**4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## **4.1** Проектирование программ сверху вниз

Самое главное условие успешного создания как крупного, так и небольшого приложения заключается в применении надежных методов проектирования.

Самое широкое распространение при написании такого рода программ получили следующие методы:

* нисходящий (сверху-вниз);
* восходящий (снизу-вверх);
* специальный (на данный конкретный случай).

В случае нисходящего метода процесс разработки программы начинается с программы высокого уровня и спускается до программ низкого уровня.

В свою очередь восходящий метод работает в обратном направлении, то есть сначала идет разработка отдельных специальных подпрограмм, на основании которых в последствии строятся более сложные, которые заканчиваются самым верхним уровнем приложения.

Специальный подход не имеет какого-то заранее установленного способа разработки, то есть решения в выборе метода происходят во время процесса разработки и могут меняться в любой момент времени.

JavaScript является, в одно время, как структурированным так и не структурированным языком. Именно поэтому в качестве разработки был выбран язык TypeScript. Он представляет собой строгий и структурированный JavaScript и лучше всего подходит для нисходящего программирования. Нисходящий подход позволяет писать ясный, легко читаемый код, что в дальнейшем не вызовет проблем с сопровождением приложения. К тому же данный метод отлично подходит для прояснения и создания всей структуры программы в целом до кодирования более низких уровней приложения.

Выбранный метод позволяет уменьшить потери времени, обусловленные неудачными и ошибочными решениями на начальном этапе. А в условиях небольшого срока разработки программы – это делает его основным методом разработки и позволяет перейти к проектированию программы.

## **4.2** Структурирование программы

Как описывалось выше, для любой общей схемы при применении метода «сверху-вниз» разработку начинают с общего описания приложения, а после этого двигаются в направлении проработки ее конкретных деталей, компонентов и прочего.

При разработке любой программы лучше всего точно определить, что и как программа будет делать на самом высоком уровне, а затем уже погружаться в детали, касающиеся каждого ее действия. На рисунке 4.1 представлена схема нисходящего метода разработки.



Рисунок 4.1 – Схема нисходящего метода разработки программы

Исходя из изображения выше можно заметить, что эта схема очень легко ложиться на структурную схему разрабатываемого приложения, что говорит о правильном выборе метода разработки. Согласно схеме, можно легко определить и выделить два основных модуля программы. Модулей два потому как приложение содержит клиентскую и серверную части, каждая из которых служит обособленным модулем высокого уровня. К модулям высокого уровня относятся:

* блок ядра клиентского приложения;
* блок ядра серверного приложения.

Теперь можно приступать к составлению перечня действий, которые будет выполнять каждый отдельный модуль.

## **4.3** Блок ядра серверного приложения

Разработка начинается с ядра серверного приложения потому как разрабатывать клиентское ядро без минимально функционирующей серверной части не имеет смысла.

Первая и самая основная цель серверной стороны – это запускать сервер. Так как сервер будет работать на платформе NodeJS, то самый удобный и безопасный способ работать с сервером – это подключить библиотеку ExpressJS. Это уже давно зарекомендовавший себя модуль, который имеет большой и гибкий набор конфигураций.

Необходимо добавить данную библиотеку в проект с помощью NPM-менеджера. Для этого нужно использовать команду: «npm install express --save». После установки нужно подключить модуль к основному файлу сервера и произвести пробный запуск сервера на 1337 порту. По умолчанию, сервер будет запущен на локальном адресе компьютера без каких-либо дополнительных настроек. Пробный запуск сервера представлен на рисунке 4.2.



Рисунок 4.2 – Тестовый запуск сервера на ExpressJS модуле

После успешного тестового запуска сервера можно приступить к настройке его конфигураций.

Имеет смысл вынести конфигурацию и настройки сервера в отдельный файл. Это упростит работу с проектом и дальнейшее ориентирование в файлах проекта. Сервер должен быть достаточно безопасным, так как будет хранить данные бюджета пользователей. Здесь будет использоваться подход, при котором к серверу будет открыто как можно меньше путей. Именно поэтому в конфигурации сервера нужно закрыть доступ ко всем скрытым файлам, начинающимся с точки. Также в конфигурации слудет указать, чтобы по умолчанию открывались только файлы с расширением html и не разрешалась никакая переадресация.

Сервер будет содержать и статический данные, такие как скрипты, файлы стилей и скриптов. Имеет смысл перенести статичесикие файлы в одну папку и запусть сервер для работы именно с ней в качестве корневого каталога. Таким образом доступ к серверу будет строго ограничен статическими файлами.

Для обмена данными между клиентом и серевером будет использоваться формат JSON – это текстовый формат обмена данными, основанный на JavaScript. По умолчанию данный модуль не включен в конфигурацию сервера, поэтому его необходимо добавить через NPM-менеджер, а затем добавить в конфигурацию.

Постоянно загружать статические файлы по запросу клиента не имеет смысла, именно поэтому в конфигурацию также включается настройка кеша. В настройке нудно указать максимыльный срок кеширования файлов, в данном случае это будет один день. А также необходимо включить использование eTag – часть HTTP. Это один из нескольких механизмов, с помощью которых HTTP обеспечивает веб-проверку кэша и который позволяет клиенту делать условный запрос. Это позволяет кэшу быть более эффективным и экономит пропускную способность, так как веб-серверу не нужно отправлять полный ответ, если содержимое не изменилось. ETag также может быть использован для оптимального управления многопоточностью как способ, чтобы помочь предотвратить одновременное обновление и перезапись ресурса.

Etag – это закрытый идентификатор, присвоенный веб-сервером на определенную версию ресурса, найденного на URL. Если содержание ресурса для этого адреса меняется на новое, назначается и новый ETag. Использование в таком ключе ETags аналогично использованию отпечатков пальцев, можно быстро сравнить и определить, являются ли две версии ресурса одинаковыми или нет. Сравнение ETag имеет смысл только c Etag с одного и того же URL, идентификаторы, полученные из разных URL-адресов, могут быть, а могут не быть равны, вне зависимости от ресурсов, так что их сравнение не имеет какого-либо смысла.

Также для кеширования будет использоваться свойство заголовка «Last modified». Данный заголовок также служит в качестве элемента кеширования данных. В нем содержится дата последнего изменени файла и если при запросе, клинетские браузер видит, что дата изменения не менялась, то подгружает файл из кеша браузера.

На данном этапе настройка конфигурации сервера заканчивается. Теперь сервер будет раздавать статический файлы, а значит можно заняться разарботкой статических файлов на клиентской стороне.

## **4.4** Блок ядра клиентского приложения

Основное ядро клиентской части приложения будет содержаться в JavaScript файле. Однако его нужно загрузить в HTML файл. HTML файл будет статически загружаться с сервера, автоматически загуражать статические стили и основной JavaScript файл, после чего начнеться посроение всех частей приложения из основного ядра.

В HTML файлe указываются пути для загрузки скриптов и стилей, также указываются мета-теги для оптимизации страницы браузера под мобильные устройства. Как описывалось ранее, в дальнейшем приложение будет обернуто в прилодение Cordova. Следовательно функции зумирования страницы должны быть отключены.

В теле страницы описываются основные элементы страницы, на основании которых будут строиться остальные части программы. К данным элементам относяться:

* блок заголовка;
* блок меню;
* блок основных элементов.

Блок заголовка постоянно виден на экране и должен содержать только элементы которые всегда должны быть на виду. В данный момент это иконка для открывания и закрывания блока меню.

Блок меню, как и блок заголовка, представляют из себя суб-модули. Все, что нужно блоку меню для работа – это список старниц и ссылок на них, а также элемент меню, который находится в статическом HTML файле. На основании полученных данных, строится список всех элементов меню, после чего, он добавляется к элементу меню. На этом работа модулю заканчивается. Даже несмотря на то, что модуль получается небольшим, его стоит отделять в качество независимого модулю на случай разростания сложности приложения.

Как можно понять из абзаца выше, приложеие будет иметь несколько страниц. Но каждый раз загружать новую страницу, как это делается на сайтах, не выгодно. Именно поэтому приложением будет представлять из себя одностраничное веб-приложение.

Одностраничное веб-приложение – это приложение, все функции которого отображаются без перехода на другие страницы, то есть загрузка нового контента происходит через запросы. Eсли контент погружается через запросы, то стоит сразу поговорить о преимуществе данного подхода:

* снижение нагрузки на сервер – запрашивается только нужная часть информации, то есть данные, а не вся страница;
* сервер передает именно данные (обычно через JSON), а не обрабатывает и отрисовывает всю страницу по-новому.

В следствии этого, при разработке сильно разделяется серверная и клиентская части. Сервер, при этом, обрабатывает лишь API запросы, а за дизайн и отображение информации отвечает клиенсткая сторона. Сразу видно, что при таком подходе имеет смысл разделить программистов на две команды – клиентскую и серверную части. Как видно, одностраничные приложения – это не только способ отображения информации, но и совершенно иной организацонный поход к разработке.

Область применения одностраничным веб-приложений давольно широка. Помимо снижения нагрузки на сервер и экономии траффика между клиентом и сервером одна из центровых причин для такого подхода к разработке была создание веб-приложений с максимально приближенным к нативному поведению. То есть если открыть тот же фотошоп – он не прогружается заново в тот момент, как открывается новый документ. Исходя из того, что данное приложение будет обернуто в мобильное приложение и должно быть похоже на нативное приложение, и была выбрана разработка одностраничного веб-приложения.

Для работы с адресами нужен отдельный модуль, потому как в адресе старнице будет указывать не только текущая страница, но необходимые параметры для ее правильной работы. Основная работа данного модуля – это уметь обрабатывать строку адреса в удобный для работы JSON формат. Это действие должно работать и в обратную сторну, то есть при получении объекта в формате JSON, он должен быть преобразован в строку для дальнейшей вставки в адресную строку браузера. Другой неотъемлемой частью данного модуля является прослушивание события браузера при любом изменении адреса строки – это необходимо для распознования перехода на новую страницу пользователя. После нажатия пользователя на другую страницу или при изменении параметров данной старницы срабатывает событие об изменении адрес. Далее модуль берет новое значение адреса преобразует его в формат JSON и информирует вышестоящий модуль об изменении в состоянии приложения. Вышестоящий модуль обновляет состояние приложения исходя из нового стостояни адреса.

Для более понятной работы с адресами будет использоваться следующий формат адреса: имя страницы будет сопровождаться знаком «#», а далее через запятую будут идти все остальные параметры так, как это представлено на рисунке нижу.



Рисунок 4.3 – Формат адреса страницы

Так в приложении будет несколько страниц, то нужен суб-модуль для работы со страницей. Основые функции данного модуля:

* хранение объектов всех страницы приложения;
* хранение текущей активной старницы;
* создание старниц, если они не были созданы до этого;
* получение объекта новой старницы, исходя из нового состояния приложения;
* замена текущей активной страницы новой;
* передача новой странице нового состояния приложения.

Перейдем к созданию страницы – это будет еще один суб-модуль. Точнее это будет несколько модулей, каждый из которых представляет из себя отдельную страницу приложения. Это очень удобный подход, поскольку каждая страница обладает своей логикой и нет смысла описывать все в одном месте, к тому же это упрошает дальнейшую работы с каждой отдельной станицей, поскольку сразу понятно в каком модуле необходимо работать.

Несмотря на отдельную логику каждой страницы, все они должны иметь стандартный набор методов, чтобы модуль работы со страницами мог с ними коммуницировать. Каждая страница должна иметь следующие методы:

* initialize;
* focus;
* blur;
* pageName.

Метод «initialize» – это метод, который используется для инициализации старницы. То есть он вызывается только при первом создании страницы и носит конфигурационный характер. Во вермя выполнения данного метода созданются все объекты страницы, элементы и начальные состаяния.

Далее идет метод «focus». Данный метод вызывается каждый раз, когда происходит изменение состояния приложения, а именно, когда пользователь переходит на страницу данного модуля или когда он меняет какие-либо параметры для данной страницы. Исходя из данного описания можно понять, что данный метод служит отправной точкой для работы со страницей.

Метод «blur» – это метод, служаший для завершения работы страницы. Данный метод вызывается для активной старницы, когда происходит переключению на другую страницу. Во время вызова данного метода, текущая страница, своего рода, ставится на паузу, при этом она отменяет все текущие операции и запросы, закрывает все диалоговые окна, если таковые имеются. Этот метод очень необходим, так как позволяет освободить память под другую страницу, что значительно увеличивает производительность всего приложения.

Метод «pageName» – это метод, служаший для возврщения имени страницы. Он используется модулем работы со старницами для поиска страниц.

Станицы могут также содержать и другие методы, косающиеся только конкретной страницы. Эти методы должны быть закрыты для вышестоящих модулей и никак не должны зависеть от других модулей приложения.

Вернемся к модулю работы со страницыми. Без описания структуры каждой отдельной страницы нельзя было обойтись для дальйшего описания структуры данного модуля.

При изменении состояния приложения данный модуль должен как-то найти необходимую ему старницу. Для этого он хранит в себе хеш-таблицу всех известных ему проинициализированных страниц.



Рисунок 4.4 – Диаграмма деятельности изменения страницы

Каждый раз когда происходи изменение состояния, модуль берет имя новой страницы и ищет его хеш-таблице. Если страница не была найдена, значит на нее заходят впервые, модуль инициализирует объект данной станицы и выполняет повторный поиск. Как только страница будет найдена, модуль проверяет не является ли данная страница активной, ведь возможен вариант, что пользователь уже находиться на данной станице, но он лишь поменял конфигурации данной страницы. Если страница таже самая, то все, что нужно сделать, это лишь вызвать метод «focus» у текущей страницы и передать в нее новое состояние. Если же новая страница не является текушей, то вызывается метод «blur» для текушей страницы, затем новая страницы становится текушей для данного модуля, и затем для уже новой страницы вызвывается метод «focus» с новым состоянием. Схема работы модулю отражена на рисунке 4.4.

Может возникнуть ситуация, при которой в адресной строке будет указана страница, которая не известна модулю. В таком случае модуль всегда имеет страницу по умолчанию, на которую перейдет, если подобное случится.

Исходя из описанных методов видно, что дальнейшая разработка клиентской стороны заключается в написании каждой отдельной станицы и ее внутренней логики, однако в конечном итоге все упреться в совместную работу клиентской и серверной сторон. Для этого необходима разработка блока работы с клиентским приложением.

## **4.5** Блок работы с клиентским приложеним

Блок работы с клинестким приложеним представляет из себя API для работы сервером. Как уже описывалось ранее, сервер должен быть максимально закрытым для всем подклчений и запросов, поэтому API должен быть доступен строго по одному адресу на сервере, а остальные его действия будут зависеть от параметров переданных в запросе. Для большей безопасности будет использоваться POST-запрос.

POST-запрос — один из многих методов запроса, поддерживаемых протоколом HTTP. Метод предназначен для запроса, при котором сервер принимает данные, заключённые в тело запроса, для хранения. Он часто используется для загрузки файлов или представления заполненной веб-формы.

В отличие от него, метод GET предназначен для получения информации от сервера. В рамках GET-запроса некоторые данные могут быть переданы в строке запроса URI, указывающие, например, условия поиска, диапазоны дат, или другую информацию, определяющую запрос. В рамках POST запроса произвольный объем данных любого типа может быть отправлен на сервер в теле запроса. Поля заголовка в POST-запросе обычно указывают на тип содержимого.

Для упрощения работы с API он должен иметь строгую, но при этом удобную для понимания и работы структуру. Поэтому его архитектура должна быть схожа с архитектурой модуля для работы со страницами, работа которого описывалась в предыдущем разделе. То есть должен быть суб-модуль, который знает все известные и возможные API-запросы. К основным функциям главное API-модуля можно отнести:

* хранение объектов всех возможных API-сущностей;
* расшифровка запросов от клиента;
* вызов методов API-сущностей в соответствии с запросом;
* предотвращение вызовов несуществующих API-запросов.

API-сущность – это отдельный модуль для работы с конекретным элементов базы данных. То есть для каждой таблицы базы данных должна существовать своя сущность. Сделать это следует для того, чтобы каждая сущность выполняла операции только с одной таблицей и никак не затрагивала другие. Это делает архитектуру более прозрачной и помогает при дальнейшем развитии проекта, ведь каждая таблица имеет свой свойства и конфигурацию.

Однако для работы модуля API все сущности должны иметь общий интерфейс – методы, которые будет вызывать API-модуль. Для этого каждая сущность должна содержать следующие методы:

* add;
* remove;
* get;
* set.

Метод «add» служит для создания новой записи в таблицу. Он также имеет свой интерфейс входных параметров, который описывает чуть выше с использованием языка TypeScript. Это очень удобная вешь, так как данный интерфейс в дальнейшем будет использоваться на клиенте при написании кода, то есть программист всегда может увидеть набор необходим параметров для данного вызова метода. Использование интерфейсов делает разработку одновременно строгой и, в тоже время, удобной и безопасной, так как при малейшем несовпадении интерфейса TypeScript выдаст ошибку о несовпадении струкрутры данных или типов данных. Как метод «add», так и остальные методы сущности должны возвращать в качестве результата Promise.

Promise (другое название «промис») – предоставляют удобный способ организации асинхронного кода. Но представляет собой специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

Объект Promise очень выжная в разработке вещь, так как делает работу сервера и клиента асинхронной. Без данной возможности страничка браузера бы просто напросто замирала на время обработки операции. Кратко описать работу Promise можно следущим образом:

* код, которому нужно выполниться осинхронно, создает объект Promise и возвращает его;
* внешняя функция, получает объект Promise и слушает его обработчики;
* при завершении процесса осинхронны код переводит Promise в одно из описанных выше состояний, то есть fulfilled или rejected, при этом автоматически вызываются обработчики во внешней функции.



Рисунок 4.5 – Состояния объекта Promise

Метод «remove» служит для удаления элемента из таблицы. Обязательным для него параметром служит идентификационный номер строки для удаления.

Метод «get» служит для получения данных из таблицы. Параметр при вызове данного метода должен быть опциональным. То есть при вызоме метода без параметра должны возвращаться все данные из таблицы, но при наличии параметра, поиск должен осуществляться сугубо по полученным параметрам. Для каждой API-сущности должен быть описан свой интерфейс для поиска – это также будет помогать разработку клиентской стороны делать необходимые и правильные запросы.

Метод «set» пердназначен для обновления записи таблицы. Он также имеет обязательный параметр с интерфейсом всех полей таблицы, чтобы программист не потерял в процессе разработки одно из полей.

Таким образом описывается каждая сущность API. В результате чего все сущности храняться в виде хеш-таблицы в API-модуле для быстрого нахождения и работы.

И все же, основным методом данного модуля является метод обработки запросов клинетской стороны. После получения запроса от клиента в формате JSON, модуль должен убедиться, что данный запрос возможен, поэтому первым делом к какой именно сущности направлен запрос.



Рисунок 4.6 – Диаграмма деятельности обработки API-запроса

Если сущность существует, то проверяется, какой метод данной сущности будет вызван – и только после данной проверки API-модуль берет необходимую сущность из хеш-таблицы и вызывает необходимый метод с нужными параметрами. Если же какой-то из этапов работы API-модуля не удался, то он возвращает ошибку, которая в итоге отправляется в качестве ответа клиенту.

Для увелечения надежности API-модуль так провеят заголовки каждого запроса, а именно:

* метод запроса;
* хост запроса;
* тип данных запроса.

Метод запроса всегда должен быть POST, а запроса должны приходить искочительно от того же хоста, на котором запушен сервер, чтобы другие ресурсы не могли получить доступ к информации. Так как сервер настроен на работу с данными в формате JSON, то имеет смысл делать проверку на тип данных запроса, он должен быть всегда «application/json». При несовпадении одного из загловком API-модуль так же вернет в качестве ответа ошибку. На рисунке 4.6 предсатавлена диаграмма деятельности по обработке API-запроса.

На этом работа над функциональностью API-модуля окончена, теперь имеет смысл вернуться на клентскую сторону и разработать модуль для работы с API.

## **4.6** Клиентский блок для работы с API

Для удобной работы с API необходим модуль, который будет основную работу по созданию запроса на север, заполнению нужных заголовков, а также проверке результата запроса.

Основное, что делает данный модуль – это создает Fetch-запрос. Это XMLHttpRequest-запрос нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на Promise. При вызове Fetch возвращает промис, который, когда получен ответ, выполняет коллбэки с объектом Response или с ошибкой, если запрос не удался. Данный метод принимает два параметра: адрес, куда пойдет запрос, и опции:

* метод запроса;
* загаловки запроса;
* тело запроса;
* мод, который указывает режим кросс-доменности, в котором должен быть выполнен запрос;
* кеш, который указывает как кешировать запроса и нужно ли это делать;
* переадресация, которая указывает как должен вести запрос при возникновении ошибки с кодами 300.

Как видно, всевозможных настроек у данного типа запросов очень много, поэтому именно он используется в качестве основного метода для отправки запросов на сервер.

При отправке запроса заполняется неаобходимый набор параметров и загаловков, а так же тело запроса. После получения ответа от сервера, модуль должен проверить соответсвуют ли заголовки ответа ожидаемым и, если все проверки пройдены, то модуль преобразует данные в JSON формат и отправляет в вышестоящий модуль для дальнейшей обработки.



Рисунок 4.7 – Диаграмма деятельности модуля клиенского API

Если же в процессе выполнения запроса произошла ошибка или ответ не соответствует ожиданиям, то модуль возвращает в вышестоящий модль ошибку. На рисунке 4.7 изображена работа модуля по отправке и обработке запроса к серверу.

Теперь связь клиента и сервера налажена и из любой части клиенского приложения можно запросить необходимые данные. Однако каждый раз вызывать метод клиентского API и вписывать в него необходимый набор параметров очень накладно и небезопасно, так как при любом изменении в API нужно будет делать правки по всему проекту. Поэтому имеет смысл сделать, своего рода, обертку над клиентским API.

Данная обертка будет содержать в себе все известные типы запросов API, а также все их возможные методы, как это показано на рисунке 4.8. То есть разработчику будет нужно лишь пройти по цепочке методов модуля и в вызванном методе лишь передать параметр соответствующий нужному интерфейсу вызова.



Рисунок 4.8 – Обертка методов для работы с клинетским API

Данная модель значительно упрощает работу разработчика, а также увеличивает надежность кода. Это также сказывается и на дальнейшем сопровождении продукта, так как при малейшем изменени в API или добавлении новых типов запросов, все правки нужно будет делать лишь в одном файле, что является большим плюсом. На рисунке 4.9 представлен пример удобного использования обертки над клинетским API модулем.



Рисунок 4.9 – Пример вызова методов обертки над клиентским API

Теперь и клиентская и серверная части полностью готовы к взаимодействию друг с другом, а это значит, что теперь может начинаться работа над интерфейсами и логикой все клиентских страниц приложения.