательно проектируйте свое программное обеспечение, чтобы свести к минимуму риски безопасности. Воспользуйтесь преимуществами технологий безопасности, таких как Secure Sockets Layer (SSL) и Transport Layer Security (TLS), чтобы предотвратить спуфинг и скрыть конфиденциальные данные от посторонних глаз, а также тщательно проверять ненадежный контент, чтобы предотвратить переполнение буфера и целых чисел.

Эта лекция поможет вам изучить эти концепции и многое другое.

## **Узнайте, почему работать в сети сложно**

Хотя написание сетевого кода может быть легким, для всех, кроме самых тривиальных сетевых потребностей, написать хороший сетевой код не так просто. В зависимости от потребностей вашего программного обеспечения ему может потребоваться адаптироваться к изменяющейся производительности сети, обрывам сетевых подключений, сбоям подключения и другим проблемам, вызванным ненадежностью самого Интернета.

## **OS X и iOS предоставляют API на многих уровнях**

Вы можете выполнить следующие сетевые задачи как в OS X, так и в iOS с идентичным или почти идентичным кодом:

1. Выполнение запросов HTTP/HTTPS, таких как запросы GET и POST
2. Установление соединения с удаленным хостом с шифрованием или аутентификацией или без них

3. Прослушивание входящих соединений
4. Отправка и получение данных с помощью протоколов без установления соединения
5. Публикация, просмотр и разрешение сетевых служб

## Безопасное общение — ваша ответственность

Надлежащая сетевая безопасность является необходимостью. Вы должны рассматривать все данные, отправленные вашим пользователем, как конфиденциальные и защищать их соответствующим образом. В частности, вы должны зашифровать его во время передачи и защитить от отправки не тому человеку или серверу.

Для этой цели большинство сетевых API OS X и iOS обеспечивают простую интеграцию с TLS. TLS является преемником протокола SSL. В дополнение к шифрованию данных по сети TLS аутентифицирует сервер с помощью сертификата для предотвращения спуфинга. Ваш сервер также должен предпринять шаги для аутентификации клиента. Эта аутентификация может быть простой, как пароль, или сложной, как токен аппаратной аутентификации, в зависимости от ваших потребностей.

Будьте осторожны со всеми входящими данными. Любые данные, полученные из ненадежного источника, могут быть вредоносными атаками. Ваше приложение должно тщательно проверять входящие данные и немедленно отбрасывать все, что выглядит подозрительно.

## Сеть должна быть динамичной и асинхронной

Сетевое окружение устройства может измениться в любой момент. Существует ряд простых (но разрушительных) сетевых ошибок, которые могут отрицательно сказаться на производительности и удобстве использования вашего приложения, например выполнение синхронного сетевого кода в основном потоке вашей программы, неспособность корректно обрабатывать сетевые изменения и так далее. Вы можете сэкономить много времени и сил, разработав свою программу так, чтобы избежать этих проблем с самого начала вместо того, чтобы отлаживать ее позже.

## Базовые классы в работе с сетью в рамках iOS SDK

NSURLSession Объект, который координирует группу связанных задач передачи данных по сети. Класс NSURLSession и связанные классы предоставляют API для загрузки данных и

отправки данных в конечные точки, указанные URL-адресами. Ваше приложение также может использовать этот API для выполнения фоновых загрузок, когда ваше приложение не запущено или, в iOS, когда ваше приложение приостановлено. Вы можете использовать связанные `NSURLSessionDelegate` и `NSURLSessionTaskDelegate` для поддержки проверки подлинности и получения таких событий, как перенаправление и выполнение задачи.

## Примечание

API-интерфейс `NSURLSession` включает в себя множество различных классов, которые работают вместе довольно сложным образом, что может быть неочевидно, если вы читаете справочную документацию отдельно. Ваше приложение создает один или несколько экземпляров `NSURLSession`, каждый из которых координирует группу связанных задач передачи данных. Например, если вы создаете приложение, то оно может создать один сеанс для каждой экран или один сеанс для интерактивного использования, а другой — для фоновых загрузок. В рамках каждого сеанса ваше приложение добавляет ряд задач, каждая из которых представляет собой запрос определенного URL-адреса (при необходимости после перенаправления HTTP).

## Типы URL-сессий

Задачи в данном сеансе URL имеют общий объект конфигурации сеанса, который определяет поведение соединения, например, максимальное количество одновременных подключений к одному хосту, могут ли соединения использовать сотовую сеть и т. д.

`NSURLSession` имеет одноэлементный сеанс `sharedSession` (у которого нет объекта конфигурации) для базовых запросов. Его не так легко настроить, как сеансы, которые вы создаете, но он служит хорошей отправной точкой, если у вас очень ограниченные требования. Вы получаете доступ к этому сеансу, вызывая метод общего класса. Для других видов сеансов вы создаете `NSURLSession` с одним из трех типов конфигураций:


1. *Сеанс по умолчанию* ведет себя так же, как общий сеанс, но позволяет настроить его. Вы также можете назначить делегата для сеанса по умолчанию для постепенного получения данных.

2. *Эфемерные сеансы* аналогичны общим сеансам, но не записывают кэши, файлы cookie или учетные данные на диск.
3. *Фоновые сеансы* позволяют выполнять загрузку и загрузку контента в фоновом режиме, когда ваше приложение не запущено.

## Типы задач URL-сессии

В рамках сеанса вы создаете задачи, которые при необходимости загружают данные на сервер, а затем извлекают данные с сервера либо в виде файла на диске, либо в виде одного или нескольких объектов NSData в памяти. API NSURLSession предоставляет четыре типа задач:

1. Задачи данных отправляют и получают данные с помощью объектов NSData.
2. Задачи данных предназначены для коротких, часто интерактивных запросов к серверу.
3. Задачи загрузки аналогичны задачам данных, но они также отправляют данные (часто в виде файла) и поддерживают фоновую загрузку, когда приложение не запущено.
4. Задачи загрузки извлекают данные в виде файла и поддерживают фоновую загрузку и выгрузку, когда приложение не запущено. Задачи WebSocket обмениваются сообщениями через TCP и TLS, используя протокол WebSocket, определенный в RFC 6455.

 **Важно!** Объект сеанса сохраняет строгую ссылку на делегата до тех пор, пока ваше приложение не завершит работу или явно не аннулирует сеанс. Если вы не аннулируете сеанс, ваше приложение будет терять память до тех пор, пока приложение не завершится.

Каждая задача, которую вы создаете с помощью сеанса, обращается к делегату сеанса, используя методы, определенные в NSURLSessionTaskDelegate. Вы также можете перехватить эти обратные вызовы до того, как они достигнут делегата сеанса, заполнив отдельный делегат, специфичный для задачи.

## Асинхронность и URL-сессии

Как и большинство сетевых API, NSURLSession API сильно асинхронен. Он возвращает данные в ваше приложение одним из трех способов, в зависимости от вызываемых вами методов: Если вы используете Swift, вы можете использовать методы, отмеченные ключевым словом `async`, для выполнения общих задач.

Например, `data(from:delegate:)` извлекает данные, а `download(from:delegate:)` загружает файлы. Ваша точка вызова использует ключевое слово `await` для приостановки работы до



завершения передачи. Вы также можете использовать метод `bytes(from:delegate:)` для получения данных в виде `AsyncSequence`. При таком подходе вы используете синтаксис `for-await-in` для перебора данных по мере их получения вашим приложением. Тип `URL` также предлагает удобные методы для извлечения байтов или строк из общего сеанса `URL`.

В `Swift` или `Objective-C` вы можете предоставить блок обработчика завершения, который запускается после завершения передачи. В `Swift` или `Objective-C` вы можете получать обратные вызовы к методу делегата по мере выполнения передачи и сразу после ее завершения. Помимо предоставления этой информации делегатам, `NSURLSession` предоставляет свойства состояния и хода выполнения. Запрашивайте эти свойства, если вам нужно принимать программные решения на основе текущего состояния задачи (с оговоркой, что ее состояние может измениться в любое время).

## Введение Сетевые функции Objective – C

Стоит подключить приложение `iOS` к Интернету — и оно становится гораздо интереснее. Например, представьте себе приложение, которое предлагает пользователям великолепные фоновые картинки для Рабочего стола. Пользователь может выбрать вариант из большого списка изображений и присвоить любой из этих рисунков в качестве фонового операционной системе `iOS`. А теперь вообразим себе приложение, которое делает то же самое, но обновляет ассортимент имеющихся изображений каждый день, неделю или месяц. Пользователь после какого-то перерыва возвращается к работе с программой и — о! Масса новых фоновых изображений динамически загружается в приложение. В этом и есть изюминка работы с веб-службами и Интернетом. Реализовать такие функции не составляет труда, если обладать базовыми знаниями о работе в Сети, применении `JSON`, `XML`. Ну, еще от разработчика приложения требуется известная креативность. Повторюсь, `iOS SDK` позволяет подключаться к Интернету, получать и отсылать данные. Это делается с помощью класса `NSURLConnection`. Сериализация и десериализация `JSON` выполняется в классе `NSJSONSerialization`. Синтаксический разбор `XML` производится с помощью `NSXMLParser`.

Повторюсь, в `SDK iOS 7` появились новые классы, работать с которыми мы научимся в этой главе. В частности, поговорим о классе `NSURLSession`, который управляет соединяемостью веб – сервисов и решает эту задачу более основательно, чем класс `NSURLConnection`. О соединяемости мы также поговорим далее в этой главе.

# Асинхронная загрузка с применением NSURLConnection Сетевые функции, JSON, XML

Необходимо асинхронно загрузить файл с имеющегося URL.

Используйте класс `NSURLConnection` с асинхронным запросом. Класс `NSURLConnection` можно использовать двумя способами — асинхронным и синхронным. При асинхронном соединении создается новый поток, и процесс загрузки выполняется в этом новом потоке. Синхронное соединение блокирует вызывающий поток, а содержимое загружается прямо в ходе обмена данными.

Многие разработчики полагают, что при синхронном соединении блокируется главный поток, но это неверно. Синхронное соединение всегда блокирует тот поток, в котором оно было инициировано. Если вы запускаете синхронное соединение из главного потока — да, главный поток будет заблокирован. Но синхронное соединение, запущенное из другого потока, будет напоминать асинхронное именно в том отношении, что оно никак не повлияет на главный поток. На самом деле единственное различие между синхронным и асинхронным соединениями заключается в том, что для асинхронного соединения среда времени исполнения создает отдельный поток, а для синхронного — нет.

Чтобы создать асинхронное соединение, необходимо следующее.

1. Иметь URL или экземпляр `NSString`.
2. Преобразовать строку в экземпляр `NSURL`.
3. Поместить URL в URL-запросе типа `NSURLRequest`, а если мы имеем дело с изменяемыми URL — в экземпляр `NSMutableURLRequest`.
4. Создать экземпляр `NSURLConnection` и передать ему URL-запрос.

Можно создать асинхронное соединение по URL с помощью метода класса `sendAsynchronousRequest:queue:completionHandler:`, относящегося к классу `NSURLConnection`. Этот метод имеет следующие параметры:

1. `sendAsynchronousRequest` — запрос типа `NSURLRequest`, рассмотренный ранее;
2. `queue` — операционная очередь. При желании можно просто выделить и инициализировать новую операционную очередь и передать ее этому методу;
3. `completionHandler` — блоковый объект, выполняемый, когда асинхронное соединение завершает работу, успешно или неуспешно. Этот блоковый объект должен принимать три параметра:
  - Объект типа `NSURLResponse`, в котором заключается ответ, полученный нами от сервера, — при наличии такого ответа;
  - Данные типа `NSData` при их наличии. Это будут данные, собранные в ходе соединения по указанному URL;

- Ошибка типа NSError в случае ее возникновения.

\*Метод `sendAsynchronousRequest:queue:completionHandler:` не вызывается в главном потоке. Поэтому, если вам потребуется решить задачу, связанную с пользовательским интерфейсом, убедитесь, что вернулись к главному потоку.

Итак, довольно теории, перейдем к примерам. В данном примере попытаемся собрать HTML-контент с домашней страницы Apple, а потом выведем эту информацию в строковом формате в окне консоли:

```
NSString *urlAsString = @"http://www.apple.com";
NSURL *url = [NSURL URLWithString:urlAsString];
NSURLRequest *urlRequest = [NSURLRequest requestWithURL:url];
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection
    sendAsynchronousRequest:urlRequest
    queue:queue
    completionHandler:^(NSURLResponse *response,
                        NSData *data,
                        NSError *error) {
    if ([data length] >0 &&
        error == nil){
        NSString *html = [[NSString alloc] initWithData:data
        encoding:NSUTF8StringEncoding];
        NSLog(@"HTML = %@", html);
    }
    else if ([data length] == 0 &&
        error == nil){
        NSLog(@"Nothing was downloaded.");
    }
    else if (error != nil){
        NSLog(@"Error happened = %@", error);
    }
} ]];
```

Да, все так просто. Если вы хотите сохранить данные, которые мы загрузили на диск в ходе соединения, это можно сделать с помощью подходящих методов класса `NSData`, получаемых от завершающего блока:

```
NSString *urlAsString = @"http://www.apple.com";
NSURL *url = [NSURL URLWithString:urlAsString];
NSURLRequest *urlRequest = [NSURLRequest requestWithURL:url];
```

```

NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection
sendAsynchronousRequest:urlRequest queue:queue
completionHandler:^(NSURLResponse *response,
                    NSData *data,
                    NSError *error) {
    if ([data length] >0 &&
        error == nil){
/* Прикрепляем имя файла к каталогу с документами. */
        NSURL *filePath =
        [[self documentsFolderUrl]
        URLByAppendingPathComponent:@"apple.html"];
        [data writeToURL:filePath atomically:YES];
        NSLog(@"Successfully saved the file to %@", filePath);
    }
    else if ([data length] == 0 &&
             error == nil){
        NSLog(@"Nothing was downloaded.");
    }
    else if (error != nil){
        NSLog(@"Error happened = %@", error);
    }
}];

```

Все действительно просто. В более ранних версиях iOS SDK соединения по URL происходили с применением делегирования, но теперь модель стала обычной блоковой и вам не придется заниматься реализацией делегатов.

## Обработка задержек при асинхронных соединениях

Необходимо задать лимит ожидания — проще говоря, задержку — при асинхронном соединении. Задать задержку в URL-запросе, посылаемом классу `NSURLConnection`.

При инстанцировании объекта типа `NSURLRequest` для передачи URL-соединения можно воспользоваться методом класса `requestWithURL:cachePolicy:timeoutInterval:`, относящимся к этому объекту, и передать желаемую длительность задержки в секундах в параметре `timeoutInterval`.

Например, если вы готовы не более 30 секунд дожидаться, пока загрузится содержимое главной страницы Apple (с применением синхронного соединения), создайте ваш URL таким образом:

```
(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *urlAsString = @"http://www.apple.com";
    NSURL *url = [NSURL URLWithString:urlAsString];
    NSURLRequest *urlRequest =
    [NSURLRequest
        requestWithURL:url
        cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
        timeoutInterval:30.0f];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [NSURLConnection
        sendAsynchronousRequest:urlRequest
        queue:queue
        completionHandler:^(NSURLResponse *response,
                                NSData *data,
                                NSError *error) {
        if ([data length] >0 &&
            error == nil){
            NSString *html = [[NSString alloc] initWithData:data
                encoding:NSUTF8StringEncoding];
            NSLog(@"HTML = %@", html);
        } else if ([data length] == 0
            && error ==
            nil){
            NSLog(@"Nothing was downloaded.");
        }
        else if (error != nil){
            NSLog(@"Error happened = %@", error);
        }
    }];
    self.window = [[UIWindow alloc]
        initWithFrame:[UIScreen mainScreen] bounds];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES; }
```

Что же здесь происходит? Дело в том, что среда времени исполнения пытается получить содержимое, расположенное по предоставленной ссылке. Если это удастся сделать в течение заданных 30 секунд и соединение устанавливается до возникновения задержки — хорошо. В противном случае среда времени исполнения выдаст вам ошибку задержки (error) в соответствующем параметре завершающего блока.

## Синхронная загрузка с применением

Необходимо синхронно загрузить информацию, расположенную по имеющемуся URL. Используйте метод класса `sendSynchronousRequest:returningResponse:error:`, относящийся к классу `NSURLConnection`. Возвращаемое значение этого метода — данные типа `NSData`.

Пользуясь методом класса `sendSynchronousRequest:returningResponse:error:`, относящимся к классу `NSURLConnection`, можно посылать синхронный запрос к URL. А теперь внимание! Повторюсь, синхронные соединения необязательно блокируют главный поток. Эти соединения блокируют актуальный поток, то есть выполняющий текущую задачу, и если этот поток не главный, то главный поток останется свободным. Если приступить к обработке глобальной параллельной очереди в GCD, а потом инициировать синхронное соединение, то вы не заблокируете главный поток.

Попробуем инициировать наше первое синхронное соединение и посмотрим, что произойдет. В данном примере мы попытаемся получить домашнюю страницу сайта Yahoo!:

```
(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSLog(@"We are here...");
    NSString *urlAsString = @"http://www.yahoo.com";
    NSURL *url = [NSURL URLWithString:urlAsString];
    NSURLRequest *urlRequest = [NSURLRequest
requestWithURL:url];
    NSURLResponse *response = nil;
    NSError *error = nil;
    NSLog(@"Firing synchronous url connection...");
    NSData *data = [NSURLConnection
sendSynchronousRequest:urlRequest returningResponse:&response
error:&error]; if ([data
length] > 0 &&
    error == nil){
```

```

    NSLog(@"%lu bytes of data was returned.", (unsigned long)[data
length]); } else if ([data length] == 0 &&
    error == nil){
    NSLog(@"No data was returned.");
}
else if (error != nil){
    NSLog(@"Error happened = %@", error);
}
NSLog(@"We are done."); self.window = [[UIWindow
alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible]; return
YES; }

```

Если запустить это приложение, а потом взглянуть в окно консоли, то там окажется выведен следующий результат:

We are here...

Firing synchronous url connection... 252117 bytes of data was returned. We are done.

Итак, вполне очевидно, что актуальный поток написал на консоли строку We are here..., дождался окончания соединения (поскольку это синхронное соединение, блокирующее актуальный поток), а потом вывел в окне консоли текст We are done. Теперь проведем эксперимент. Поместим то же самое синхронное соединение в глобальной параллельной очереди в GCD, то есть гарантированно обеспечим параллелизм, и посмотрим, что произойдет:

```

(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSLog(@"We are here...");
    NSString *urlAsString = @"http://www.yahoo.com";
    NSLog(@"Firing synchronous url connection...");
    dispatch_queue_t dispatchQueue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0);
    dispatch_async(dispatchQueue, ^(void) {
        NSURL *url = [NSURL URLWithString:urlAsString];
        NSURLRequest *urlRequest = [NSURLRequest requestWithURL:url];

```

```

        NSURLResponse *response = nil;
        NSError *error = nil;
        NSData *data = [NSURLConnection
sendSynchronousRequest:urlRequest

returningResponse:&response

error:&error];
        if ([data length] > 0 &&
            error == nil){
            NSLog(@"%lu bytes of data was returned.", (unsigned
long)[data length]);
        }
        else if ([data length] == 0 &&
            error == nil){
            NSLog(@"No data was returned.");
        }
        else if (error != nil){
            NSLog(@"Error happened = %@", error);
        }
    });
    NSLog(@"We are done.");
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

return YES; }

```

Вывод будет примерно таким:

We are here...

Firing synchronous url connection... We are done. 252450 bytes  
of data was returned.

Итак, в данном примере текущий поток вывел текст We are done в окне консоли, не дожидаясь, пока синхронное соединение завершит считывание с заданного URL. Интересно, правда? Таким образом, этот пример доказывает, что при умелом обращении синхронное URL-соединение не обязательно блокирует главный поток. Тем не менее оно гарантированно блокирует текущий поток.



## Изменение URL-запроса с применением NSMutableURLRequest

Требуется корректировать различные HTTP-заголовки и настройки URL-запроса перед передачей его URL-соединению. Эта техника лежит в основе некоторых разделов, рассмотренных далее в этой главе. Пользуйтесь NSMutableURLRequest вместо NSURLRequest.

URL-запрос может быть изменяемым или неизменяемым. URL-запросы, относящиеся к первой категории, поддаются изменениям после выделения и инициализации, а те, что относятся ко второй категории, — нет. Этот раздел посвящен изменяемым URL-запросам. Их можно создавать с помощью класса NSMutableURLRequest.

Рассмотрим пример, в котором длительность задержки при URL-запросе изменяется после выделения и инициализации этого запроса:

```
NSString *urlAsString = @"http://www.apple.com";
NSURL *url = [NSURL URLWithString:urlAsString];
NSMutableURLRequest *urlRequest = [NSMutableURLRequest
requestWithURL:url];
[urlRequest setTimeoutInterval:30.0f];
```

Теперь обратимся к другому примеру, где URL и время задержки при URL-запросе задаются после выделения и инициализации:

```
NSString *urlAsString = @"http://www.apple.com";
NSURL *url = [NSURL URLWithString:urlAsString];
NSMutableURLRequest *urlRequest = [NSMutableURLRequest new];
[urlRequest setTimeoutInterval:30.0f];
[urlRequest setURL:url];
```

В других разделах этой главы мы изучим некоторые очень тонкие приемы, которые осуществимы с помощью изменяемых URL-запросов.

## Отправка запросов HTTP GET с применением NSURLConnection

Необходимо отправить запрос GET по протоколу HTTP и, возможно, передать получателю вместе с этим запросом какие-либо параметры.

По определению GET-запросы допускают указание параметров в строках запросов в общеизвестной форме: `http://example.com/?param1=value1&param2=value2...`

Строки можно использовать для перечисления параметров в обычном формате.

GET-запрос — это запрос к веб-серверу на получение данных. Обычно запрос сопровождается параметрами, которые отправляются в строке запроса как часть URL.

Чтобы протестировать вызов GET, необходимо найти веб-сервер, принимающий такие вызовы и способный отослать какие-либо данные в ответ. Это просто. Как вы уже знаете, при открытии веб-страницы в браузере этот браузер по умолчанию посылает запрос GET к конечной точке. Поэтому данный раздел вы можете опробовать на любом сайте по своему усмотрению.

Для симулирования отправки параметров строки запроса в GET-запросе к той же веб-службе с помощью `NSURLConnection` воспользуемся изменяемым URL-запросом и явно укажем ваш HTTP-метод для GET с помощью метода `setHTTPMethod:`, относящегося к `NSMutableURLRequest`. Параметры оформляются как часть URL, следующим образом:

```
(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *urlAsString = <# Здесь укажите URL веб-сервера #>;
    urlAsString = [urlAsString
stringByAppendingString:@"?param1=First"]; urlAsString
    = [urlAsString
stringByAppendingString:@"&param2=Second"];
    NSURL *url = [NSURL URLWithString:urlAsString];
    NSMutableURLRequest *urlRequest = [NSMutableURLRequest
                                     requestWithURL:url];

    [urlRequest setTimeoutInterval:30.0f];
    [urlRequest setHTTPMethod:@"GET"];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [NSURLConnection
     sendAsynchronousRequest:urlRequest queue:queue
     completionHandler:^(NSURLResponse *response,
                           NSData *data,
                           NSError *error) {
        if ([data length] >0 &&
            error == nil){
            NSString *html = [[NSString alloc] initWithData:data
            NSLog(@"HTML = %@", html);
        } else if ([data length] == 0
&&
            error == nil){
            NSLog(@"Nothing was downloaded.");
        }
        else if (error != nil){
            NSLog(@"Error happened = %@", error);
        }
    }];
}
```

```

}

    }]; self.window = [[UIWindow alloc]
initWithFrame:
    [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}

```

Единственный момент, который необходимо учитывать, заключается в том, что перед первым параметром ставится вопросительный знак, а перед всеми последующими — амперсанд (&). Вот и все! Теперь вы можете пользоваться методом HTTP GET и отправлять параметры в строке запроса.

## Отправка запросов HTTP POST с применением `NSURLConnection`

Необходимо вызвать метод HTTP POST веб-сервера и, возможно, передать параметры (в теле HTTP или в строке запроса) определенной веб-службе. Как и в случае с

методом GET, можно использовать метод POST с применением `NSURLConnection`. Следует явно задать метод нашего URL как POST.

Напишем простое приложение, которое может создать асинхронное соединение и отослать ряд параметров в виде строки запроса и нескольких параметров в теле HTTP-запроса по URL:

```

(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *urlAsString = <# Здесь укажите URL веб-сервера #>;
    urlAsString = [urlAsString
stringByAppendingString:@"?param1=First"];
    urlAsString = [urlAsString
stringByAppendingString:@"&param2=Second"];
    ;
    NSURL *url = [NSURL URLWithString:urlAsString];
    NSMutableURLRequest *urlRequest = [NSMutableURLRequest
requestWithURL:url];
    [urlRequest setTimeoutInterval:30.0f];
    [urlRequest setHTTPMethod:@"POST"];
    NSString *body = @"bodyParam1=BodyValue1&bodyParam2=BodyValue2";
    [urlRequest setHTTPBody:[body

```

```

dataUsingEncoding:NSUTF8StringEncoding]];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [NSURLConnection
    sendAsynchronousRequest:urlRequest queue:queue
    completionHandler:^(NSURLResponse *response,
                          NSData *data,
                          NSError *error) {
        if ([data length] >0 &&
            error == nil){
            NSString *html = [[NSString alloc] initWithData:data
            encoding:NSUTF8StringEncoding];
            NSLog(@"HTML = %@", html);
        }
        else if ([data length] == 0 &&
            error == nil){
            NSLog(@"Nothing was downloaded.");
        }
        else if (error != nil){
            NSLog(@"Error happened = %@", error);
        }
    }];
    self.window = [[UIWindow alloc]
    initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible]; return
    YES;
}

```

Первый параметр, пересылаемый в теле HTTP, не обязательно предварять вопросительным знаком, а пересылаемый в строке запроса — обязательно.

## Отправка запросов HTTP DELETE с применением NSURLConnection

Требуется вызвать веб-службу методом HTTP DELETE, чтобы удалить ресурс, расположенный по ссылке URL, и, возможно, передать веб-службе определенные параметры, которые будут находиться в теле HTTP или в строке запроса. Как и методы GET и POST, метод DELETE можно использовать с помощью

NSURLConnection. Необходимо явно задать метод вашего URL как DELETE.

Напишем простое приложение, которое будет создавать асинхронное соединение и отправлять несколько параметров в строке запроса, а несколько — в теле HTTP. Отправка будет происходить по указанному URL с помощью метода DELETE HTTP:

```
(BOOL) application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *urlAsString = <# Здесь укажите URL веб-сервера #>;

    urlAsString = [urlAsString
stringByAppendingString:@"?param1=First"];
    urlAsString = [urlAsString
stringByAppendingString:@"&param2=Second"];

    NSURL *url = [NSURL URLWithString:urlAsString];
    NSMutableURLRequest *urlRequest = [NSMutableURLRequest
requestWithURL:url];
    [urlRequest setTimeoutInterval:30.0f];
    [urlRequest setHTTPMethod:@"DELETE"];

    NSString *body =
@"bodyParam1=BodyValue1&bodyParam2=BodyValue2";
    [urlRequest setHTTPBody:[body
dataUsingEncoding:NSUTF8StringEncoding]];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [NSURLConnection
sendAsynchronousRequest:urlRequest queue:queue
completionHandler:^(NSURLResponse *response,
                    NSData *data,
                    NSError *error) {
        if ([data length] >0 &&
            error == nil){
            NSString *html = [[NSString alloc] initWithData:data
            NSLog(@"HTML = %@", html);
        }
        else if ([data length] == 0 &&
            error == nil){
            NSLog(@"Nothing was downloaded.");
        }
        else if (error != nil){
            NSLog(@"Error happened = %@", error);
        }
    }
}
```

```

    }]; self.window = [[UIWindow alloc]
initWithFrame:
    [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}

```

Этот пример очень напоминает код, рассмотренный. Разница заключается в том, что здесь мы использовали HTTP-метод DELETE. Прочая информация практически идентична той, что была изложена в упомянутых разделах.

## Отправка запросов HTTP PUT с применением `NSURLConnection`

Требуется вызывать веб-службу методом HTTP PUT, чтобы размещать ресурс на веб-сервере и, возможно, передать веб-службе определенные параметры, которые будут находиться в теле HTTP или в строке запроса. Как и методы GET, POST и DELETE, метод PUT можно использовать с помощью `NSURLConnection`. Необходимо явно задать метод вашего URL как PUT.

Напишем простое приложение, которое будет создавать асинхронное соединение и отправлять несколько параметров в строке запроса, а несколько — в теле HTTP. Отправка будет происходить по указанному URL с помощью метода PUT:

```

(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *urlAsString = <# Здесь укажите URL веб-сервера #>;
    urlAsString = [urlAsString
stringByAppendingString:@"?param1=First"];
    urlAsString = [urlAsString
stringByAppendingString:@"&param2=Second"];
    NSURL *url = [NSURL URLWithString:urlAsString];
    NSMutableURLRequest *urlRequest = [NSMutableURLRequest
requestWithURL:url];
    [urlRequest setTimeoutInterval:30.0f];
}

```

```

[URLRequest setHTTPMethod:@"PUT"];
NSString *body = @"bodyParam1=BodyValue1&bodyParam2=BodyValue2";
[URLRequest setHTTPBody:[body
dataUsingEncoding:NSUTF8StringEncoding]];
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection
sendAsynchronousRequest:urlRequest queue:queue
completionHandler:^(NSURLResponse *response,
                    NSData *data,
                    NSError *error) {
    if ([data length] >0 &&
        error == nil){
        NSString *html = [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];
        NSLog(@"HTML = %@", html);
    }
    else if ([data length] == 0 &&
             error == nil){
        NSLog(@"Nothing was downloaded.");
    }
    else if (error != nil){
        NSLog(@"Error happened = %@", error);
    }
}]; self.window = [[UIWindow alloc]
initWithFrame:
                [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible]; return
YES;
}

```

## Сериализация массивов и словарей в JSON

### Постановка задачи

Необходимо сериализовать словарь или массив в объект JSON, который можно передавать по сети или просто сохранять на диск. Воспользуйтесь методом `dataWithJSONObject:options:error:` класса `NSJSONSerialization`.

Метод `dataWithJSONObject:options:error:` класса `NSJSONSerialization` может сериализовывать словари и массивы, в которых содержатся лишь экземпляры переменных `NSString`,

NSNumber, NSArray, NSDictionary либо NSNull для нулевых значений. Как было указано ранее, объект, передаваемый этому методу, должен быть либо массивом, либо словарем.

Теперь создадим простой массив с несколькими ключами и значениями:

```
NSDictionary *dictionary = @{
    @"First Name" : @"Anthony",
    @"Last Name" : @"Robbins",
    @"Age" : @51,
    @"children" : @[
        @"Anthony's Son 1",
        @"Anthony's Daughter 1",
        @"Anthony's Son 2",
        @"Anthony's Son 3",
        @"Anthony's Daughter 2"
    ],
};
```

Как видите, в этом словаре содержатся имя, фамилия и возраст Энтони Роббинса. Ключ словаря, называемый children, содержит имена детей Энтони. Это массив строк, где каждой строкой представлен один ребенок. Итак, на данный момент переменная dictionary содержит все значения, которые мы хотели в нее поместить. Теперь нужно сериализовать ее в объект JSON:

```
NSError *error = nil;
NSData *jsonData = [NSJSONSerialization
    dataWithJSONObject:dictionary
    options:NSJSONWritingPrettyPrinted
    error:&error];
if ([jsonData length] > 0 && error
    == nil){
    NSLog(@"Successfully serialized the dictionary into data = %@", jsonData);
} else if ([jsonData length] == 0
    && error == nil){
    NSLog(@"No data was returned after serialization.");
}
else if (error != nil){
    NSLog(@"An error happened = %@", error); }
```

Возвращаемым значением метода dataWithJSONObject:options:error: являются данные типа NSData. Правда, эти данные можно просто преобразовать в строку и вывести на консоль. Для этого применяется метод-инициализатор initWithData:encoding: класса NSString. Далее приведен полный пример, в котором словарь преобразуется в объект JSON. Этот объект превращается в строку, а строка выводится в окне консоли:

```
(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSDictionary *dictionary = @{
```



```

        @"First Name" : @"Anthony",
        @"Last Name" : @"Robbins",
        @"Age" : @51,
        @"children" : @[
            @"Anthony's Son 1",
            @"Anthony's Daughter 1",
            @"Anthony's Son 2",
            @"Anthony's Son 3",
            @"Anthony's Daughter 2"
        ],
    };

    NSError *error = nil;
    NSData *jsonData = [NSJSONSerialization
        dataWithJSONObject:dictionary
        options:NSJSONWritingPrettyPrinted
        error:&error];

    if ([jsonData length] > 0 &&
        error == nil){
        NSLog(@"Successfully serialized the dictionary into data.");
        NSString *jsonString = [[NSString alloc] initWithData:jsonData
            encoding:NSUTF8StringEncoding];
        NSLog(@"JSON String = %@", jsonString);
    }
    else if ([jsonData length] == 0 &&
        error == nil){
        NSLog(@"No data was returned after serialization.");
    }
    else if (error != nil){
        NSLog(@"An error happened = %@", error);
        self.window = [[UIWindow alloc]
            initWithFrame:[UIScreen mainScreen] bounds]];
        // Точка переопределения для дополнительной настройки после
        запуска приложения
        self.window.backgroundColor = [UIColor whiteColor];
        [self.window makeKeyAndVisible];
        return YES;
    }

    Successfully serialized the dictionary into data.
    JSON String = {
        "Last Name" : "Robbins",

```

```
"First Name" : "Anthony",
"children" : [
    "Anthony's Son 1",
    "Anthony's Daughter 1",
    "Anthony's Son 2",
    "Anthony's Son 3",
    "Anthony's Daughter 2"
],
```

```
"Age" : 51 }
```

## Десериализация нотации JSON в массивы и словари

Имеются данные в формате JSON, их необходимо десериализовать в словарь или массив. Воспользуйтесь методом `JSONObjectWithData:options:error:` класса `NSJSONSerialization`.

Если вы уже сериализовали ваш словарь или массив в объект JSON (заклученный в экземпляре `NSData`), то эти данные нужно будет десериализовать обратно в словарь или массив. Это делается с помощью метода `JSONObjectWithData:options:error:`, относящегося к классу `NSJSONSerialization`. Объект, возвращаемый этим методом, будет представлять собой либо словарь, либо массив в зависимости от того, какие данные ему были переданы.

Рассмотрим пример:

```
/* Сейчас попытаемся сериализовать объект JSON в словарь. */
error = nil;
id jsonObject = [NSJSONSerialization
                 JSONObjectWithData:jsonData
                 options:NSJSONReadingAllowFragments
                 error:&error];
if (jsonObject != nil &&
    error == nil){
    NSLog(@"Successfully deserialized...");
    if ([jsonObject isKindOfClass:[NSDictionary class]]){
        NSDictionary *deserializedDictionary = (NSDictionary
*)jsonObject;
        NSLog(@"Deserialized JSON Dictionary = %@",
deserializedDictionary);
    }
    else if ([jsonObject isKindOfClass:[NSArray class]]){
        NSArray *deserializedArray = (NSArray *)jsonObject;
        NSLog(@"Deserialized JSON Array = %@", deserializedArray);
```

```

}

else {
    /* Был возвращен какой-то другой объект. Мы не знаем,
       что делать в этой ситуации, так как десериализатор
       возвращает только словари или массивы. */
}
}

else if (error != nil){
    NSLog(@"An error happened while deserializing the JSON
data.");
}
}

```

Если теперь объединить этот код с кодом из предыдущего пункта, то можно будет сначала сериализовать словарь в объект JSON, десериализовать объект JSON обратно в словарь, а потом вывести результаты на консоль, чтобы убедиться, что все работает нормально:

```

(BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSDictionary *dictionary = @{
        @"First Name" : @"Anthony",
        @"Last Name" : @"Robbins",
        @"Age" : @51,
        @"Children" : @[
            @"Anthony's Son 1",
            @"Anthony's Daughter 1",
            @"Anthony's Son 2",
            @"Anthony's Son 3",
            @"Anthony's Daughter 2",
        ],
    };

    NSError *error = nil;
    NSData *jsonData = [NSJSONSerialization
                        dataWithJSONObject:dictionary
                        options:NSJSONWritingPrettyPrinted
                        error:&error];
    if ([jsonData length] > 0 &&
        error == nil){
        NSLog(@"Successfully serialized the dictionary into data.");
        /* Сейчас попытаемся сериализовать объект JSON в словарь. */
        error = nil; id jsonObject = [NSJSONSerialization

```

```

JSONObjectWithData:jsonData
options:NSJSONReadingAllowFragments error:&error];
if (jsonObject != nil &&
    error == nil){
    NSLog(@"Successfully deserialized...");
    if ([jsonObject isKindOfClass:[NSDictionary class]]){
        NSDictionary *deserializedDictionary = (NSDictionary
*)jsonObject;
        NSLog(@"Deserialized JSON Dictionary = %@",
deserializedDictionary);
    }
    else if ([jsonObject isKindOfClass:[NSArray class]]){
        NSArray *deserializedArray = (NSArray *)jsonObject;
        NSLog(@"Deserialized JSON Array = %@", deserializedArray);
    }
    else {
        /* Был возвращен какой-то другой объект. Мы не знаем,
что делать
        в этой ситуации, так как десериализатор возвращает
только словари
        или массивы. */
    }
    }
    else if (error != nil){
        NSLog(@"An error happened while deserializing the JSON
data.");
    }
}
else if ([jsonData length] == 0 &&
    error == nil){
    NSLog(@"No data was returned after serialization.");
}
else if (error != nil){
    NSLog(@"An error happened = %@", error);
} self.window = [[UIWindow alloc]
initWithFrame:
    [[UIScreen mainScreen] bounds]];

```

```
        // Точка переопределения для дополнительной настройки после
запуска приложения
        self.window.backgroundColor = [UIColor whiteColor];
        [self.window makeKeyAndVisible];
        return YES;
    }
```