Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехника и комплексная автоматизация

КАФЕДРА Системы автоматизированного проектирования

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ НА ТЕМУ:

РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ МОБИЛЬНОГО ПРИЛОЖЕНИЯ ПО ПРОКАТУ И АРЕНДЕ АВТОМОБИЛЕЙ

Студент РК6-83Б (Группа)	_____ (Подпись, дата)	Г.Ю.Карпушкин (И.О.Фамилия)
Руководитель ВКР	_____ (Подпись, дата)	Н.В.Пивоварова (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Нормоконтролер	_____ (Подпись, дата)	С.В.Грошев (И.О.Фамилия)

2018 г.

Аннотация

Данная работа посвящена проектированию серверной части современного многофункционального мобильного приложения, анализу существующих рыночных решений и выбору подходящих средств реализации мобильного сервиса проката и аренды автомобилей.

В ходе выполнения были проведены следующие шаги проектирования:

- Обзор существующих конкурентных решений;
- Выбор серверной архитектуры;
- Выбор клиентской архитектуры;
- Составление успешного сценария использования;
- Создание инфологической модели;
- Низкоуровневая реализация.

Конечным итогом проделанной работы является спроектированное полнофункциональное мобильное приложение, работающее по принципу клиент-серверной архитектуры, соответствующее всем требованиям технического задания.

Работа содержит 88 страниц, 7 разделов, 21 иллюстрацию и 1 таблицу. Выполнена с использованием 14 источников.

Содержание

ВВЕДЕНИЕ

В современном мире популярность мобильного интернета растет стремительными темпами. Если в 2012 году среднее количество времени, которое пользователь проводил в мобильном интернете, составляло 74,4 минуты в день, то во втором квартале 2014 года данный показатель достиг 108,6 минут в день. Это, по данным авторитетного интернет-издания Statista.com, составляет прирост практически в полтора раза. Как следствие, в 2014 году произошло значимое для онлайн-бизнеса событие – впервые в истории интернета активность мобильных пользователей превзошла активность пользователей ПК и составила 55% от общего времени пользования глобальной сетью.

Данная статистика однозначно указывает на то, что в настоящее время конечный пользователь предпочитает использовать мобильные устройства: смартфоны и планшеты, нежели стационарные компьютеры или ноутбуки.

Мобильные технологии имеют ряд неоспоримых преимуществ перед любыми другими видами маркетинговых коммуникаций. Рассмотрим кратко каждые из них:

1. Мобильные платежи.

Скорость современной жизни диктует свои условия: клиент хочет иметь возможность совершать покупки и оплачивать их быстро, удобно и в любом месте. Интернет-платежей уже не хватает. Мобильные платежи означают, что продавец, независимо от размеров своего предприятия, может принять и оформить платеж кредитной картой при помощи смартфона, счета и соответствующего приложения.

2. Геолокация.

Теперь не только клиент может посмотреть в своем смартфоне свое месторасположение и окружающие его объекты, но и владелец бизнеса, используя так называемый LBS-сервис, может находить потенциальных клиентов и оповещать их о своих услугах.

3. Создание мобильной версии для своего коммерческого сайта.

Теперь нет необходимости для того, чтобы воспользоваться интернет-сервисом, каждый раз подходить к компьютеру или ноутбуку, находящимся дома или в офисе. Всё теперь «в кармане» – в быстром доступе.

4. Мобильные приложения делают услуги доступнее для клиентов.

Они позволяют размещать информацию о товарах и услугах в полном объеме, оповещать клиентов об акциях и специальных предложениях без использования SMS-рассылок, а также в необходимый именно вам момент.

5. Оффлайн-режим.

Мобильные приложения могут работать в режиме оффлайн, то есть в то время, когда подключение к интернету отсутствует. Это огромное преимущество, так как оно позволяет компании всегда быть «на связи» со своим потенциальным клиентом, даже тогда, когда у него нет средств на счёте мобильного телефона. Как только пользователь сможет подключиться к сети, тогда все изменения будут автоматически загружены в его приложение.

Наличие приложения на мобильном устройстве позволяет пользователю всегда быть в курсе событий: своевременно получать сведения о новых продуктах и товарах, оплачивать счета, осуществлять бронирование различных услуг, заказ товаров и т.д. Наличие специально разработанных приложений дает существенные плюсы и компаниям, среди которых и возможность постоянно «держать за руку» своих клиентов. Продавец может получать отзывы напрямую от своих клиентов, без посредников (таких, как интернет-сайты или ваши сотрудники) и прямо в момент совершения покупки.

С мобильным приложением может появиться доступ к каждому отдельному клиенту! Мобильное приложение можно наполнить не только текстовым, но и фото/видео контентом. Ограничений практически не существует.

1. Предметная область

Сегодня большим спросом пользуется услуга по аренде и прокату автомобилей по всему миру. Отсутствие личного или даже служебного автомобиля может стать серьезной проблемой, особенно, если необходимо встретить в аэропорту партнеров по бизнесу, родственников, друзей или просто провести день в разъездах. Общественный транспорт здесь не поможет, такси – слишком дорогое удовольствие, поэтому в случае поломки или отсутствия собственного авто лучшим выходом будет его аренда.

Прокат автомобиля без водителя связан с рядом важных преимуществ, среди которых:

1. Возможность выбора автомобиля, соответствующего случаю марки и класса – для разъездов по городу, для деловых поездок или для встречи высокопоставленных гостей.
2. Отсутствие необходимости в техническом обслуживании и содержании машины при полной уверенности в ее исправности, высокой степени надежности и безопасности.
3. Высокая мобильность в любых ситуациях, возможность быстрой подмены вышедшего из строя личного авто.

С развитием интернета арендовать автомобиль стало значительно легче. Прокатные компании предоставляют услуги онлайн-бронирования через свои сайты, где можно быстро найти подходящий по цене и классу автомобиль в нужном месте и заранее его забронировать. Но что, если у выбранной прокатной компании нет офиса в нужной локации или нет автомобиля под ограниченный бюджет? Очевидно, надо искать другого прокатчика. Это становится крайне неудобным, если финансовый бюджет сильно ограничен или желаемое место аренды не самое популярное с точки зрения туризма. В таком случае поиск автомобиля может занять слишком много времени, пока арендатор не найдет подходящий сайт и не выберет удовлетворяющий его автомобиль.

Было бы удобно пользоваться интернет-сервисом, который сможет найти автомобиль по заданным критериям не в одной прокатной компании, а сразу

во многих. Таким образом, поиск будет осуществляться в разы быстрее и эффективнее, что пойдет на пользу клиенту, который будет удовлетворён выбору, арендодателю, который получит свою выгоду с аренды, а сам сервис будет иметь финансовую прибыль за предоставление посреднических услуг.

Необходимо разработать мобильное приложение, предоставляющее возможность человеку арендовать автомобиль с помощью мобильного телефона или планшета в любой точке мира. Оно позволит пользователю не заботиться заранее о бронировании, но осуществить его по надобности с помощью мобильного устройства, что экономит много времени и порой крайне необходимо в экстренной ситуации. Поскольку мобильных платформ много, необходимо разработать несколько его версий под разные системы, но в первую очередь под iOS и Android, как самых популярных.

Мобильное приложение не должно быть ограничено лишь русским языком. Многообразие языков должно привлечь пользователей со всего мира и позволит обойти многих конкурентов. Дополнительный плюс обусловлен тем, что оплатить услугу бронирования намного комфортнее со смартфона, нежели чем через веб-сайт. К тому же, мы учтем недостатки существующих решений. Следует также предусмотреть и тщательно продумать интерфейс (дизайн) приложения, чтобы у пользователя, скачавшего его на своё устройство, было желание открыть его снова. Ну, и конечно, сам алгоритм поиска должен быть эффективным, за это отвечает серверная часть. Поэтому важно грамотно подобрать технологии и спроектировать не только мобильное приложение, как клиентскую сторону, но и серверную, как главную составляющую успеха.

1.1. Обзор существующих решений

На сегодняшний день разработано и успешно функционирует достаточное число таких сервисов. Проведя сравнение самых популярных из них, я выделил несколько важных недостатков и преимуществ.

Недостатки:

1. Нет мобильности использования. Комфортное пользование сервисами осуществимо только с помощью компьютера или ноутбука, но не с мобильного устройства небольшого размера. Такой минус не дает гибкости пользования продуктом.
2. Отсутствие кроссплатформенности. Некоторые частные арендодатели всё же имеют мобильное приложение, но только под одну из известных операционных систем: iOS или Android, что не дает в полной мере реализовать мобильный потенциал для большинства пользователей.
3. Отсутствие мультиязычности. Большинство сайтов предоставляют контент строго на одном языке, что заранее ограничивает круг потенциальных пользователей.
4. Отталкивающий дизайн. Большинство сервисов, даже те, которые пользуются большим спросом в сети, визуально неприятны. Этот фактор является ключевым для успеха интернет-проекта, поскольку, прежде всего, конечный пользователь обращает внимание не на качество продукта, а не его «обёртку».

Преимущества:

1. Эффективный поиск. Действительно, можно достаточно быстро найти автомобиль, который удовлетворит даже самого требовательного пользователя.
2. Широкий круг предоставляемых услуг. Помимо услуги по бронированию автомобиля, многие сервисы предоставляют возможность забронировать авиабилеты, номера в гостинице и т.п.

1.2. Техническое задание

Разработать клиент-серверную архитектуру, предназначенную для функционирования мобильного приложения по прокату и аренде автомобилей. Изучить предметную область, учесть преимущества и недостатки существующих рыночных решений.

Требования к серверной части:

- Отказоустойчивость;
- Быстрая обработка данных;
- Надежное хранилище;
- Возможности для масштабирования;
- Интеграция со сторонними сервисами;
- REST API архитектура;
- Применение свободного программного обеспечения;
- Кроссплатформенность;
- Мультиязычность.

2. Высокоуровневая архитектура

Исключительная задача мобильного приложения – предоставить данные пользователю в удобном формате. Однако сами данные генерирует не мобильное приложение, а его серверная часть, которая располагается на удаленном узле. Алгоритм получения данных следующий: мобильное приложение запрашивает данные по сети интернет у приложения сервера, сервер в зависимости от запроса собирает данные (или генерирует их) и возвращает их пользователю, попутно обращаясь в хранилище. Мобильное приложение, получив сырой набор данных, формирует визуальное представление и отображает его пользователю. На этом принципе работает клиент-серверная архитектура. Поэтому важно понимать, что мобильное приложение лишь средство получения и отображения данных.

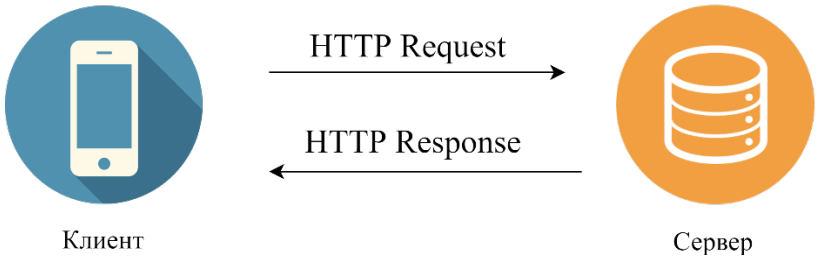


Рис. 1. Высокоуровневая клиент-серверная архитектура

Возникает справедливый вопрос: каким образом происходит взаимодействие клиента и сервера: с помощью чего они общаются? Для организации диалога между клиентом и сервером разработаны некоторые правила – протоколы взаимодействия, в частности протокол HTTP, на котором базируется весь интернет.

Поскольку наше мобильное приложение постоянно должно получать актуальную информацию от сервера через сеть интернет, необходимо определить, по средствам каких протоколов будет осуществляться это взаимодействие. На текущий момент базовым протоколом в сети интернет является протокол HTTP.

2.1. Протокол HTTP

HTTP – широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов. Он предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту. После этого клиентское приложение может продолжить отправлять другие запросы, которые будут обработаны аналогичным образом.

Задача, которая традиционно решается с помощью протокола HTTP, – обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером.

Как правило, передача данных по протоколу HTTP осуществляется через TCP/IP-соединения. Серверное программное обеспечение при этом обычно использует TCP-порт 80, хотя может использовать и любой другой.

Каждый запрос, отправленный через HTTP протокол, должен включать следующее:

1. Строка запроса с указанным методом и версией HTTP;
2. Заголовки запросов и также их значение;
3. Тело запроса.

Метод HTTP запроса определяет основные свойства запроса и может быть одним из последующих значений: OPTIONS, GET, HEAD, POST, PUT, PATCH, DELETE, TRACE и CONNECT.

3. Серверное приложение

В промышленной веб-разработке серверное приложение является наиболее важным звеном клиент-серверной архитектуры. Это связано, прежде всего, с тем, что сервер является источником данных, а клиент – их получателем. Поэтому первостепенной задачей серверного приложения является обеспечение целостности и сохранности данных. Второй важный аспект – их эффективная обработка. Как правило, наибольшие трудности вызывает именно второй пункт. В этой главе мы разберемся с серверной архитектурой нашего приложения и выберем оптимальный стек технологий для реализации требований технического задания.

Любое серверное многофункциональное приложение, работающее по принципу клиент-серверной архитектуры, состоит из двух взаимодействующих базовых элементов: управляющая программа, написанная на каком-либо языке программирования, и хранилище данных. В программе заложена логика работы приложения: она определяет, как обработать тот или иной запрос, пришедший от клиента. Хранилище же предназначено исключительно для хранения информации, и, как правило, для этого используется база данных. Конечно, программа может обращаться в хранилище за данными, и она же должна их должным образом обрабатывать.

Как следствие, качество работы приложения определяется эффективным программным обеспечением и скоростью обработки данных. На сегодняшний день существует большое количество технологий, с помощью которых можно реализовать практически любой требуемый функционал. Так в чем же различие и превосходство одной технологии над другой? Как правило, в промышленной разработке важную роль играет не только качество производимого продукта, но скорость достижения результата. Именно этот фактор заставляет выбирать среди множества технологий одну – наиболее оптимальную.

Но прежде, чем что-то выбирать, необходимо посмотреть, какие технологии бывают, чем они отличаются и в каких случаях необходимо использовать ту или иную из них.

Введем некоторые базовые понятия.

Традиционно в современных технологиях выделяют 3 уровня абстракции:

1. Чистый (нативный) язык программирования – это материал, из которого можно сделать все, что угодно. Ограничения накладываются самим языком. Под словом «чистый» понимается отсутствие каких-либо сторонних дополнений (изменений), вносимых в функции языка извне. На чистом языке написаны все крупнейшие веб-приложения мира с многомиллионной посещаемостью.

2. Фреймворк – это некая среда разработки для программиста с готовыми правилами и инструментами. Задача фреймворка – упростить процесс разработки за счет внедрения уже готового функционала. С одной стороны, фреймворк помогает и ускоряет разработку, а с другой – накладывает определенные ограничения. При помощи фреймворков разрабатываются проекты средней и большой сложности. На сегодняшний день это самый популярный способ разработки.

3. CMS – это уже готовое решение, конструктор, в котором пользователь по частям собирает нужный проект. Его скорее не программируют, а настраивают. Здесь ограничений на порядок больше, чем в фреймворке, и выйти за их границы крайне сложно и неэффективно. Обычно на CMS разрабатываются простые сайты с посещаемостью до миллиона пользователей в месяц. Как показывает статистика последних лет, фреймворки постепенно вытесняют CMS с рынка веб-разработки.

Чаще всего один уровень абстракции базируется на другом. То есть, на чистом языке делают фреймворки, а на фреймворках пишут CMS, но не наоборот. В связи с этим производительность написанного ПО и скорость его создания тоже разная: чем выше уровень абстракции – тем хуже производительность, и чем ниже абстракция, тем дольше разрабатывать приложение.

Наша задача выбрать оптимальную технологию, которая позволит создать не только эффективно работающее приложение, но и реализовать его за достаточно короткие сроки.

Выделим наиболее важные критерии при выборе технологий:

- Размер и сложность проекта;
- Скорость разработки;
- Наличие готовых решений и большого сообщества;
- Стоимость и доступность специалистов;
- Отказоустойчивость решения;
- Тренд его развития технологии;
- Наличие подробной документации;
- Стоимость поддержки;
- Требования к нагрузкам;
- Требования к безопасности;
- Возможности интеграции с другими решениями.

Для начала необходимо определиться с масштабами и сложностью нашего проекта. Часто тип приложения говорит сам за себя, и можно сразу определить, какое решение будет наиболее оптимальным: использовать CMS платформу, фреймворк или что-то другое.

Традиционно проекты принято разделять на три группы:

1. Простые.

К ним относятся сайты-визитки, одностраничные лендинги и простые интернет-магазины. Они не предназначены на большую аудиторию и большие серверные нагрузки. Такие задачи имеют массу стандартных, проверенных годами, решений. Обычно их реализуют на стандартных («коробочных») решениях, CMS-платформах или шаблонах, чтобы максимально быстро и дешево получить требуемый результат.

2. Средние.

К ним относятся сложные интернет-магазины и маркетплейсы, порталы национального масштаба и разнообразные многофункциональные сервисы. Стандартного набора функционала CMS систем не хватает, поэтому такие решения обычно разрабатываются с применением фреймворков.

3. Сложные.

К ним относятся огромные порталы, направленные на многомиллионную аудиторию, социальные сети, инновационные и нетиповые проекты. Для их реализации требуется максимальная производительность от серверной части. Обычно такие сервисы реализуют на нативном языке программирования и их разработка, как правило, самая долгая.

Проанализировав поставленную задачу, наше приложение можно отнести ко второй группе. Действительно, разрабатываемое приложение предназначено для большой аудитории пользователей, оно должно взаимодействовать с массой сторонних сервисов и шаблонное решение здесь явно не подойдет, хотя задача не уникальная. В то же время предметная область не предполагает больших нагрузок, таких, как, например, в социальных сетях. Как говорилось выше, важно, чтобы разрабатываемое программное обеспечение было написано максимально быстро, но и максимально эффективно: качество не должно превалировать над скоростью.

Для достижения этого эффекта целесообразно воспользоваться готовым фундаментом – фреймворком. Так и поступим. Для большей ясности дадим более подробную характеристику понятию фреймворк.

Фреймворк – это некая программная платформа, которая определяет структуру приложения, облегчающая разработку технически сложных проектов.

Как правило, фреймворк содержит базовые программные модули, а вся логика и функционал проекта реализуется разработчиком на их основе. За счет этого достигается не только высокая скорость разработки, но и надежность решения. Важно понимать, что фреймворк – это программное обеспечение, написанное и работающее на одном из языков программирования. Соответственно, чтобы разрабатывать на фреймворке, нужно знать язык, на котором он написан.

Таким образом, мы подошли к самой важной части проектирования серверного приложения – выбор серверного языка программирования.

3.1. Язык программирования

На сегодняшний день существует огромное количество разных языков программирования, пригодных для использования на серверной стороне клиентского приложения. И, более того, на всех популярных языках есть примеры огромных проектов. Если 10 лет назад, говоря о технологиях больших приложений, говорили преимущественно про Java, то сегодня это может быть почти любой язык программирования и утверждать, что использовать нужно конкретно один – стереотип. Это, прежде всего, связанно с развитием самих языков: за последнее десятилетие многие сильно продвинулись в развитии и получили широкие возможности. Конечно, каждый язык чем-то отличается и, выбирая тот или иной из них, необходимо руководствоваться объективными критериями с оглядкой на задачи проекта.

На чистом языке, без использования фреймворков и коробочных решений, пишутся огромные проекты с повышенными требованиями по гибкости, нагрузкам и безопасности. Для таких задач часто бюджет и сроки не играют такого значения, как эффективность. Чем больше проект, тем больше требований, а значит, проще писать все с нуля, выделяя на это лучших специалистов. Но неужели фреймворки имеют настолько узкий функционал, что не способны помочь в таких ситуациях? Совсем нет. Проблема фреймворков кроется в другом. В погоне за универсальностью и богатым функционалом для любых нужд страдает производительность. Тот объем кода, который обеспечивает простоту пользования фреймворком, потребляет слишком много ресурсов сервера, что является недопустимой роскошью для высоконагруженных проектов, поэтому разработчикам приходится использовать нативный подход в разработке.

Чем больше проект, тем больше стек технологий, который в нем используется. В огромных порталах может использоваться сразу несколько языков программирования. Опять же, это связано с объективным критерием выбора технологий. Часто один язык может хорошо делать одну задачу, а другой – другую.

За годы развития IT наиболее широкую популярность приобрели следующие языки программирования: PHP, Python, Ruby, Java, C++, C# и JavaScript. Про каждый из них можно говорить достаточно долго, но я выделю наиболее важные аспекты для дальнейшего выбора применимо к нашему проекту.

PHP – самый популярный серверный язык программирования. Разработан специально для веб-программирования. Его используют как простые, так и большие многофункциональные приложения. Имеет самую большую коллекцию коробочных решений и фреймворков под любые задачи. Относительно недорогие программисты и лёгкий порог вхождения. Несколько лет назад считался антитрендом в индустрии до выхода 7 версии языка, где он получил действительно мощные возможности и десятикратный прирост производительности.

Наиболее яркие примеры: Facebook, ВКонтакте.

Python – один из самых современных и продвинутых языков общего назначения. Имеет элегантный синтаксис и одну из самых больших библиотек расширений. Наибольшую популярность получил в научных сферах деятельности. Используется для средних и больших проектов. Программистов найти сложнее, и стоят они дороже. Имеет более скромную коллекцию фреймворков, чем предшественник.

Наиболее яркие примеры: Instagram, Pinterest.

Ruby – современный язык программирования, разработка на нем такая же быстрая как на PHP и Python. Обычно используется для разработки простых и средних проектов, часто применяют для стартапов. Программистов крайне мало, и они дорогие. Самый популярный, и, пожалуй, единственный серьезный фреймворк: Ruby on Rails.

Наиболее яркие примеры: 500px, Groupon.

Java – сильно типизированный объектно-ориентированный язык программирования. Разработка на нем крайне долгая и дорогая. Его используют в основном для больших проектов со специфическими требованиями, чаще всего в коммерческой сфере. Однако является самым популярным языком программирования в рейтинге TIOBE.

Наиболее яркие примеры: Ebay, Amazon.

C# – аналог Java, также используются для крупных приложений. Имеет огромную экосистему .NetFramework, поддерживаемую корпорацией Microsoft.В отличие от всех остальных технологий не является кроссплатформенным решением, что накладывает серьезные экономические ограничения.

Наиболее яркие примеры: Guru, Stack Overflow, Bank of America.

JavaScript – очень быстро развивающаяся технология, тренд последних лет. Огромное количество наработок и библиотек, можно написать все, что угодно, даже игры. Применим для средних и больших проектов, но действительно мощные возможности этот язык получил сравнительно недавно с появлением Node.js фреймворка, потому примеров больших проектов пока мало. Специалисты самые дорогие и найти их сложнее всего.

Наиболее яркие примеры: LinkedIn, Walmart, PayPal.

C++ – самый возрастной и сложный для освоения язык программирования, но в то же время самый быстрый. Используется только в высоконагруженных проектах, где требуется максимальная производительность. Найти грамотных специалистов сложнее всего и разработка на нем самая длительная.

Наиболее яркие примеры: Mail.ru, Google.

Выше я описал самые популярные на сегодняшний день серверные технологии. Каждая из них завоевала свою аудиторию и имеет успешные проекты, которые доказывают, что большие веб-приложения могут быть написаны на разных языках программирования, и иметь многомиллионную аудиторию. Конечно, существует много новых языков, которые очень быстро набирают популярность, в частности Scala, но пока они довольно молодые и сырые.

Этот анализ показал, что принципиальной разницы между современными технологиями нет, успеха можно добиться с любой из них, грамотно подобрав инструменты реализации. Но, чтобы процесс разработки был максимально эффективным, выбирать технологию нужно под требования конкретного проекта.

Наше приложение не рассчитано на большие нагрузки, но в то же время хотелось бы получить максимально эффективную систему для работы с данными за кратчайший промежуток времени. Также важно отметить, что приложение должно обладать широким надёжным функционалом, который легко поддерживать и развивать. В связи с этим мой выбор падает на язык PHP: он прост в освоении, на нем легко писать, имеет огромное количество фреймворков и библиотек, что является определяющим фактором, а версия языка 7.2 по производительности обгоняет Python, Ruby, JavaScript и Java. Также этот язык является кроссплатформенным, что позволит запускать его на любой операционной системе, в отличие, например, от C#.

3.2. Фреймворк

Одним из важнейших аргументов в пользу PHP при выборе технологии было большое количество фреймворков. Действительно, за свое более чем двадцатилетнее существование языка сообществом было написано свыше 30 фреймворков, различающихся как размером, так и функционалом. На текущий момент в группу лидеров выбилось пять из них: Symfony, Laravel, CodeIgniter, Zend Framework и Yii. По данным исследовательского портала coderseye.com на февраль 2018 года самым популярным фреймворком является Laravel. Ниже приведена диаграмма, иллюстрирующая процентное распределение используемых на рынке фреймворков.

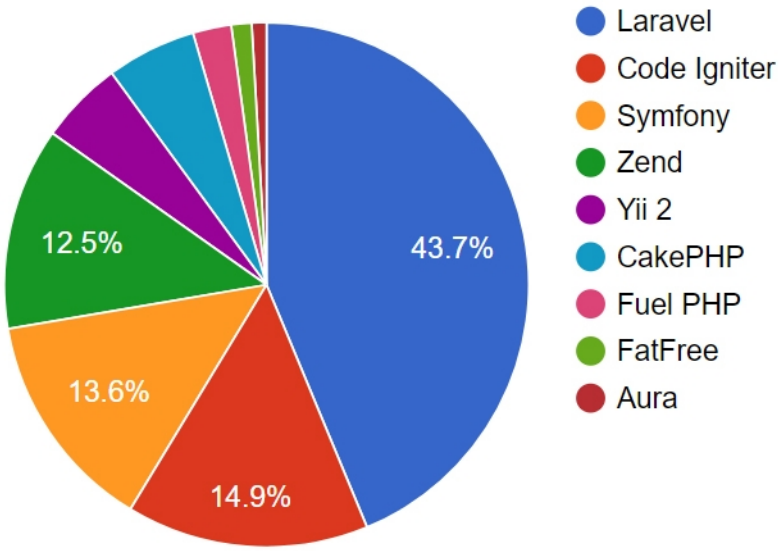


Рис.2. Диаграмма долей использования PHP фреймворков

В целом, большинство популярных фреймворков предлагают схожий функционал: комфортную среду для быстрого старта, широкие возможности для расширения и внутреннюю экосистему. Различия же кроятся во внутренней философии разработки, которой придерживались авторы – создатели. Вот почему для каждого проекта следует подбирать определенный инструмент для достижения результата.

При выборе фреймворка важно обратить внимание на следующие аспекты:

1. Размер сообщества.

Большая популярность среди сообщества разработчиков свидетельствует о качестве используемого инструмента. В современном мире IT известность и положительная динамика роста свидетельствует о том, что фреймворк успешно функционирует, будет развиваться и обрастать новыми возможностями. Поэтому стоит обратить внимание на те инструменты, которые находятся в тренде.

2. Устойчивость.

При выборе фреймворка для долгосрочного проекта необходимо удостовериться, что авторы будут поддерживать его разработку в течение продолжительного времени. В дальнейшем это гарантирует своевременную модернизацию и сопровождение уже написанных приложений.

3. Поддержка.

Стоит обратить внимание на то, насколько легко возможно найти ответы на возникающие вопросы в ходе разработки. Как правило, если фреймворк создан достаточно давно, то с этим аспектом не должно быть проблем.

4. Безопасность.

Любое приложение потенциально уязвимо. Чтобы снизить риск взлома, необходимо выбрать фреймворк, который сможет предоставить надежные методы защиты информации.

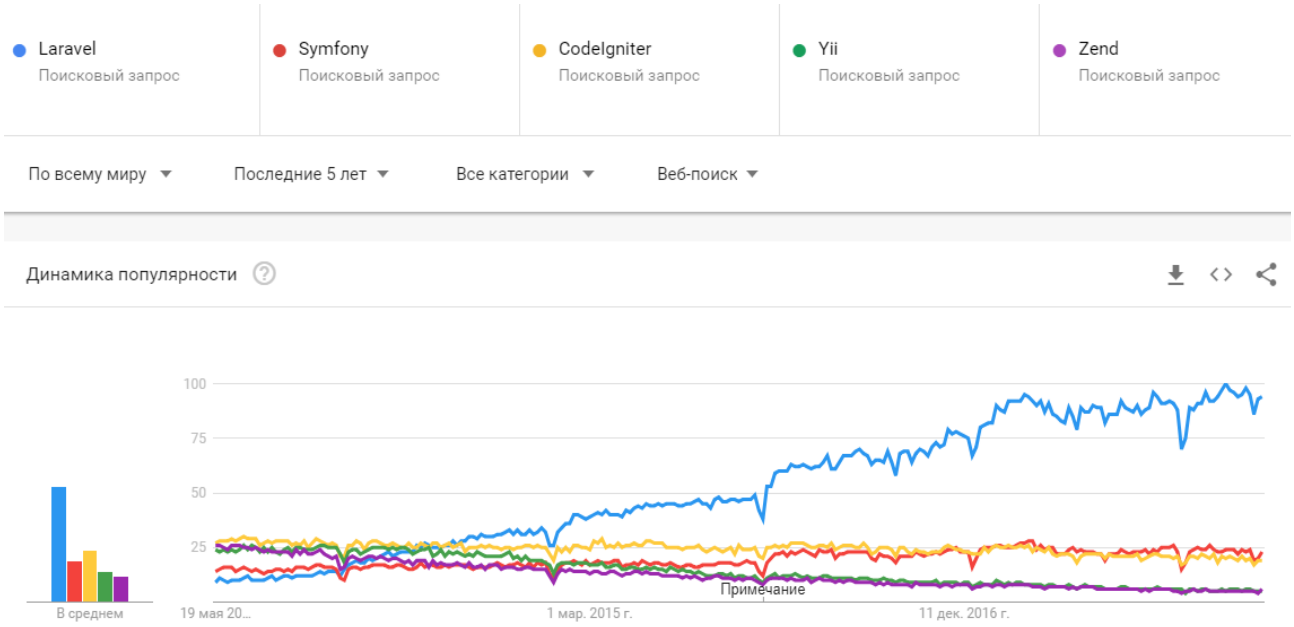
5. Документация.

Необходимо оценить объём и качество написанной документации. Во-первых, инструмент, у которого есть хорошая документация вызывает большее доверие. Во-вторых, такой инструмент проще использовать. Как правило, качество написанной документации полностью отображает отношение создателей фреймворка к людям, которые его используют.

6. Количество специалистов.

Для грамотной разработки на выбранном фреймворке необходимы специалисты, хорошо его знающие. Рынок должен обладать такими специалистами.

Ранее я говорил о пяти самых популярных PHP фреймворков. Каждый из них довольно давно находится на рынке и используется для создания больших веб-приложений. Чтобы оценить тенденцию и рост популярности каждого из них, я воспользовался статистикой поисковых запросов за пять последних лет, которая предоставляется порталом google.com. Ниже приведен график динамики популярности:



Наглядно видно, что фреймворк Laravel активно набирает популярность, в то время как его ближайшие конкуренты в лице Symfony, Yii, CodeIgniter и Zend Framework постепенно сдают свои позиции. Если фреймворк набирает популярность, то разработчикам он нравится, и, следовательно, стоит прислушаться к выбору сообщества.

Однако есть еще как минимум 5 критериев, которые нужно учесть при принятии окончательного решения. Для объективного выбора фреймворка сравним лучшие из них и остановимся на одном.

Заранее стоит отметить, что каждый из сравниваемых фреймворков находится в свободном доступе и разрабатывается большим сообществом опытных программистов.

1. Laravel

   Плюсы:

   - Самый быстроразвивающийся фреймворк в мире;
   - Самая подробная и понятная документация;
   - Превосходная экосистема;
   - Легкая масштабируемость;
   - Идеально подходит для коммерческих проектов;
   - Масса доступных плагинов;
   - Наименьший порог входа;
   - Надежные пакеты шифрования;
   - Превосходный синтаксис.

   Минусы:

   - Не самая оптимальная работа с базами данных;
   - Скромная производительность.

2. Symfony

   Плюсы:

   - Отличная производительность;
   - Один из самых возрастных фреймворков;
   - Самое большое сообщество;
   - Некоторые компоненты фреймворка считаются эталоном;
   - Имеет самые широкие возможности для расширения.

   Минусы:

   - Высокий порог входа;
   - Плохая документация;
   - Самые дорогие специалисты.

3. Yii

Плюсы:

- Самый производительный PHP фреймворк;
- Имеет низкий порог входа;
- Один из старейших фреймворков с большим сообществом;
- Предназначен для быстрой разработки;
- Чрезвычайно гибкий;
- Неплохая документация.

Минусы:

- Сложная организация кода;
- Имеет огромные внутренние ограничения;
- Сложно конфигурируется.

4. CodeIgniter

Плюсы:

- Очень дружелюбный к разработчику;
- Грамотная внутренняя архитектура;
- Имеет хорошую документацию;
- Предназначен для быстрой разработки.

Минусы:

- Плохо тестируется;
- Сложно настраивать сторонние пакеты;
- Морально устарел;
- Мало специалистов.

5. Zend

Плюсы:

- Авторы фреймворка – создатели языка PHP;
- Идеально подходит для корпоративной разработки;
- Имеет хорошую документацию;
- Хорошо оптимизирован;
- Содержит огромное количество средств защиты.

Минусы:

- Плохая документация;
- Очень долгая разработка;
- Сложно найти специалистов.

Как видно, каждый фреймворк имеет свои преимущества и недостатки. Авторитетные порталы medium.com и coderseye.com выделяют группу из трех самых прогрессивных фреймворков: Laravel, Symfony и Yii. Я разделяю это мнение и считаю, что CodeIgniter и Zend Framework проигрывают конкурентам по всем статьям, разумеется, в контексте наших задач. Оставшиеся три достаточно сложно сравнивать.

Laravel имеет превосходную, пожалуй, лучшую документацию из всех. Его легко масштабировать, он идеально подходит для коммерческой разработки, но имеет не самую лучшую производительность. Yii же, напротив, потребляет меньше ресурсов сервера, но накладывает большие ограничения на разработку. Symfony построен на идеальной архитектуре, имеет самое большое сообщество разработчиков, но крайне сложен для освоения. Каждый фреймворк предоставляет надежные механизмы защиты от любого вида атак. Устойчивость и развитие гарантирует большое сообщество программистов и сторонних компаний, принимающих участие в жизни фреймворков.

Среди оставшихся трех кандидатов фреймворк Yii проигрывает Laravel и Symfony, прежде всего, своим масштабом: последние два являются титанами веб-разработки.

На мой взгляд, самый совершенный из них – Laravel. Его синтаксис близок к идеалу, документация и экосистема располагает к изучению. Он находится в тренде, имеет большое развивающееся сообщество и массу полезных инструментов, позволяющих оптимизировать написанное приложение.

Подводя итог, я однозначно делаю выбор в пользу Laravel.

Итак, мы определились с серверным языком программирования и фреймворком, на базе которого будет работать управляющая программа серверного приложения. На текущем этапе схема серверного приложения выглядит так:

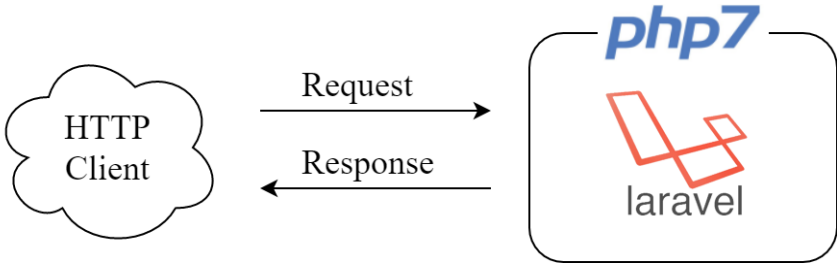


Рис. 3. Схема взаимодействия клиента и серверной программы

Следующим шагом необходимо определиться с хранилищем данных. В роли хранилища традиционно выступает специальное программное обеспечение, называемое базой данных. Как правило, самая большая нагрузка оказывается именно на нее. Чтобы максимально разгрузить базу без потери информации, разработчики серверной архитектуры применяют кэширование данных. Таким образом, достигается наибольшая скорость извлечения и обновления информации.

Далее мы подробно опишем логику взаимодействия программы, базы данных и кэш-хранилища нашего приложения.

3.3. База данных

Самое ценное, что есть в мире технологий – информация. С самого первого дня развития технологий вопрос о способах хранения, обработки и извлечения информации являлся одним из наиболее острых. По мере развития технологий сложность структур данных и объем информации, который нужно хранить и обрабатывать, стремительно росли вверх. Это привело к тому, что сегодня невозможно представить работу какого-либо приложения без специального программного обеспечения. Одним из таких приложений, получивших наибольший спрос, стала СУБД.

СУБД (Система управления базами данных) – специализированное программное обеспечение, предназначенное для организации и ведения баз данных. Эти приложения управляют или помогают управлять пользователю наборами хранимых данных. Так как эти данные могут быть разного формата и размера, были созданы разные виды таких систем.

СУБД основаны на моделях баз данных – определённых структурах для обработки данных. Каждая СУБД создана для работы с одной из них с учётом особенностей, выполняемых операций над информацией. Хотя решений, реализующих различные модели баз данных, очень много, периодически некоторые из них становятся очень популярными и используются на протяжении многих лет. Каждая СУБД реализует одну из моделей баз данных для логической структуризации используемых данных. Эти модели являются главным критерием того, как будет управлять информацией приложение. Существует несколько таких моделей, среди которых самой популярной является реляционная – РСУБД. Хотя она и является весьма мощной и гибкой, есть ситуации, решения на которые она предложить не может. Тут на помощь придёт сравнительно новая модель, называемая NoSQL. Она набирает популярность и предлагает весьма интересные решения. Из-за того, что эти системы не используют строгую структуризацию данных, они предлагают большую свободу действий при обработке информации.

3.3.1. Реляционная модель

Представленная в 70-х годах, реляционная модель является самой старой и проверенной. Она предлагает математический способ структуризации, хранения и использования информации в виде набора данных с предопределенными связями между ними. Эти данные организованны в виде набора таблиц, состоящих из столбцов и строк. В таблицах хранится информация об объектах, представленных в базе данных. В каждом столбце таблицы хранится определенный тип данных, в каждой ячейке – значение атрибута. Каждая строка таблицы представляет собой набор связанных значений, относящихся к одному объекту или сущности. Каждая строка в таблице может быть помечена уникальным идентификатором, называемым первичным ключом, а строки из нескольких таблиц могут быть связаны с помощью внешних ключей. К этим данным можно получить доступ многими способами, и при этом реорганизовывать таблицы БД не требуется.

Для взаимодействия с РСУБД используется специально разработанный язык SQL, который в 1890-х годах стал международным стандартом. Этот стандарт поддерживается всеми популярными ядрами реляционных баз данных. Некоторые из них также включают расширения стандарта, поддерживающие специфичный для них функционал. SQL используется для добавления, обновления и удаления строк ста блиц, извлечения наборов данных для обработки транзакций и аналитических приложений, а также управления всеми аспектами работы базы данных.

Благодаря десятилетиям исследований и разработки, РСУБД работают производительно и надёжно. В сочетании с большим опытом использования разработчиками реляционные базы данных стали выбором, гарантирующим целостность и защиту информации от потерь.

Несмотря на строгие принципы формирования и обработки данных, РСУБД могут быть весьма гибкими, если приложить немного усилий.

3.3.2. Нереляционная модель

NoSQL – способ структуризации данных (или нереляционный подход), заключается в избавлении от ограничений при хранении и использовании информации. Такие СУБД используют неструктуризированный подход, предлагающий много эффективных способов обработки данных. Они могут использовать различные модели данных, включая столбчатые, документные, графовые данные и хранилища пар «ключ-значение» в памяти. NoSQL базы данных получили широкое распространение благодаря обеспечению надежной отказоустойчивости и низкой задержки на извлечение данных.

В отличие от традиционных РСУБД некоторые базы данных NoSQL позволяют группировать коллекции данных с другими базами данных. Такие СУБД хранят данные как одно целое. Эти данные могут представлять собой одиночный объект наподобие JSON и вместе с тем корректно отвечать на запросы к полям.

NoSQL подход не использует общий формат запроса, как SQL в реляционных базах данных. Каждое решение использует собственную систему запросов.

Базы данных NoSQL отлично подходят для приложений, требующих более высокой масштабируемости и более низкого времени отклика, чем могут обеспечить традиционные реляционные модели. В их числе многие приложения для больших данных, мобильные и интернет-приложения. За счет использования упрощенных структур данных и горизонтального масштабирования базы данных NoSQL, как правило, обеспечивают более высокую производительность.

3.3.3. Сравнение SQL и NoSQL

Для того, чтобы прийти к объективному выводу, проанализируем разницу между SQL и NoSQL подходами:

- Структура и тип хранящихся данных. Реляционные базы данных требуют наличия однозначно определённой структуры хранения данных, а NoSQL базы данных таких ограничений не ставят.
- Запросы. Все РСУБД реализуют единые SQL-стандарты, поэтому из них можно получать данные при помощи языка SQL. Каждая NoSQL база данных реализует свой способ работы с данными.
- Масштабируемость. Оба решения легко растягиваются вертикально (например, путём увеличения системных ресурсов). Тем не менее, из-за своей современности, решения NoSQL обычно предоставляют более простые способы горизонтального масштабирования (например, создания кластера из нескольких машин).
- Надёжность. Когда речь заходит о надёжности, SQL базы данных однозначно впереди.
- Поддержка. РСУБД имеют очень долгую историю. Они очень популярны, и поэтому получить поддержку очень легко. При необходимости решить проблемы с ними гораздо проще, чем с NoSQL, особенно, если проблема сложна по своей природе.
- Хранение и доступ к сложным структурам данных. Реляционные базы данных предполагают работу со сложными ситуациями, поэтому и здесь они превосходят NoSQL-решения.

Поскольку наш проект должен обеспечивать максимальную надежность данных и иметь возможности для масштабирования, мой выбор однозначно за реляционными базами данных.

3.3.4. Выбор СУБД

Алгоритм выбора СУБД для проекта практически аналогичен выбору языка программирования. Для выявления объективных критериев в пользу той или иной СУБД дадим каждой из них небольшую характеристику и определим плюсы и минусы той или иной системы.

Выбирать будем из тех, которые распространяются в свободном доступе. На сегодняшний день самыми популярными являются: SQLite, MySQL иPostgreSQL.

1. SQLite

Легко встраиваемая в приложения база данных. Так как эта система базируется на файлах, то предоставляет довольно широкий набор инструментов для работы с ней по сравнению с сетевыми СУБД. При работе с этой СУБД обращения происходят напрямую к файлам, вместо обращения к портам и сокетам в сетевых СУБД. Именно поэтому, благодаря технологиям обслуживающих библиотек, SQLite очень быстрая и мощная.

Преимущества:

- Файловая структура. Вся база данных состоит из одного файла, поэтому её очень легко переносить на разные машины.
- Используемые стандарты. Хотя может показаться, что эта СУБД примитивная, но она использует SQL. Некоторые особенности опущены, но основные все-таки поддерживаются.
- Отличная при разработке и тестировании. В процессе разработки приложений часто появляется необходимость масштабирования. SQLite предлагает всё, что необходимо для этих целей, так как состоит всего из одного файла и библиотеки, написанной на языке C.

Недостатки:

- Отсутствие системы пользователей. Более крупные СУБД включают в свой состав системы управления правами доступа пользователей. Обычно применение этой функции не так критично, так как эта СУБД используется в небольших приложениях.
- Отсутствие возможности увеличения производительности. Опять же, исходя из проектирования, довольно сложно выжать что-то более производительное из этой СУБД.

Заранее можно сказать, что эта СУБД является непригодной для нашего приложения.

2. MySQL

MySQL – это самая популярная в мире, полноценная серверная СУБД. Она очень функциональная, свободно распространяемая СУБД, которая успешно работает с различными сайтами и веб-приложениями. Обучиться использованию довольно просто, так как на просторах интернета легко найти большое количество информации. Стоит заметить, что благодаря популярности этой СУБД, существует огромное количество различных плагинов и расширений, облегчающих работу с этой системой.

Несмотря на то, что в этой СУБД не реализован весь SQL функционал, MySQL предлагает довольно много инструментов для разработки приложений. Так как это серверная СУБД, приложения для доступа к данным, в отличие от SQLite, работают со службами MySQL.

Преимущества:

- Простота в работе. Установить MySQL довольно просто, а дополнительные приложения с графическим интерфейсом, позволяют довольно легко работать с этой СУБД.

- Богатый функционал. MySQL поддерживает большинство функционала SQL.
- Безопасность. Большое количество функций, обеспечивающих безопасность, которые поддерживаются по умолчанию.
- Масштабируемость. MySQL легко работает с большими объемами данных и её крайне просто масштабировать.
- Скорость. Упрощение некоторых стандартов позволяет MySQL значительно увеличить производительность.

Недостатки:

- Известные ограничения. По задумке в MySQL заложены некоторые ограничения функционала, которые иногда необходимы в особо требовательных приложениях.
- Проблемы с надежностью. Из-за некоторых способов обработки данных MySQL (связи, транзакции, аудиты) иногда уступает другим СУБД по надежности.

3. PostgreSQL

PostgreSQL является самым профессиональным из всех трех рассмотренных нами СУБД. Она свободно распространяемая и максимально соответствует стандартам SQL. От других СУБД отличается поддержкой востребованного объектно-ориентированного и реляционного подхода к базам данных. Благодаря мощным технологиям Postgres очень производительна. Параллельность достигнута не за счет блокировки операций чтения, а благодаря реализации управления многовариантным параллелизмом. Хотя PostgreSQL и не может похвастаться большой популярностью (в отличие от MySQL). Несмотря на всю мощность функционала, существует довольно большое число приложений, облегчающих работу с этой СУБД. Сейчас довольно легко установить эту систему, используя стандартные менеджеры

пакетов операционных систем, затем работать с ней с помощью графических интерфейсов.

Достоинства:

- Открытое ПО соответствующее стандарту SQL.PostgreSQLбесплатное ПО с открытым исходным кодом. Эта СУБД является очень мощной системой, полностью соответствующей стандарту SQL.
- Большое сообщество.
- Имеет массу дополнений. Несмотря на огромное количество встроенных функций, существует очень много дополнений, позволяющих разрабатывать данные для этой СУБД и управлять ими.
- Расширения. Существует возможность расширения функционала за счет сохранения своих процедур.
- Объектность. PostgreSQLэто не только реляционная СУБД, но также и объектно-ориентированная система с поддержкой наследования и много другого.

Недостатки:

- Производительность. При простых операциях чтения PostgreSQL может значительно замедлить сервер и быть медленнее своих конкурентов, таких как MySQL.
- Популярность. По своей природе, популярностью эта СУБД похвастаться не может, хотя существует довольно большое сообщество.
- Хостинг. В силу вышеперечисленных факторов иногда довольно сложно найти хостинг с поддержкой этой СУБД.

Проанализировав каждый вариант, я делаю свой выбор в пользу PostgreSQL. Этот выбор основан на современном подходе к развитию этой СУБД, а также больших возможностей для расширения в дальнейшем.

В целом, Postgres практически не уступает MySQL, здесь скорее большую роль при выборе сыграл личностный фактор: лично я больше симпатизирую тренду развития PostgreSQL.

На текущий момент схема серверного приложения выглядит следующим образом:

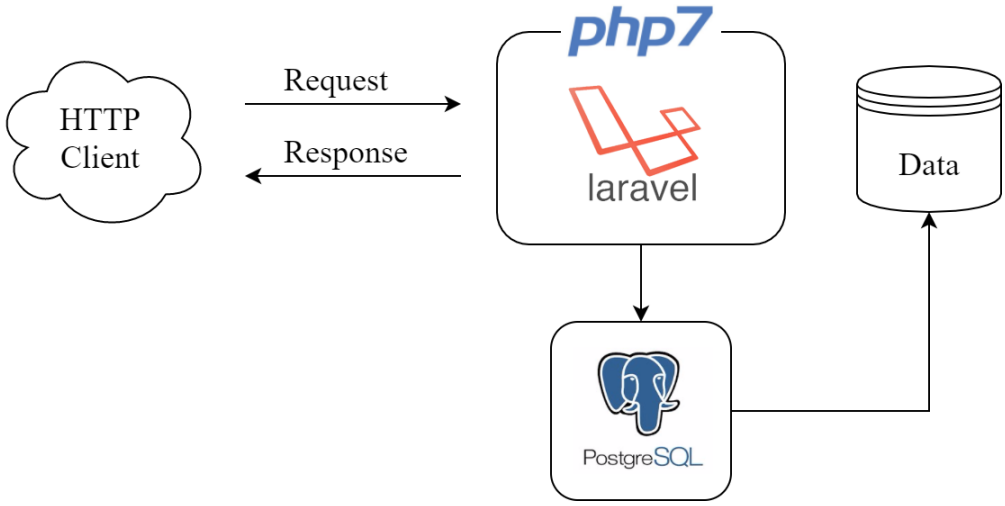


Рис. 4. Схема взаимодействия клиента, программы и базы данных

Чтобы максимально оптимизировать наше приложение и снизить количество обращений к базе данных будем использовать кэширование.

3.4. Кэширование

Кэширование – один из наиболее эффективных методов увеличения скорости извлечения информации. В частности, это единственный способ сделать больше и быстрее при использовании ограниченных ресурсов. А, как известно, ресурсы всегда ограничены: как серверные, так и пользовательские. Правильно настроенное кэширование дает огромный прирост в производительности, экономит трафик и уменьшает затраты на сервер. Представим, что необходимо осуществить быструю передачу информации. Может возникнуть проблема, когда скорость доступа к данным крайне низкая. Или другая ситуация: скорость хорошая, но мало доступной памяти или процессорные и дисковые факторы мешают осуществить задачу. В этом случае кэширование – единственный выход из ситуации.

Кэш – это некий промежуточный буфер, в котором хранятся данные. Эти данные не воссоздаются заново для каждого пользователя от вызова к вызову, а извлекаются из буфера. Детали хранения и способы организации структур данных у разных кэш-систем отличаются, но наибольшую популярность получили системы, хранящие информацию по принципу ключ-значение.

Под серверным кэшированием понимаются все виды кэширования, при котором данные хранятся на серверной стороне. Эти данные не доступны клиентским браузерам. Кэш создаётся и хранится по принципу «один ко многим» («многие» в данном случае – клиентские устройства).

Кешировать можно разными способами, например, записывать данные непосредственно на диск или в оперативную память. Выбор способа зависит от того, какие именно данные кэшируются, насколько интенсивно используется кэш и где находится «слабое звено» в производительности.

Конечно, кэширование не лишено недостатков. Одним из таких является потребность в большом количестве памяти сервера. Чем больше данных хранится в кэше – тем больше памяти он потребляет. Второй существенный недостаток – слабая надежность. Большинство кэш-хранилищ хранят информацию в ОП сервера. Это означает, что любой сбой приведет к потере данных.

Конечно, кэшировать можно все, но стоит ли? Очевидно, что на все не хватит ресурсов, поэтому стоит выделить те объекты, которые действительно важны.

Что нужно кэшировать:

1. Результаты вычислений и запросов в Базу данных. Кеширование данных, получаемых в результате сложных вычислений, может очень серьёзно ускорить работу приложения, а также уменьшить количество обращений и снизить нагрузку на базу данных.
2. Фрагментированные данные. Это наиболее распространённый тип кэширования, когда сохраняются отдельные блоки веб-страниц или объектов, которые одинаковы для большой группы пользователей.

В нашем случае наиболее актуальным является первый пункт, а именно кэширование результатов запросов базы данных. Существует правило: обновлений данных должно быть значительно меньше, чем чтения для их отдачи. То есть, не имеет смысл агрегировать то, что изменится в тот же момент, при этом важна актуальность агрегированных данных. Поэтому при разработке серверного приложения будем применять кэширование в тех местах, где чтение данных превалирует над их обновлением, чтобы напрасно не занимать ресурсы сервера. Если бы мы не применяли инструменты кэширования, данные пришлось бы регулярно извлекать из базы данных, что не очень эффективно. Теперь же мы имеем возможность получить данные в обход обращения к БД. Соответственно встает вопрос обеспечения актуальности информации, находящейся в кэше.

Общая логика обработки сервером клиентского запроса следующая.

Клиент запрашивает определенные данные по конкретному URL адресу сервера. Серверная программа (в нашем случае под управлением PHP) принимает этот запрос, определяет, какие данные хочет получить пользователь и пытается всячески найти их.

Первым делом приложение обращается в кэш-хранилище. Если оно содержит требуемую информацию, то программа извлекает ее и возвращает клиенту. Если кэш пуст, необходимо сделать запрос в базу данных, затем обновить соответствующим образом кэш-хранилище и только потом отдать данные клиенту. Блок-схема этого алгоритма представлена ниже:

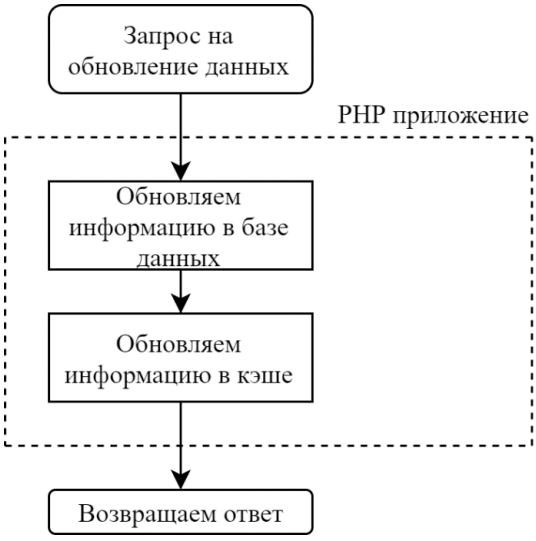


Рис. 5. Алгоритм чтения данных из Базы данных и кэш-хранилища

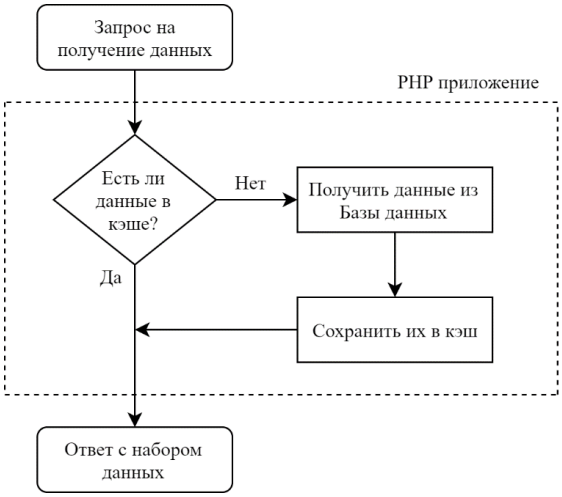


Рис. 6. Алгоритм обновления данных в Базе данных и кэш-хранилища

В случае запроса на обновление сначала нужно обновить информацию в базе данных, а затем информацию в кэше. Таким образом, мы поддерживаем постоянную актуальность информации, находящейся в кэше и исключаем риск потери данных.

3.4.1. Выбор хранилища

На текущий момент самыми популярными системами являются: Memcached и Redis. Оба инструмента – это мощные, очень быстрые хранилища данных в оперативной памяти, которые полезны в качестве кэша. Их возможности расширений, плагинов и функциональности практически идентичны, каждая из них легка в использовании и предоставляет возможности для масштабирования. Однако есть несколько фактов, которые определяют Redis более «крутой» системой:

1. Redis позволяет переносить данные на диск по истечению определенного времени. Это позволит повысить степень сохранность данных и откроет большие возможности.
2. Redis эффективнее работает с памятью сервера.
3. Redis набирает обороты. Уже сейчас многие большие компании переходят с Memcached на Redis, видя больше перспектив за его развитием.

Именно поэтому я выбираю Redis в качестве кэш-хранилища для серверного приложения.

На текущий момент схема серверного приложения выглядит следующим образом:

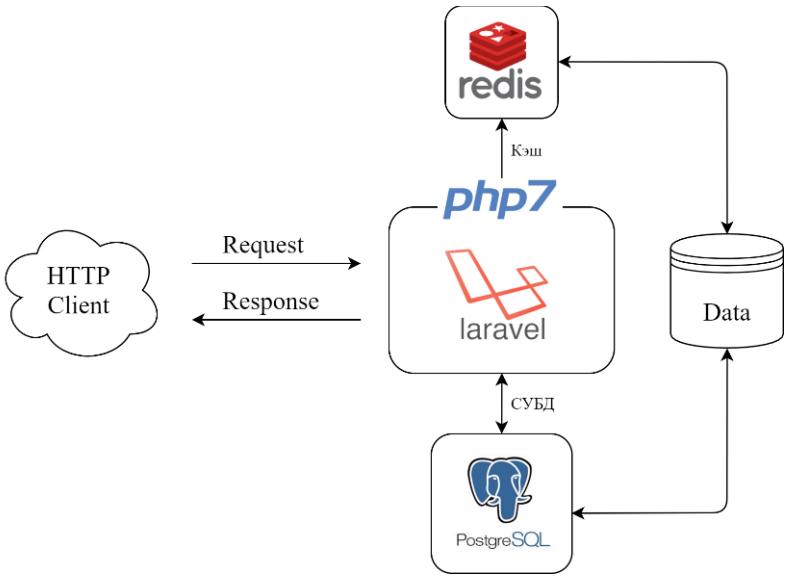


Рис. 7. Схема взаимодействия программы, базы данных и кэш-хранилища

3.5. HTTP Сервер

Веб-сервер – специализированное программное обеспечение, сервер, принимающий HTTP-запросы от клиентов и выдающий им HTTP ответы. Веб-сервер не является веб-приложением или языком программирования, подобно PHP. Его задача принимать запросы от клиентов, передавать их веб-приложению (например, PHP), получать ответ от веб-приложения и отдавать его пользователю. Необходимость в таком промежуточном звене возникает из-за сложности разбора запросов, посылаемых клиентами, и ради минимизации накладных расходов, возникающих при увеличении числа клиентов. Кроме того, вынесение куска функциональности в отдельную программу соответствует принципу модульности.

Клиент, которым обычно является веб-браузер или мобильное устройство, передаёт веб-серверу запросы на получение ресурсов, обозначенных URL адресами. Ресурсы – это HTML страницы, изображения, файлы, медиа-потоки или другие данные, которые необходимы клиенту.
В ответ веб-сервер передаёт клиенту запрошенные данные. Этот обмен происходит по вышерассмотренному протоколу HTTP.

На сегодняшний день существует достаточное количество веб-серверов, как коммерческих, так и бесплатных. Наиболее популярные из них: Apache, Nginx и IIS. Нам предстоит выбрать один, который будет оптимально взаимодействовать с нашим Laravel приложением. Это важный фактор, поскольку не всякий веб-сервер может взаимодействовать с PHP. Например, веб-сервер IIS, работающий только в экосистеме Microsoft и .Net, по определению не справится с этой задачей. Поэтому будем сравнивать наиболее мощные и популярные решения: Apache и Nginx.

3.5.1. Apache

Apache HTTP web-сервер – полное название платформы, распространяемой организацией Apache Software Foundation как открытое программное обеспечение. Веб-сервер Apache может работать на всех популярных операционных системах, но чаще всего он используется в рамках Linux. Именно в паре с СУБД MySQL и PHP скриптами образуется известный комплекс программного обеспечения LAMP web – сервер (Linux, Apache, MySQL, PHP), который повсеместно используется в сети Интернет.

С точки зрения функционала Apache имеет впечатляющие характеристики. Многие функции реализуются как совместимые модули, расширяющие базовый функционал, диапазон которых варьируется от поддержки языков программирования до обеспечения различных схем аутентификации. Например, это могут быть языки Perl, Python или PHP. Модули аутентификации включают в себя элементы управления доступом к различным директориям сервера, пароль, установление подлинности и так далее. Многие другие функции, такие как Secure Sockets Layer (SSL) или TLS (Transport Layer Security), также обеспечиваются модульной системой. Помимо этого, Apache поддерживает возможность развернуть несколько веб-сайтов или графических интерфейсов приложений. Веб-сервер может сжимать страницы, чтобы уменьшить их размер, что обеспечивает высокую скорость их загрузки. Наряду с высоким показателем безопасности, это является конкурентной чертой Apache.

Администраторы часто выбирают Apache из-за его гибкости, мощности и широкой распространенности. Он может быть расширен с помощью системы динамически загружаемых модулей и исполнять программы на большом количестве интерпретируемых языков программирования без использования внешнего программного обеспечения. Большим плюсом является возможность обрабатывать динамический контент средствами самого Apache, что упрощает конфигурирование: нет необходимости настраивать взаимодействие с дополнительным софтом.

Преимущества:

- Самое большое сообщество;
- Использует легко расширяемую систему динамических модулей;
- Очень просто перезаписывать URL в любой директории проекта;
- Очень просто настраивается;
- Имеет надежные механизмы защиты;
- Лучше других работает с динамическим контентом;
- Имеет долгую историю взаимодействия с PHP.

3.5.2. Nginx

Ngnix – это HTTP сервер и обратный прокси-сервер, а также TCP/UDP прокси-сервер общего назначения. Считается очень быстрым HTTP сервером. Вместо Apache или совместно с ним Nginx используют, чтобы ускорить обработку запросов и уменьшить нагрузку на сервер. Дело в том, что огромные возможности, заложенные модульной архитектурой Apache, большинству пользователей не требуются. Платить же за эту невостребованную функциональность приходится значительным расходом системных ресурсов. Для обычных сайтов, как правило, характерно явное «засилье» статичных файлов, а не скриптов. Никакого специального функционала для передачи этих файлов посетителю ресурса не требуется, так как задача весьма проста. А, значит, и веб-сервер для обработки таких запросов должен быть простым и легковесным, как Nginx.

Рабочие процессы в Nginx одновременно обслуживают множество соединений, обеспечивая их вызовами операционной системы. Обработку разобранного запроса осуществляет цепочка модулей, задаваемая конфигурацией. Формирование ответа клиенту происходит в буферах, которые могут указывать на отрезок файла или хранить данные в памяти. Последовательность передачи данных клиенту определяется цепочками, в которые группируются буферы.

Nginx не имеет возможности самостоятельно обрабатывать запросы к динамическому контенту. Для обработки запросов к PHP или другому

динамическому контенту Nginx должен передать запрос внешнему процессору для исполнения, подождать, пока ответ будет сгенерирован, и получить его. Затем результат будет передан клиенту. Для администраторов это означает, что нужно настроить взаимодействие Nginx с таким процессором, используя один из протоколов, который известен серверу. Это может немного усложнить процесс настройки, так как будет использоваться дополнительное соединение с процессором на каждый пользовательский запрос.

Основные преимущества:

- Обработка статического контента, индексных файлов, листинг директорий, кэш дескрипторов открытых файлов;
- Распределение нагрузки и отказоустойчивость;
- Модульная структура, поддержка различных фильтров: SSI, XSLT, GZIP, докачка, chunked ответы;
- Поддержка SSL и расширения TLS SNI;
- Ip-based или Name-based виртуальные сервера;
- Выполнение различных действий в зависимости от адреса клиента;
- Изменение URI с помощью регулярных выражений;
- Ограничение доступа на основе адреса клиента или по паролю;
- Ограничение скорости ответа клиенту;
- Ограничение количества одновременных подключений и запросов;
- Изменение настроек и обновление сервера без остановки работы.

Nginx позиционируется производителем как простой, быстрый и надёжный сервер, не перегруженный функциями. Применение Ngnix целесообразно, прежде всего, для статических веб-сайтов и как прокси-сервера перед динамическими сайтами. Наиболее эффективен для отдачи статических файлов.

3.5.3. Выбор HTTP сервера

На сегодняшний день Apache и Nginx – это два самых популярных веб-сервера в мире. С точки зрения возможностей, удобства использования и поддержки языков программирования они вполне схожи, но принципиальное отличие состоит в том, как эти сервера обрабатывают соединения и трафик.

Асинхронный ввод-вывод позволяет Ngnix экономить на этих запросах, в которых происходит передача информации. 10 000 клиентов, скачивающих файл на очень медленной скорости не страшны для Ngnix, но способны повергнуть в ужас Apache. Соответственно, Ngnix идеально подходит для отдачи статического контента. Конечно, для Apache есть решения, позволяющие справляться с большим числом одновременных запросов, но их ещё надо прикручивать, а в Ngnix это есть изначально.

В свою очередь Apache – это универсальный веб-сервер, который умеет делать всё: отдавать статические файлы и исполнять динамический код, причём с динамикой он справляется лучше.

Ниже приведены графики сравнения работы веб-серверов Apache и Ngnix:

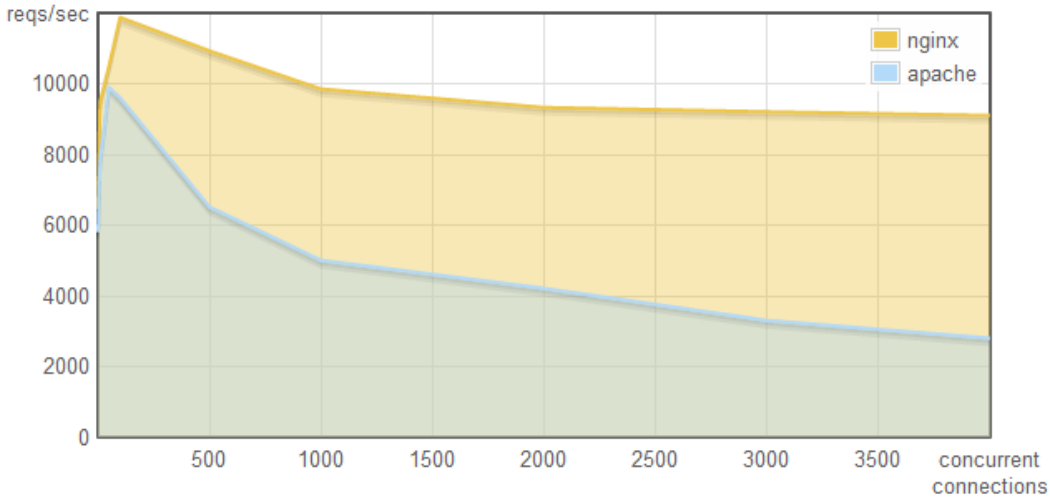


Рис. 8. Сравнение скорости отдачи страниц Nginx и Apache

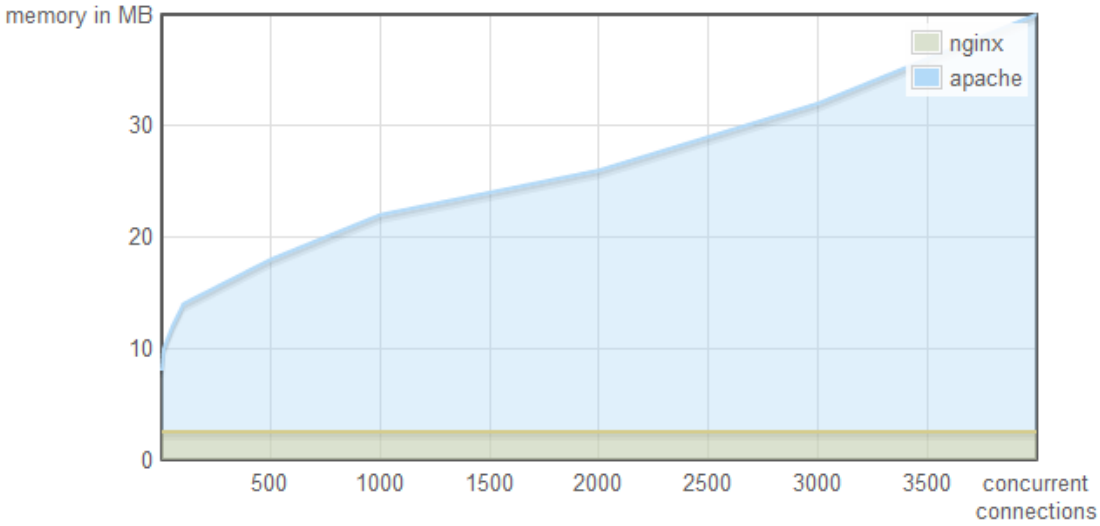


Рис. 9. Сравнение потребляемой памяти между Ngnix и Apache

Опираясь на вышеуказанные графики и известную статистику, можно сделать однозначный вывод: Ngnix более архитектурно совершенен, чем Apache.

Чтобы HTTP сервер работал наиболее оптимально, учтем сильные стороны и Apache, и Nginx.Поэтому сконфигурируем систему из двух веб-серверов. Поскольку Nginx прекрасно работает со статикой и может выступать в качестве сервера-балансировщика, то разумно возложить на него ответственность за отдачу статического контента (картинки, JS, CSS файлы). Apacheбудет отвечать за динамический контент и взаимодействие с PHP приложением. В этой схеме Nginx настраивается таким образом, чтобы весь программный код отдавался на исполнение Apache, а тот возвращал в Nginx только результат обработки – HTML или JSON (в нашем случае) код. К тому же Ngnix прекрасно умеет извлекать данные из кэш-хранилища Redis напрямую, в обход Apache и серверной программы соответственно.

Эта конфигурация позволяет горизонтально масштабировать приложение: мы сможем установить несколько back-end серверов за одним front-end, и Nginx будет распределять нагрузку между ними, увеличивая тем самым отказоустойчивость приложения.

Поскольку в качестве серверного языка программирования будет использоваться PHP, необходимо связать сервер Apache с интерпретатором PHP. Для этого необходима CGIпрограмма.

CGI (Common Gateway Interface, "общий интерфейс шлюза") – это стандарт, который описывает, как веб-сервер должен запускать прикладные программы (скрипты), как должен передавать им параметры HTTP-запроса, как программы должны передавать результаты своей работы веб-серверу. Прикладную программу, взаимодействующую с веб-сервером по протоколу CGI принято называть шлюзом, хотя более распространено название CGI-скрипт или CGI-программа. В качестве такого модуля будем использовать PHP-FPM, самый быстрый и функциональный шлюз.

На приведенной ниже схеме изображена полная схема взаимодействия клиента (мобильное приложение), HTTP-сервера и серверного приложения:

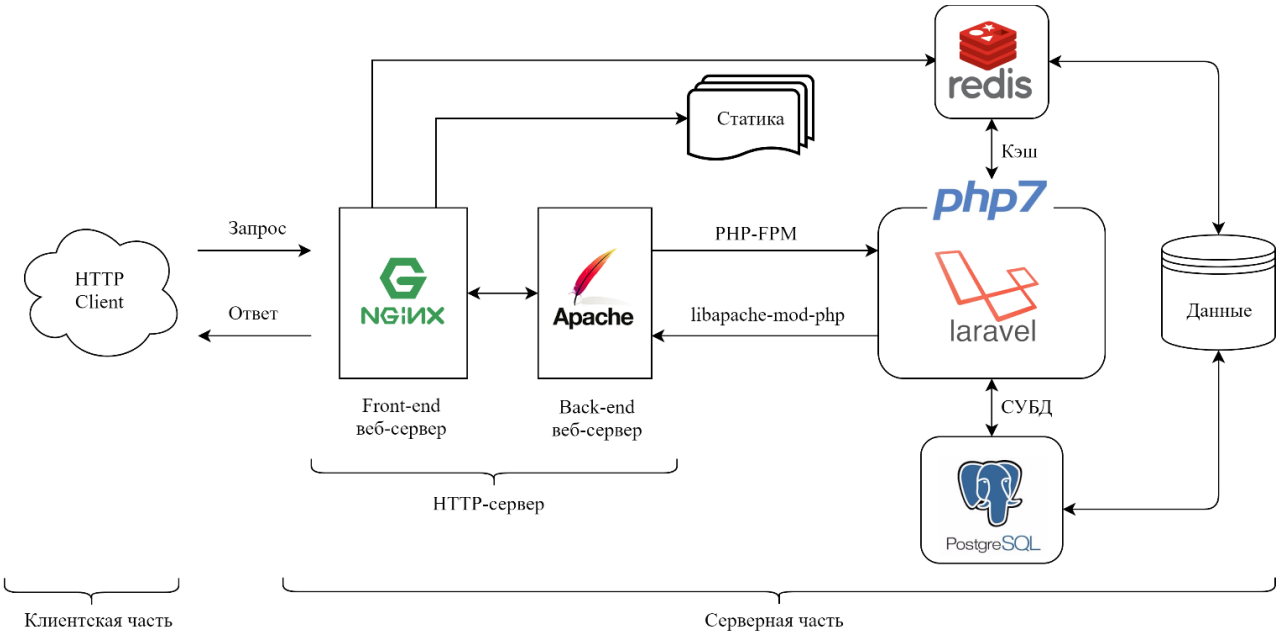


Рис. 10. Готовая архитектурна серверного приложения

3.6. API

Чтобы клиентское приложение могло взаимодействовать с сервером, необходимо организовать систему правил, следуя которым, клиент будет извлекать нужные данные, предоставляемые серверным приложением. Для этих целей разработчики используют концепцию API.

API (от англ. Application Program Interface) – это интерфейс взаимодействия между приложением клиента и сервером. Представляет собой ресурс, который сервер открывает для работы извне, т.е. клиент может воспользоваться им для получения доступа к функционалу программы, модуля или данных. API делает возможным работу ресурсов, которые используют потенциал, предоставляемый сервером. У нас такой ресурс – мобильное приложение.

Таким образом, необходимо организовать открытое API, обращаясь к которому клиент будет взаимодействовать с сервером.

Функции API делятся на 2 направления:

1. Возвращающие. На запрос стороннего приложения какого-либо метода с заданными параметрами сервер дает запрашиваемую информацию в определенном формате.
2. Изменяющие. Клиент вызывает некоторую функцию сервера, которая вводит новую информацию или изменяет на нем определенные настройки.

Чтобы сделать функции API наиболее понятными, в современной веб-разработке принято использовать так называемый RESTfulAPI стиль. Он накладывает определённые правила организации URL адресов таким образом, что переход по ним вызывает соответствующую функцию серверного приложения. В современном мире IT такой подход является правилом хорошего тона, поэтому будем его придерживаться.

3.6.1. REST API архитектура

REST (сокр. от англ. Representational State Transfer – «передача состояния представления») – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой системы:

1. Единый интерфейс;
2. Отсутствие состояний;
3. Кеширование ответа;
4. Клиент-сервер;
5. Многоуровневая система;
6. «Код по требованию».

Спланированная ранее архитектура серверного приложения в состоянии реализовать эти принципы, к тому же Laravel из коробки предоставляет удобные функции для создания полнофункционального RESTAPI.

Этот стиль предполагает использовать 4 (базовых) метода HTTP протокола таким образом, чтобы придавать запросам определённый смысл:

- GET – для получения данных;
- POST – для создания данных;
- PUT – для изменения данных;
- DELETE – для удаления данных.

Например, запрос типа GET по адресу: `/api/user` предполагает, что сервер вернёт список всех пользователей (user), имеющихся на сервере. Причем клиенту, пославшему этот запрос, совершенно не важно, откуда придут данные: из базы данных, серверного или сетевого кэша.

Чтобы запросить у сервера информацию о пользователе с номером 20 следует отправить следующий GET запрос:`/api/user/20`. Метод типа PUT по адресу `/api/user/20?name=Alex` заменит, имея пользователя с номером 20, на Alex.

Как можно заметить, URL адреса имеют определенную структуру следующего вида: `/api/{entity}/{id}`. Этот адрес состоит из трех частей:

1. `api` – идентификатор обращения к API;
2. `entity` – название сущности, например, `user` или `article`;
3. `id` – идентификатор сущности.

Конечно, запрос может содержать параметры для конкретизации выборки, но я опущу этот момент.

Таким образом, REST-подход позволяет удобно манипулировать сервером, используя единый интерфейс и интуитивно понятную логику запросов. В качестве формата обмена данными между клиентом и сервером будем использовать JSON. Это текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми, может использоваться практически с любым языком программирования. К тому же для многих языков программирования существует готовый код для создания и обработки данных в формате JSON.

4. Мобильное приложение

4.1. Функционал

Определившись с тем, откуда, как и в каком формате мобильное приложение будет получать данные, пора описать функционал мобильного приложения.

4.1.1. Поиск

Главным модулем приложения является поиск автомобиля и его оплата. Разберем его по шагам:

1. Поиск. Этот шаг представляет из себя страницу, с помощью которой пользователь задаст критерии и фильтры поиска. Например, адрес получения машины, максимальная цена, количество пассажирских мест, страховка и т.п. Эта страница является главной, поскольку выполняет основную функцию – поиск.
2. Выбор автомобиля. После задания критериев, мобильное приложение обратится к серверу с соответствующим запросом. Сервер вернет список доступных автомобилей и их прокатчиков. Пользователь просмотрит список и выберет подходящий автомобиль.
3. Дополнительные услуги. Перед тем, как оплатить выбранный автомобиль, следует предоставить пользователю возможность выбрать дополнительные опции аренды. Это может быть установка детского кресла, аренда навигатора, неограниченный пробег и т.п.
4. Оплата. После подсчета основной стоимости аренды и дополнительных услуг, пользователю необходимо произвести оплату. Для этого мобильное приложение предоставит возможность воспользоваться функциями ApplePay на устройствах под управлением ОС iOS и GooglePay на ОС Android. Если устройство не поддерживает такие возможности, то пользователь совершит оплату, заполнив небольшую форму. Быстро и удобно.

После успешного завершения оплаты, клиент получит квитанцию-чек с подробной информацией о месте и времени получения автомобиля, арендодателе, дополнительных услугах и прочей информацией.

Самая важная и сложная часть – обеспечение безопасность платежа. Для этого воспользуемся API системы PayPal. PayPal крупнейшая дебетовая электронная платёжная система, которая позволяет клиентам оплачивать счета и покупки, отправлять и принимать денежные переводы.

4.1.2. Личный кабинет

Личный кабинет – второй по важности модуль приложения. Он необходим для однозначной идентификации пользователя устройства. С помощью него мы сможем создать большую экосистему пользователей сервиса, что добавит нового функционала:

1. Создание отзыва на прокатную компанию, сервис или автомобиль;
2. Внутренний чат среди пользователей для обмена опытом;
3. Возможность сохранения понравившиеся машины или прокатной компании в раздел «избранное» личного кабинета;
4. Сохранять платёжные карты в особый раздел, чтобы каждый раз не заполнять платёжную форму, а использовать уже готовую;
5. Просматривать историю поиска и совершенных аренд.

Таким образом, мы позволим пользователям самим оценивать качество сервиса, что позволит новым клиентам доверять нам, а это немаловажно в сфере ведения бизнеса.

4.1.3. Информационный контент

Выше мы определили два основных динамических модуля, реализующих динамический функционал мобильного приложения. На начальном этапе проектирования этого достаточно.

Однако немаловажно иметь следующие информационные разделы:

- О нас. Этот раздел должен содержать контактную информацию о владельцах сервиса, описание функционала и возможностей приложения.
- Вопросы и ответы. Раздел для самых актуальных вопросов и ответов на них. Он необходим для того, чтобы пользователь мог оперативно получить ответы на волнующие вопросы, которые, как правило, однотипны.
- Отзывы. Отзывы пользователей необходимы для оценки качества услуг сервиса. Это стратегически важное значение, как некоторая оценочная характеристика.

Таким образом, 2 динамических и 3 информационных модуля организуют полный функционал сервиса. Далее определим внутреннюю архитектуру мобильного приложения.

4.2. Архитектура

Разработка любого программного продукта начинается с разработки архитектуры. Наиболее популярный паттерн проектирования – MVC.

4.2.1. MVC

Model-View-Controller (MVC) — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо друг от друга). Кратко опишем каждый компонент.

Модель (Model) предоставляет данные и реагирует на команды контроллера, изменяя свое состояние.

Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели.

Контроллер (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Основная цель применения этой концепции состоит в отделении бизнес-логики (модели) от её визуализации (представления, вида). За счёт такого разделения повышается возможность повторного использования кода. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения.

Этот паттерн является признаком хорошего тона программирования и активно используется при разработке любого рода программного обеспечения.

Ниже показана типичная схема концепции MVC:

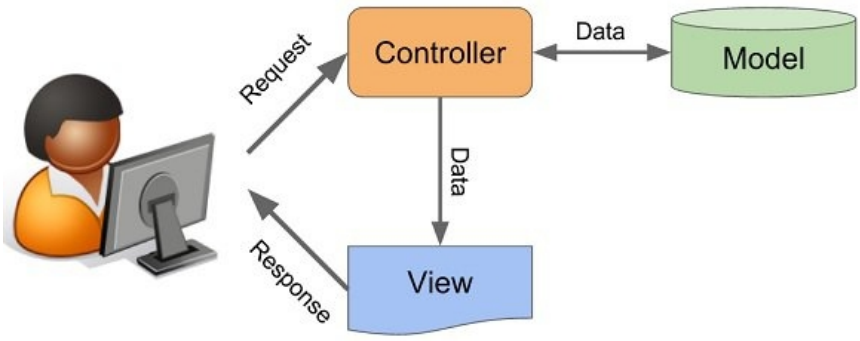


Рис. 11. Структурная схема MVC

В этой схеме главным звеном является контроллер. Его задача состоит в том, чтобы связать вид и модель. В свою очередь модель отвечает за сбор данных, в частности реализует обращение и диалог с сервером, а вид отвечает за визуализацию контента на устройстве пользователя.

Ранее мы говорили о том, что данные будут приходить в JSON формате, но не упомянули о том, какими будут эти данные: собранными видами или «сырым» набором информации (массивами). Наше приложение должно работать на нескольких платформах – это значит, что виды, ввиду особенностей операционных систем, будут различными. Во-вторых, логически неправильно заставлять сервер заниматься генерацией видов, его задача лишь в том, чтобы отдать собранный набор данных. Тогда клиентское приложение должно иметь набор шаблонов, которые будут подключаться соответствующим контроллером, который внедрит эти данные и соберет готовую страницу, а затем отдаст её пользователю. Таким образом, часть логики мы перенесем на клиентскую сторону, что также снизит нагрузку на сервер.

Стоит отметить, что концепция будет одинакова для всех мобильных платформ, реализация же может отличаться. Вся логика приложения будет идентична как для iOS устройств, так и для Android. Различия будут лишь на уровне программного кода.

4.2.2. Языки программирования

Поскольку мы планируем выпускать мобильные приложения для двух совершенно разных операционных систем (iOS и Android), средства разработки и выбор языков программирования для них совершенно разный.

4.2.3. Разработка под iOS

iOS-разработчику необходимо глубоко знать операционную систему macOS и саму iOS. У Apple своя экосистема, отличная от популярных Windows и Linux, со своими языками: Objective-C и Swift. Самый используемый язык программирования для iOS устройств: Objective-C, просто потому, что появился гораздо раньше (в середине восьмидесятых годов прошлого века), тогда как Swift лишь в 2014 году. Apple возлагает большие надежды на новый язык и много инвестирует в него. На сегодняшний день для поддержки старого ПО используется Objective-C, а новый пишется на Swift. Поэтому сразу определимся, что для разработки будем использовать Swift, и вот несколько причин:

- Swift более читаемый язык, чем Objective-C;
- Swift легче поддерживать;
- Swift требует меньшее количество кода;
- Swift быстрее.

Главный инструмент разработчика под macOS и вместе с ней iOS – среда программирования Xcode. Она включает средства для создания приложений для всех платформ Apple. Xcode содержит средство построения, редактор кода, поддерживающего все современные средства работы с кодом. Кроме того, не выходя из Xcode, можно протестировать приложение; если оно разрабатывается для внешнего устройства, то его можно запустить в эмуляторе. В систему включены эмуляторы всех устройств, новые версии которых можно скачать дополнительно.

4.2.4. Разработка под Android

Приложения для Android пишутся на языке программирования Java. Инструменты Android SDK (Software Development Kit – комплект разработки программного обеспечения) компилируют написанный программистом код – и все требуемые файлы данных и ресурсов – в файл APK, программный пакет Android, который представляет собой файл архива с расширением .apk. В файле APK находится все, что требуется для работы Android-приложения, и он позволяет установить приложение на любом устройстве под управлением системы Android.

Каждое приложение Android, установленное на устройстве, работает в собственной «песочнице» (изолированной программной среде):

- Операционная система Android представляет собой многопользовательскую систему Linux, в которой каждое приложение является отдельным пользователем;
- По умолчанию система назначает каждому приложению уникальный идентификатор пользователя Linux (этот идентификатор используется только системой и неизвестен приложению); система устанавливает полномочия для всех файлов в приложении, с тем, чтобы доступ к ним был разрешен только пользователю с идентификатором, назначенным этому приложению;
- У каждого процесса имеется собственная виртуальная машина (ВМ), так что код приложения выполняется изолированно от других приложений;
- По умолчанию каждое приложение выполняется в собственном процессе Linux. Android запускает процесс, когда требуется выполнить какой-либо компонент приложения, а затем завершает процесс, когда он больше не нужен, либо когда системе требуется освободить память для других приложений.

Таким образом, система Android реализует принцип предоставления минимальных прав. То есть, каждое приложение по умолчанию имеет доступ только к тем компонентам, которые ему необходимы для работы, и ни к каким другим. Благодаря этому формируется исключительно безопасная среда, в которой приложение не имеет доступа к недозволенным областям системы.

Поскольку Java является базовым и рекомендованным средством для разработки ПО, мы выберем его для написания Android приложения.

Итак, мы определились с тем, что для создания мобильного приложения под управлением ОС iOS будем использовать современный прогрессирующий язык Swift, а для устройств под управлением Android воспользуемся «опытным бойцом» Java.

В техническое задание не входит разработка клиентского приложения, поэтому подробно рассматривать вышеуказанные технологии не будем. Серверному приложению совершенно не важно, какой клиент к нему обращается.

5. Сценарии использования

Сценарий использования описывает поведение системы, когда она взаимодействует с кем-то (чем-то) из внешней среды. Система может отвечать на внешние запросы, а может сама выступать инициатором взаимодействия. Другими словами, сценарий использования описывает как внешний объект может взаимодействовать с рассматриваемой системой. Методика сценариев использования применяется для выявления требований к поведению системы, известных также как пользовательские и функциональные требования.

В системном проектировании сценарии использования применяются на более высоком уровне, чем при разработке программного обеспечения, часто представляя цели заинтересованных лиц. На стадии анализа требований сценарии использования могут быть преобразованы в ряд детальных требований.

Каждый сценарий использования сосредотачивается на описании того, как достигнуть цели или задачи. Это означает, что потребуется множество сценариев использования, чтобы определить необходимый набор свойств новой системы. Степень формальности программного проекта и его стадии будет влиять на необходимый уровень детализации для каждого сценария использования. Сценарии использования не должны путаться с понятием свойств системы, но тесно взаимодействуют с ними.

Сценарий использования определяет взаимодействие между внешними пользователями и системой, направленное на достижение цели. Актёр представляет собой роль, которую играет пользователь, взаимодействуя с системой. Например, человек может играть роль клиента, использующего банкомат, чтобы забрать наличные деньги, или играть роль сотрудника банка, использующего систему для пополнения банкомата купюрами.

В нашем случае основной функцией мобильного приложения является поиск автомобиля и его последующее бронирование. Поэтому определим её в качестве главного сценария для дальнейшей проработки.

5.1. Главный успешный сценарий

Главный успешный сценарий – это реализация одного из вариантов использования системы. Другие варианты будут добавляться в виде обновления и расширения данного приложения.

Составим последовательность действий пользователя, начиная от открытия мобильного приложения и заканчивая бронированием автомобиля:

1. Пользователь нажимает кнопку поиска;
2. Приложение отображает форму выбора локации;
3. Пользователь заполняет поля формы и подтверждает выбор;
4. Приложение предоставляет список доступных автомобилей по выбранной локации;
5. Пользователь выбирает наиболее понравившееся предложение;
6. Приложение показывает полную информацию по выбранному авто;
7. Пользователь запрашивает страницу бронирования;
8. Приложение предлагает заполнить форму бронирования для оплаты предложения;
9. Пользователь заполняет форму и подтверждает выбор;
10. Приложение высылает полную информацию о бронировании пользователя на почту или телефон;
11. Пользователь завершает диалог.

5.1.1. Расширения

3.1. В случае некорректного заполнения формы уведомить пользователя об ошибке и предложить обновить введенные данные.

4.1. Приложение не нашло автомобили в данной локации и предлагает изменить параметры поиска.

8.1. Приложение предлагает пользователю выбрать дополнительные опции: бронирование гостиницы, аренда детского кресла и т.п.

5.2. Системная диаграмма последовательности

Системная диаграмма последовательностей – это артефакт, иллюстрирующий последовательность действий в системе при выполнении сценария. На этой диаграмме отображаются внешние исполнители, которые непосредственно взаимодействуют с системой, сама система и связные с ней компоненты, из которой система извлекает, либо добавляет необходимые данные. Отображаются также системные события, инициированные пользователем. При этом порядок событий должен соответствовать их последовательности в описании сценария. В ходе составления диаграммы вариант использования может уточняться и дополняться.

В ходе описания архитектуры серверного приложения я говорил о том, что серверное приложение состоит из управляющей программы и хранилища данных.

Напомню, что управляющая программа написана на PHP фреймворке Laravel, а хранилище состоит из двух слоев: кэш-хранилища и базы данных. Вместе эти компоненты образуют единый узел – серверное приложение. Мольное приложение на диаграмме иллюстрирует совокупность клиентских реализаций единого интерфейса приложения под iOS и Android устройства. Пользователь – человек, непосредственно осуществляющий взаимодействие с устройством.

Ниже приведена схема высокоуровневой диаграммы последовательности действий, реализующей успешный сценарий. Сразу отмечу, что мы не будем подробно рассматривать клиентскую составляющую, поскольку наибольший интерес вызывает именно серверное приложение. Тем не менее, на общей диаграмме её стоит указать для глобального понимания схемы взаимодействия.

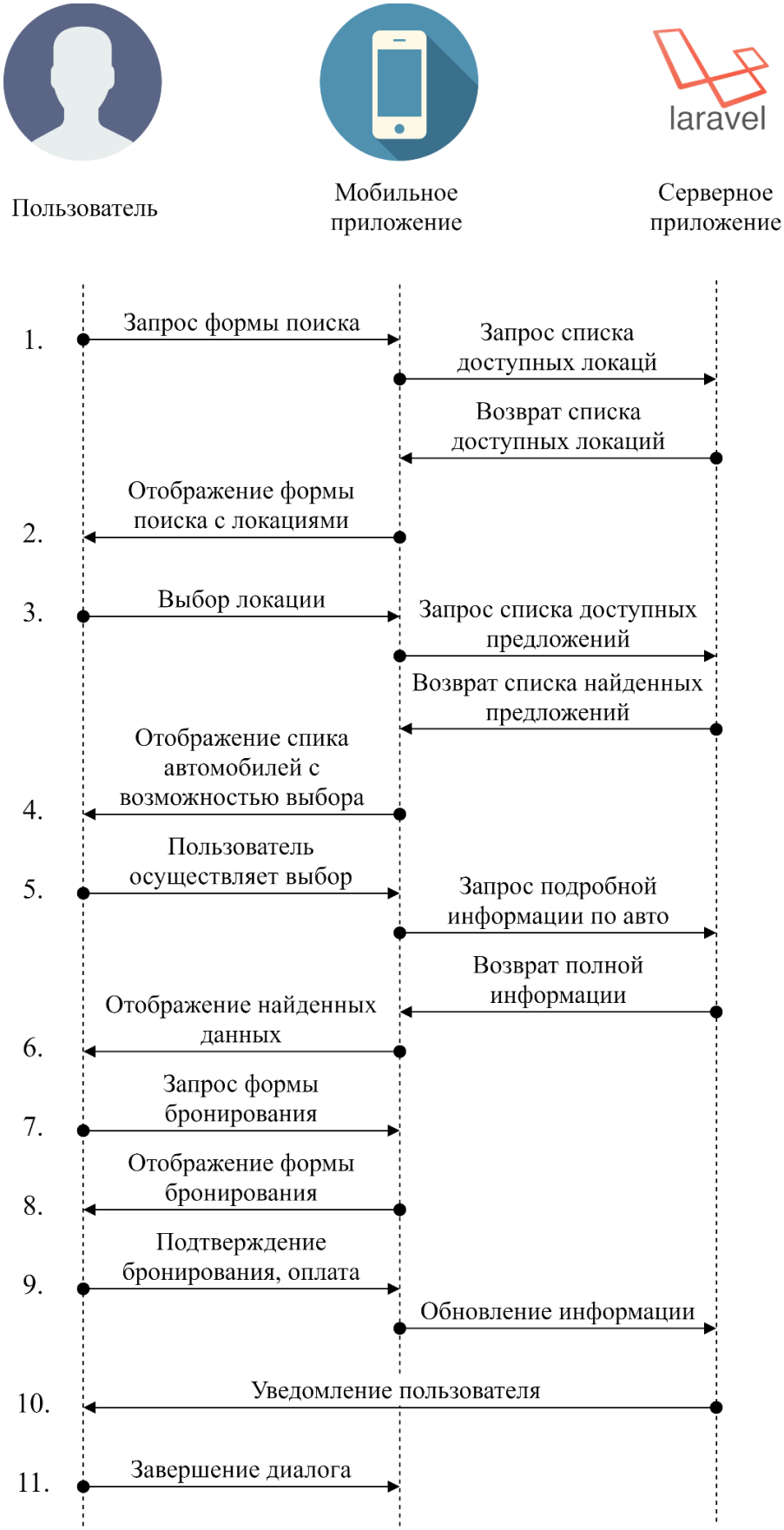


Рис. 12. UML диаграмма главного сценария

Теперь более подробно рассмотрим высокоуровневые диаграммы последовательности работы серверного приложения.

В случае запроса списка локаций:

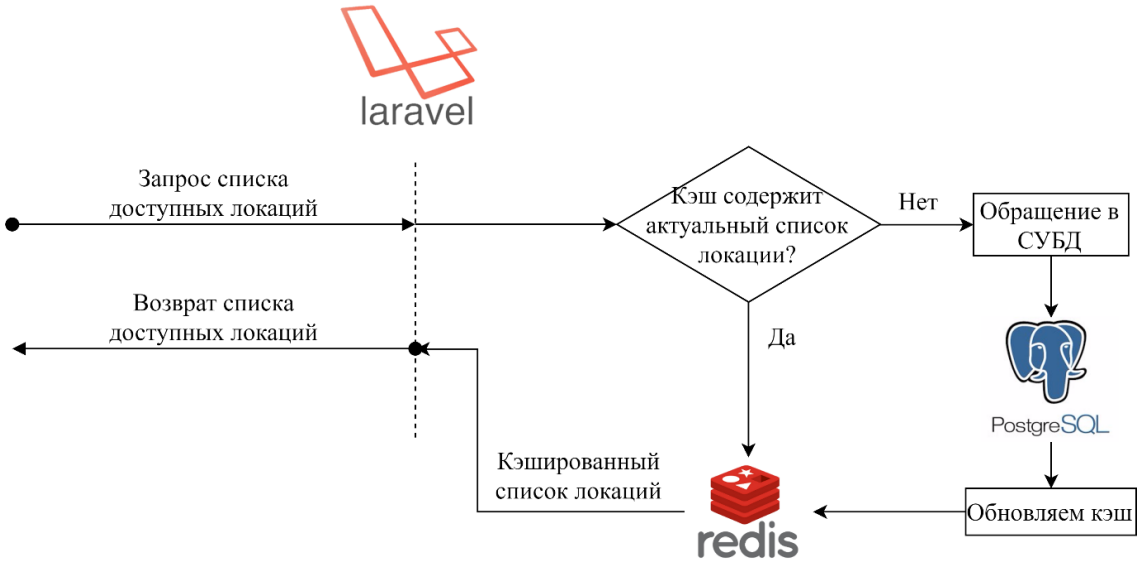


Рис. 13. Алгоритм извлечения списка доступных локаций

В случае запроса списка доступных предложений по прокату автомобилей:

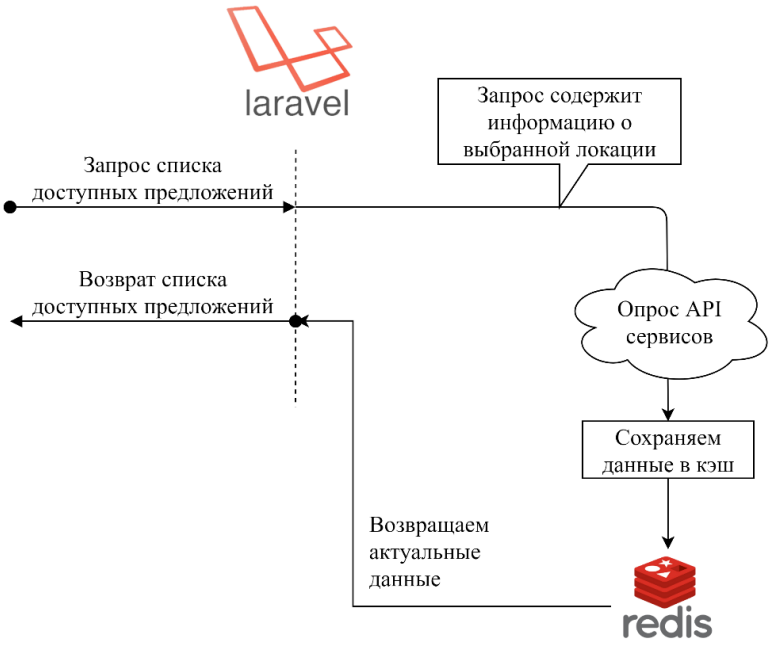


Рис. 14. Алгоритм извлечения списка доступных предложений

5.3. Низкоуровневая реализация

В предыдущем разделе мы описали высокоуровневую последовательность действий, приводящую к выполнению успешного сценария. Она не затрагивает способы программной реализации и архитектурные принципы, заложенные в программе. Однако этот момент стоит детально рассмотреть, поскольку во многом это обеспечивает надежность разрабатываемого ПО и дальнейшую простоту его сопровождения.

Для этого составим низкоуровневую реализацию серверного приложения. Она подразумевает проектирование двух наиболее важных аспектов: описание жизненного цикла обработки HTTP запроса и очередность вызываемых программных компонентов.

5.3.1. Жизненный цикл запроса

Ранее мы определились, что будем строить наше приложение на базе фреймворка Laravel. Этот инструмент из коробки предоставляет большие возможности и мощные компоненты, способные покрыть любую задачу. Но для того, чтобы правильно использовать тот или иной инструмент, необходимо понимать, как он работает. Фреймворк не исключение. Во многом знание архитектуры помогает избежать ошибок при написании кода.

Базовым вопросом является последовательность прохождения HTTP запроса через внутренние компоненты приложения. Источником запроса в нашем случае является мобильное приложение, но совсем необязательно только оно. Рассмотрим типичный сценарий прохождения запроса от клиента – источника до сервера.

Мобильное приложение отправляет HTTP запрос по заданному URL адресу и с конкретным набором данных формата JSON. На серверной стороне работает принимающий HTTP запросы front-endсервер Nginx. Определив, что запрос не относится к статике, Nginx перенаправляет его на один из back-end серверов Apache. Традиционно он настраивается таким образом, что все приходящие на него запросы перенаправляются в единую точку входа

управляющей программы, называемой front-controller. Задача этого контроллера загрузить фреймворк Laravel и передать ему управление.

Далее входящий запрос посылается в HTTP ядро фреймворка. Это ядро служит центральным местом, через которое протекают все запросы. Оно определяет список загрузчиков, которые будут запущены перед выполнением запроса. Эти загрузчики настраивают обработку ошибок, ведение журналов, среду приложения и выполняют другие задачи, которые необходимо выполнить перед непосредственной обработкой запроса. HTTP ядро также определяет список посредников HTTP, через которые должны пройти все запросы, прежде чем будут обработаны приложением. Эти посредники обрабатывают чтение и запись HTTP сессии, определяя, находится ли приложение в режиме обслуживания, проверяют CSRF-последовательность и т.п.

Одно из важнейших действий при загрузке Laravel – загрузка поставщиков услуг для приложения. Они отвечают за начальную загрузку всевозможных компонентов фреймворка: БД, очередь, проверка ввода и маршрутизация. Поставщики услуг – важнейший элемент всего процесса начальной загрузки Laravel, так как они отвечают за загрузку и настройку всех возможностей, необходимых фреймворку.

Когда приложение загружено, и все поставщики услуг зарегистрированы, запрос будет передан роутеру для отправки. Роутер отправит запрос по заданному программистом маршруту, а также запустит посредника, обрабатывающего этот маршрут. В конечном итоге, запрос попадает в контроллер приложения, и его дальнейшая обработка определяется логикой, заложенной в методах этого контроллера.

Таким образом, программисту остается реализовать всего несколько моментов: описать маршрут, класс контроллера – обработчика и класс модели. Все остальное сделает фреймворк.

Ниже приведена схема прохождения HTTP запроса через приложения:

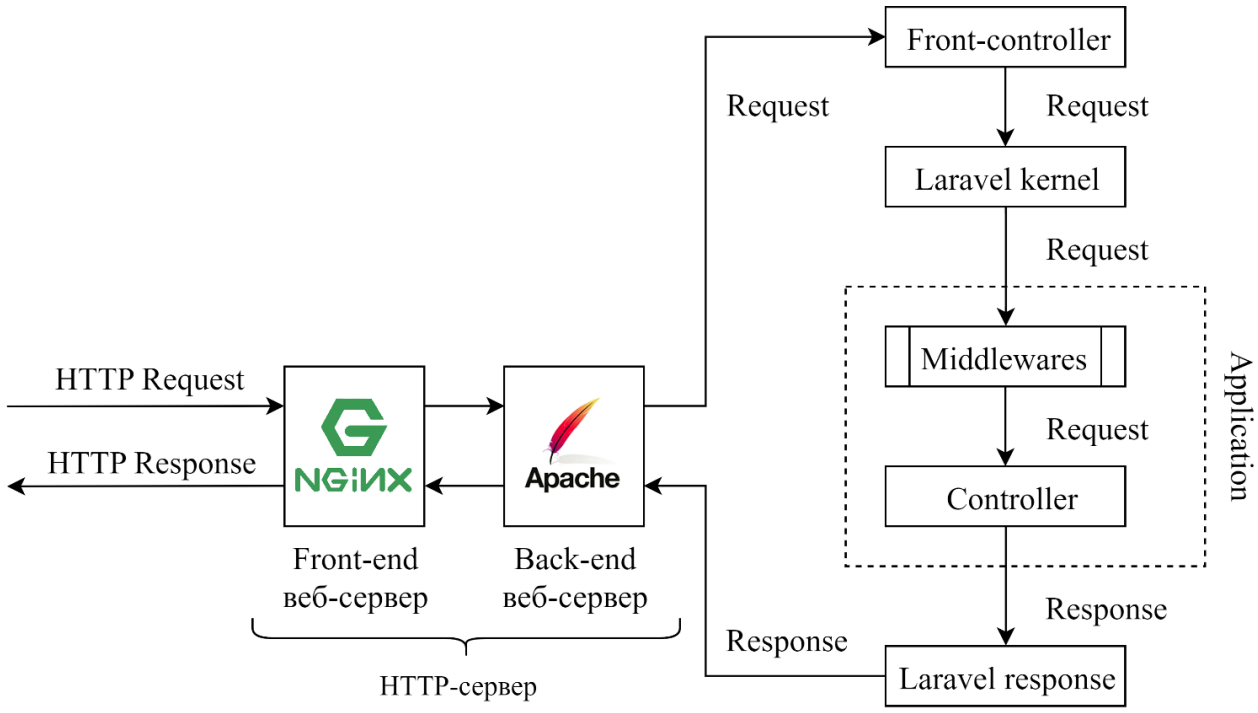


Рис. 15. Жизненный цикл HTTP запроса

В этой схеме любой HTTP запрос, направленный от HTTP сервера, приходит на единую точку входа – front-controller. Как правило, это файл `index.php` в корневой директории проекта. Как я говорил выше, основная задача этого файла загрузить фреймворк. Далее основную задачу по настройке, регистрации модулей и компонентов, а также запуск самого приложения возьмет на себя Laravel, в частности ядро фреймворка – Laravel kernel.

Разработчик занимается исключительно описанием внутренних компонентов приложения (на схеме это application) и настройкой конфигурации фреймворка. Остальной функционал уже готов из коробки.

В целом, логика работы идентична для всех запросов. Конечно, возможности фреймворка достаточно велики. Они позволяют разделять логику на сервисы, организовывать фильтры запросов, middleware слои и прочее, но мы не будем на этом подробно останавливаться.

5.3.2. Инфологическая модель

Для создания инфологической модели воспользуемся описанием предметной области и главного успешного сценария. Выделим ключевые существительные, необходимые для реализации системы, а затем из них логически определим сущности и атрибуты сущностей, необходимые для разработки инфологической модели.

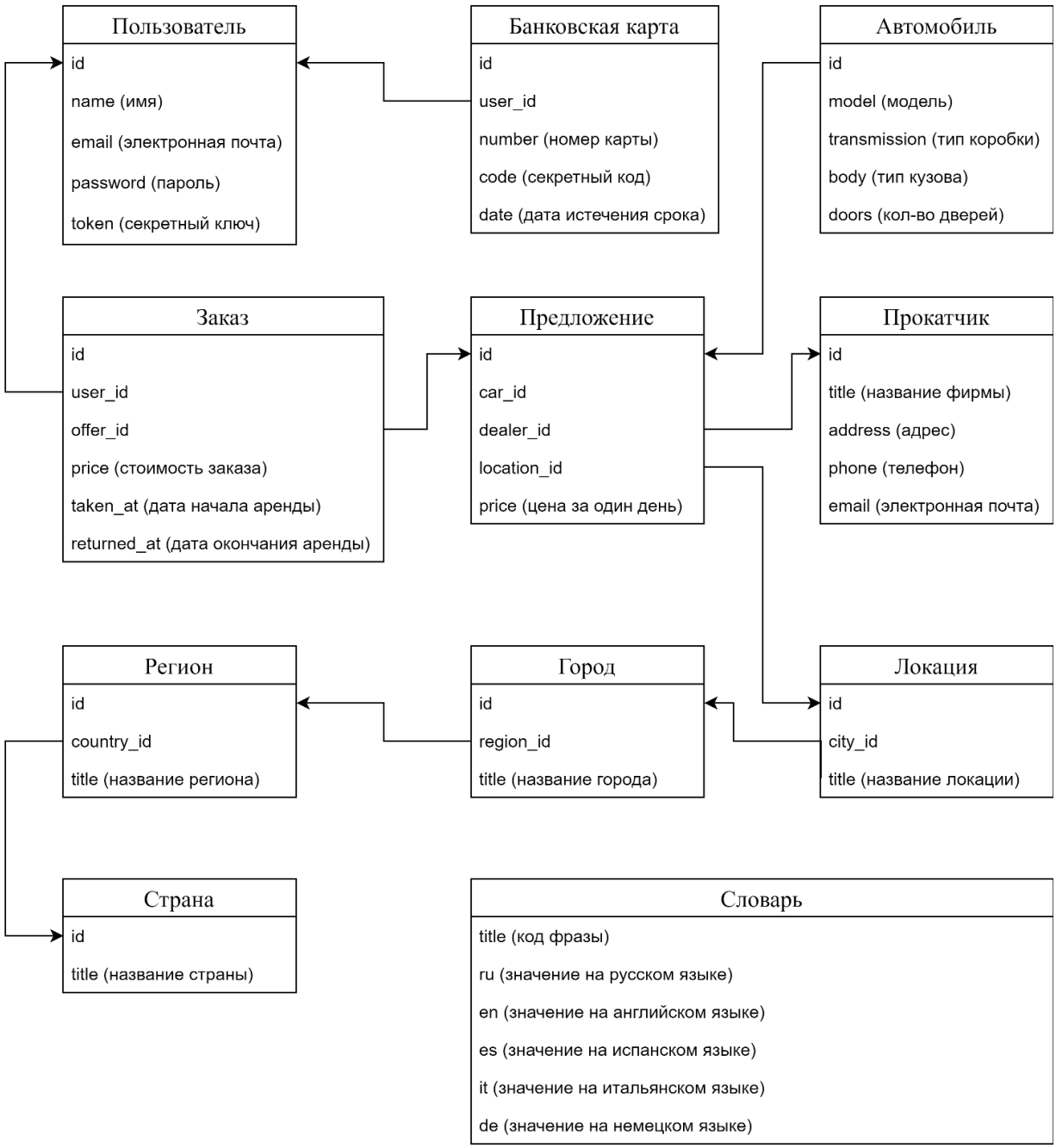


Рис. 16. Инфологическая модель

Составление инфологической модели позволяет должным образом спроектировать базу данных, определив её структуру: состав таблиц, поля и логические связи. Сущность – любой конкретный или абстрактный объект в рассматриваемой предметной области. Это базовый тип информации, который хранится в БД. Понятие типа сущности относится к набору однородных личностей, предметов или событий, выступающих как целый независимый объект. Экземпляр сущности относится, например, к конкретной личности в наборе. Типом сущности может быть студент, а экземпляром – его фамилия: Петров, Сидоров и т. д.

На схеме выше изображена инфологическая модель, иллюстрирующая набор сущностей, атрибутов и связей между ними. В терминах реляционного представления каждая сущность эквивалентна записи в соответствующей таблице, где столбцы хранят значения атрибутов этой сущности. Связь между таблицами строится с помощью вторичных ключей.

Внешний (вторичный) ключ – это один или несколько столбцов в таблице, содержащих ссылку на поле первичного ключа в другой таблице. Внешний ключ определяет способ объединения таблиц. В нашей схеме все первичные ключи соответствуют полю id, а вторичные ключи заканчиваются на постфикс_id. Стрелка на схеме устанавливает ссылку от вторичного ключа к первичному.

Таким образом, схема базы данных в общем виде будет эквивалентна составленной выше инфологической модели с соответствующими таблицами и связями между ними. В то же время, создание первичных и внешних ключей способствует увеличению скорости выборки данных на чтение за счет построения индексов по этим столбцам.

5.3.3. Программные компоненты

Окунемся немного глубже в архитектуру нашего Laravel приложения. Ранее мы говорили о том, что внутренняя логика приложения распределена по контроллерам, которые в свою очередь взаимодействуют с моделями по принципу MVC.Для реализации успешного сценария распишем необходимые объекты, комплекс которых будет составлять наше приложение.

1. Модели

В структуре MVC, которой придерживается фреймворк Laravel, модель – это класс, описывающий одну сущность предметной области и бизнес-логику, связанную с ней. Модель отвечает за представление данных и реализует методы для их обработки. Чтобы организовать удобное лаконичное взаимодействие модели с хранилищем информации (базой данных и кэшем), воспользуемся ORM системой Eloquent.

ORM (Object-Relational Mapping) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». По сути – это прослойка между базой данных и кодом, который пишет программист. Она позволяет использовать созданные в программе объекты модели для обращения в базу данных. Такая система в фреймворке Laravel называется Eloquent. По факту, один класс модели Eloquent описывает структуру одной таблицы, где свойства модели – значения в соответствующих столбцах таблицы. В нашем проекте 10 сущностей, соответственно должно быть 10 моделей. Чтобы воспользоваться этим функционалом фреймворка, необходимо описать каждую сущность в виде отдельного класса.

Например, в нашей бизнес-логике присутствует сущность пользователя, которой соответствует таблица users. Разработчику необходимо описать класс User, который будет наследником объекта базовой модели Eloquent. Впоследствии этот класс можно использовать для запросов в базу. Аналогично и для всех остальных сущностей.

2. Контроллеры

Контроллеры – набор классов, которые определяют основную логику работы приложения. Для реализации структуры API необходимо описать для каждой сущности один контроллер, методы которого будут отвечать за обработку GET, POST, PUT и DELETE запросов. Результатом работы контроллера должен быть объект типа JsonResponse. В общем виде контроллеры похожи, различия заключаются лишь в сущности, которую они обрабатывают.

Возвращаясь к предыдущему примеру с сущностью user, необходимо описать UserController со следующими методами:

- show – для GET запроса,
- create – для POST запроса,
- update – для PUT запроса,
- delete – для DELETE запроса.

Обратите внимание на названия методов. Правила REST архитектуры рекомендуют называть методы контроллера так, чтобы они отображали смысл приходящего запроса. Согласно этим правилам, метод createдолжен создать новый объект класса User, метод delete – удалять, edit – изменять, a show– отображать. Такого рода контроллер имеет особое название – ресурсный контроллер, поскольку реализует все CRUD операции над сущностью.

Ко всему прочему отметим, что каждый метод должен принимать объект класса Requestи id–идентификатор модели в базе данных. Аналогичные принципы справедливы для всех остальных сущностей.

Подведем итог.

Для каждой сущности предметной области, описанной в инфологической модели, необходимо:

1. Создать класс Eloquent модели для организации взаимодействия сущности с базой данных и кэшем.
2. Создать класс ресурсного контроллера, для обработки входящих REST запросов.

В таблице 1 перечислены сущности, с которыми можно взаимодействовать через наше API. Разумеется, все контроллеры, модели и маршруты реализованы в кодовой базе Laravel приложения.

Таблица 1. Список сущностей и их контроллеров

Сущность (модель)	**Метод**	**URL адрес**	**Контроллер**	**Метод**
Пользователь (User)	GET	/api/user/{id}	UserCOntroller	show
	POST	/api/user		create
	PUT	/api/user/{id}		edit
	DELETE	/api/user/{id}		delete
Автомобиль (Car)	GET	/api/car	CarController	index
	GET	/api/car/{id}		show
Предложение (Offer)	GET	/api/offer	OfferController	index
	GET	/api/offer/{id}		show
Заказ (Contract)	GET	/api/contract/{id}	ContractController	show
	POST	/api/contract		create
	PUT	/api/contract/{id}		edit
	DELETE	/api/contract/{id}		delete
Локация (Location)	GET	/api/location	LocationControlle r	index
	GET	/api/location/{id}		show
Город (City)	GET	/api/city	CityController	index
	GET	/api/city/{id}		show
Region (Регион)	GET	/api/region	RegionController	index
	GET	/api/region/{id}		show
Country (Страна)	GET	/api/country	CountryController	index
	GET	/api/country/{id}		show
Bank (Банковский реквизит)	GET	/api/bank/{id}	BankController	show
	POST	/api/bank		create
	PUT	/api/bank/{id}		edit
	DELETE	/api/bank/{id}		delete

5.3.4. Безопасность

На предыдущем этапе я описал доступные URLадреса, которые можно использовать для манипулирования серверными данными через API. Как можно заметить, большинство сущностей доступны только на чтение. Это сделано специально, чтобы ограничить вмешательство в бизнес-логику на уровне API. Так, например, нельзя создать новое предложение по аренде, поскольку эта логика строго определена в серверной программе. Однако таких ограничений недостаточно.

Второй причиной, почему открытое API не есть хорошо, является свободный доступ к конфиденциальной информации. Теоретически любой злоумышленник может получить данные пользователя простым GET запросом. Чтобы ограничить доступ к данным через API воспользуемся системой токенов JWT.

JWT (JSON Web Token) – это JSON объект, который определен в открытом стандарте RFC 7519. Он считается одним из безопасных способов передачи информации между двумя участниками, в нашем случае мобильным приложением и сервером. Для его создания необходимо определить HTTP заголовок с общей информацией по токену, полезные данные и подписи.

Опишем алгоритм, по которому приложение использует JWT для проверки аутентификации пользователя:

1. Пользователь заходит на сервер аутентификации с помощью аутентификационного ключа. Это может быть пара логин/пароль. Сервер аутентификации в данном случае наше серверное приложение.
2. Сервер аутентификации создает JWT токен и отправляет его пользователю.
3. Когда пользователь делает запрос к API приложения, он добавляет к нему полученный ранее JWT токен.
4. Когда пользователь делает API запрос, приложение может проверить по переданному с запросом JWT, является ли пользователь тем, за кого себя выдает.

Очень важно понимать, что использование JWT не скрывает и не маскирует данные. Причина, почему JWT используются – это проверка, что отправленные данные были действительно отправлены авторизованным пользователем. Отметим, что данные внутри JWT закодированы и подписаны, а это не одно и тоже, что зашифрованы. Подписанные данные позволяют получателю проверить аутентификацию их источника.

Опыт использования системы токенов международным сообществом говорит о её высокой надежности. JWT не вызывает проблем при внедрении в приложение и легок в использовании.

Таким образом, каждое вновь скаченное мобильное приложение должно пройти JWT аутентификацию и получить секретный ключ доступа. Тем самым, мы ограничим доступ к нашим данным только реальным клиентам нашего приложения.

На этом шаге проектирование нашего приложения считается завершенным. Следующий этап предполагает программную реализацию описанных выше компонентов системы и их тестирование.

6. Нагрузочное тестирование

Чтобы объективно оценить производительность серверного приложения, необходимо провести нагрузочное тестирование. Оно имитирует одновременную работу некоторого заданного количества пользователей, обращающихся к тестируемому сервису. По окончании тестирования разработчик получает полную информацию о качестве работы сервиса.

Чтобы автоматизировать этот процесс, воспользуемся специальной программой. Apache JMeter – инструмент для нагрузочного тестирования, который способен проводить тесты для JDBC-соединений, FTP, LDAP, SOAP, JMS, POP3, IMAP, HTTP и TCP из коробки и еще множество других протоколов и решений, используя различные плагины.

Выделим две важнейшие метрики тестирования:

- Максимальное количество одновременно обрабатываемых запросов.

В практическом смысле эта характеристика эквивалентна количеству одновременно обрабатываемых пользователей, которое способен выдержать сервер.

- Среднее время отклика. Эта характеристика равна среднему времени, которое тратит сервер на обработку одного запроса.

Все тесты проводились на одном сервере Amazonco следующими характеристиками:

- Размер ОЗУ: 32 Гб;
- ОС: Ubuntu Server 16;
- Процессор: Intel Core i9, 4.4 ГГц, 16 ядер, 32 потока.

Параметры тестирования:

- Цикл запросов: 10 тыс. циклов;
- База данных: 10 млн. записей в каждой таблице;
- Кэш-хранилище: пустое;
- Количество одновременных запросов: изменяемое от 1 до 5 тыс.

В результате тестирования, экспериментально были определены следующие характеристики:

- Максимальное количество запросов в секунду: 2114.
- Среднее время ответа сервера: 0.5с.

- Пиковое значение потребляемой памяти: 22 Гб.

Весьма впечатляющие показатели. По статистике, каждый пользователь мобильного приложения в среднем посылает один запрос в 3 секунды при общем количестве запросов 10 штук для не real-time приложений.

Если полагаться на эти данные, то нашей производительности хватит с запасом. Стоит учитывать тот факт, что тестирование проводилось на одном сервере, где одновременно располагалась база данных, кэш-хранилище и Laravel-приложение. В реальной жизни эти сервисы будут находиться на трех разных машинах, что позволит каждому из них эффективнее выполнять свою задачу за счет большего потребления серверных ресурсов.

Подытоживая, можно сделать следующий вывод: спроектированная серверная архитектура мобильного приложения является крайне производительным решением и удовлетворяет всем требованиям технического задания.

7. Пример работы приложения

Ниже приведены скриншоты мобильного приложения, работающего в демо-режиме.

Открывая мобильное приложение, пользователь сразу же видит страницу, на которой предлагается выбрать место аренды. Сначала необходимо выбрать страну, затем город и место проката из предложенного списка:

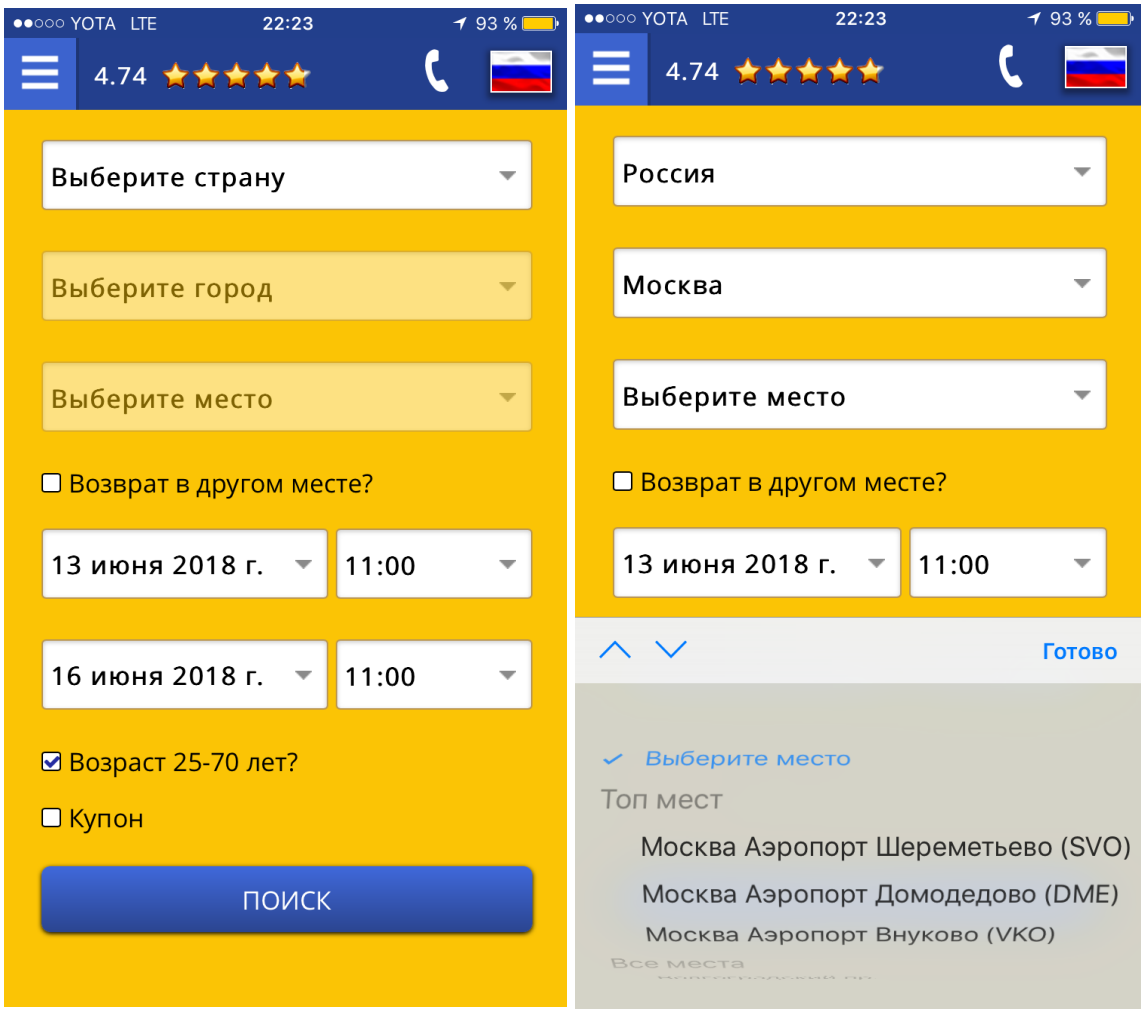


Рис. 17. Интерфейс приложения: выбор локации

После того, как пользователь заполнил все поля, он может нажать на кнопку поиска. В этот момент мобильное приложение отправит HTTP запрос на сервер, по которому получит список доступных предложений.

На втором шаге пользователю предстоит выбрать одно предложение из списка доступных. В данном случае было найдено 33 автомобиля. Например, я хотел бы подыскать пятиместный седан представительского класса. Пусть это будет BMW 3 серии, и, к счастью такой автомобиль нашелся:

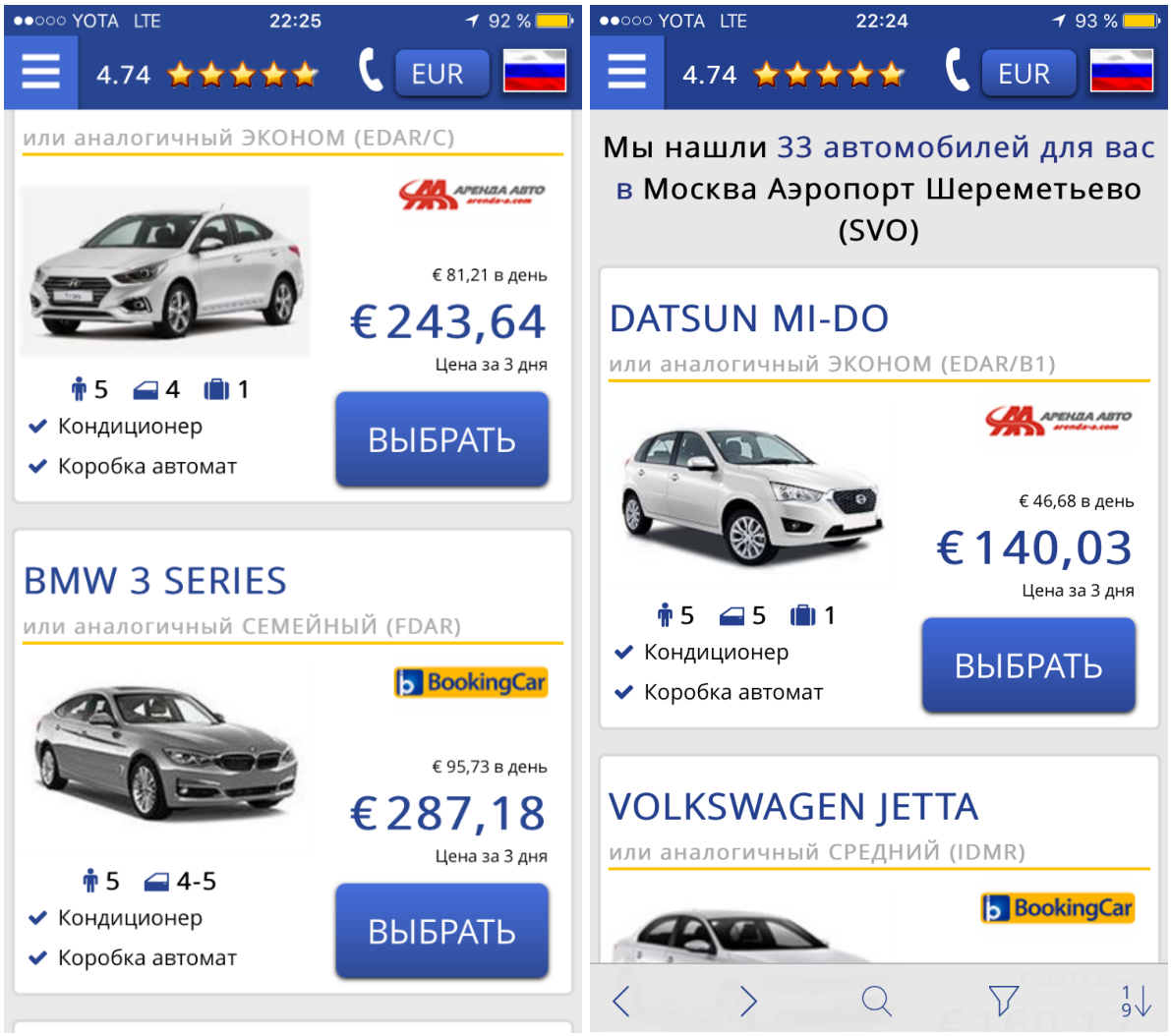


Рис. 18. Интерфейс приложения: выбор предложения

Найдя подходящее предложение, пользователь нажимает кнопку «Выбрать». В этот момент мобильное приложение запоминает выбор пользователя и отправляет HTTP запрос на сервер для получения дополнительной информации по выбранному автомобилю.

На третьем шаге пользователь получает более детальную информацию по выбранному предложению. Так же есть возможность выбрать дополнительные опции за дополнительную плату:

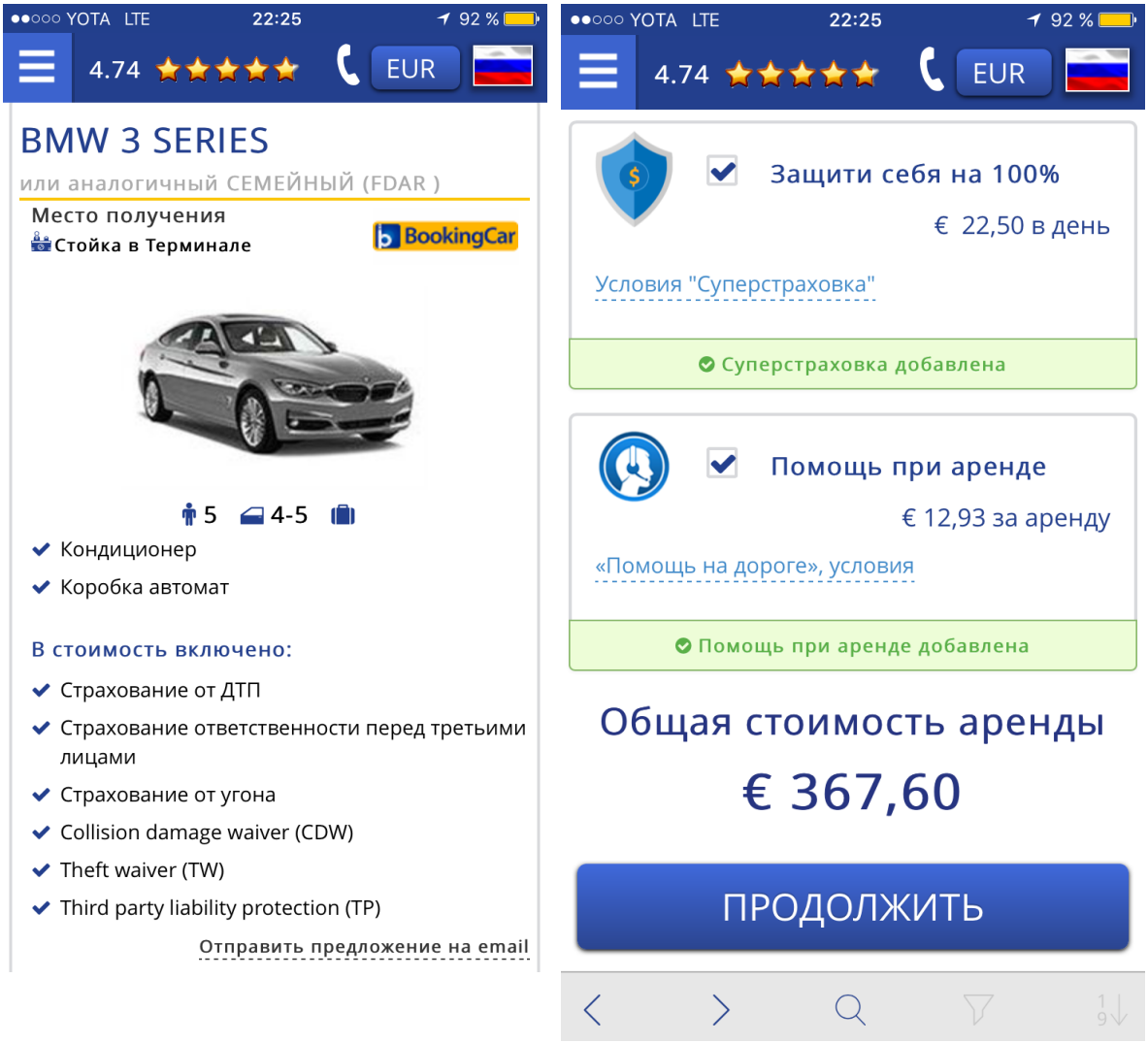


Рис. 19. Интерфейс приложения: подтверждение выбора

Чтобы выбрать дополнительную опцию, пользователю достаточно поставить галочку в соответствующем пункте выбора. Для перехода на следующий шаг следует нажать кнопку «Продолжить».

На последнем шаге пользователь получает полную информацию о заказе. Проверив все детали, остается заполнить форму с банковскими реквизитами и нажать кнопку «Забронировать». В этот момент мобильное приложение отправит HTTP запрос для проверки статуса заказа и возможности провести транзакцию.

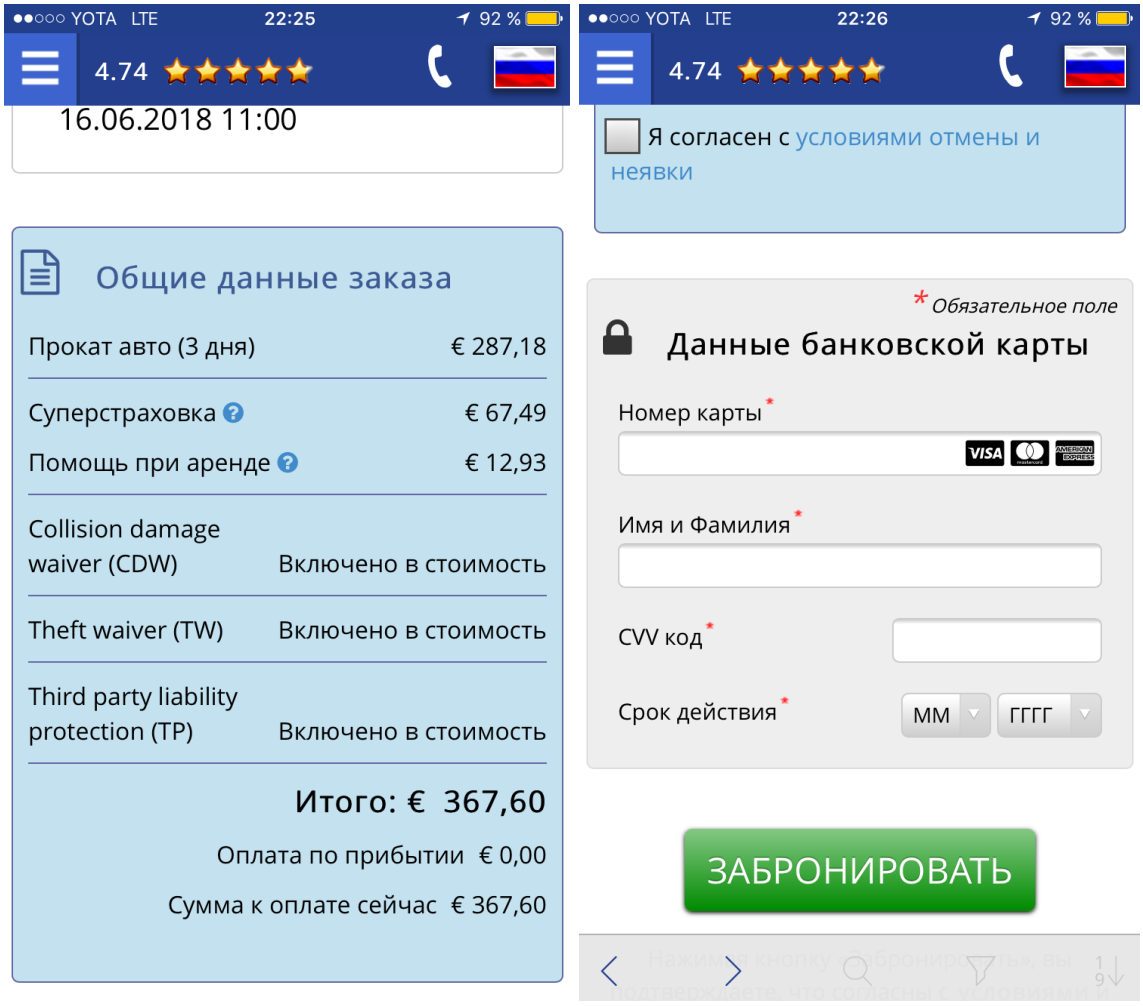


Рис. 20. Интерфейс приложения: бронирование автомобиля

Если операция перевода средств пройдет успешно, пользователь получит СМС (или email) оповещение об успешном резервировании автомобиля с подробной информацией о заказе. На этом успешный сценарий считается выполненным.

Для того, чтобы облегчить пользовательский поиск, была предусмотрена удобная система фильтрации предложений. Есть возможность выбрать класс автомобиля, необходимую комплектацию, коробку автомат и прочие атрибуты.

Другой важной особенностью этого приложения является мультиязычность. Пользователь может в любой момент сменить язык интерфейса без каких-либо проблем, например, на немецкий.

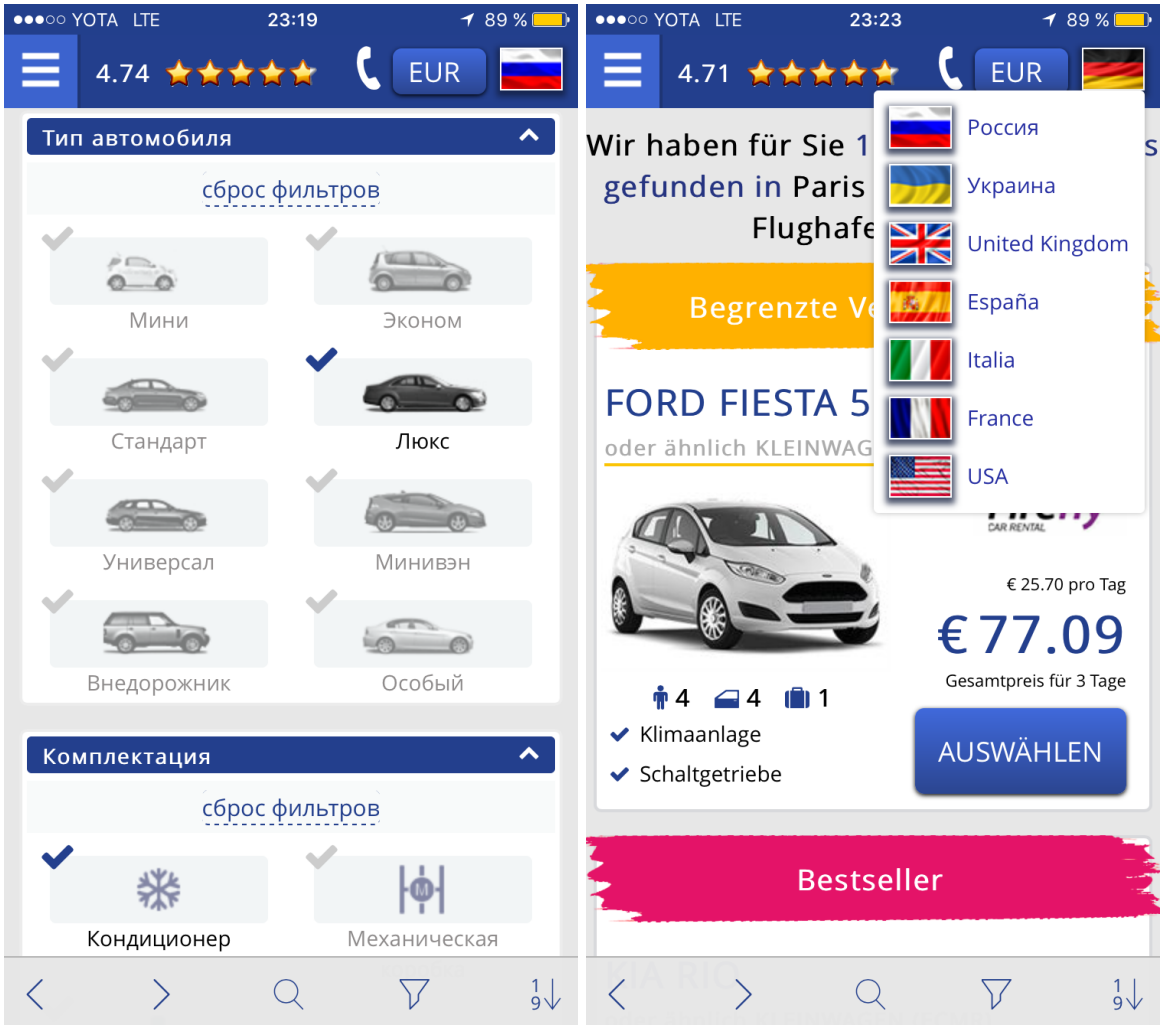


Рис. 21. Интерфейс приложения: список фильтров

При этом, в случае внезапного разрыва соединения с сетью, пользователь не потеряет данные текущей сессии – мобильное приложение успешно все восстановит. Как можно заметить, интерфейс мобильного приложения интуитивно понятен, ненавязчив и вполне комфортен. Цель была достигнута.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта была разработана клиент-серверная архитектура мобильного приложения по прокату и аренде автомобилей.

Была изучена предметная область, составлена инфологическая модель и определен оптимальный стек технологий клиентской и серверной архитектуры для реализации требований технического задания.

В ходе проектирования серверной части были выбраны следующие базовые технологии:

- Язык программирования: PHP 7;
- Фреймворк: Laravel;
- СУБД: PostgreSQL;
- Кэш-хранилище: Redis;
- Front-end HTTP сервер: Nginx;
- Back-end HTTP сервер: Apache;
- Операционная система: Ubuntu Linux.

Вышеописанный стек позволяет достичь:

- Серверную отказоустойчивость;
- Быструю обработку данных;
- Надежное хранение информации;
- Горизонтальное масштабирование;
- Интеграцию со сторонними сервисами;
- Быструю разработку.

Результаты проведенного нагрузочного тестирования свидетельствуют об успешном выполнении требований технического задания.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Андрей Робачевский. Интернет изнутри: Экосистема глобальной сети. Изд.: Альпина Паблишер. 224 стр. 2017 год.
2. Димитрий Айвалиотис. Администрирование сервера NGINX. Подробное руководство по настройке NGINX в любой ситуации, с многочисленными примерами и справочными таблицами для всех директив. Изд. ДМК Пресс. 288 стр. 2017 год.
3. Дмитрий Котеров, Игорь Симдянов. PHP 7. БХВ-Петербург. 413 стр. 2017 год.
4. Джоэл Грас. Data Science. Наука о данных с нуля. Изд.: БХВ-Петербург. 336 стр. 2018 год.
5. Крэг Ларман. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и интерактивную разработку. Изд. Дом Вильясм. М.-С.-П.-Киев., 2013 год.
6. Мартин Фаулер. UML Основы. Символ. С.-П., 2013 год.
7. Мэтт Зандстра. PHP. Объекты, шаблоны и методики программирования. Изд. Вильямс. 576стр. 2016 год.
8. Саймон Ригс, Ханну Кросинг. Администрирование Postgre SQL 9. Книга рецептов. ДМК Пресс., 364стр. 2015 год.
9. Сергей Тарасов. СУБД для программиста. Базы данных изнутри. Изд. Соломон. 320 стр. 2015 год.
10. Эндрю Таненбаум, Дэвид Уэзеролл. Компьютерные сети. Изд.: Питер. 960 стр. 2016 год.
11. Mark Masse. REST API Design Rulebook. O`Really. 115стр. 2011 год.
12. Martin Bean. Laravel 5 Essentials. Изд. ООО “Книга по требованию”. 2017 год.
13. Matt Stauffer. Laravel: Up and Running. Изд.: O`Really. 2018 год.
14. Usama Dar. Nginx Module Extension. Изд.: ООО “Книга по требованию”. 106 стр. 2017 год.

ПРИЛОЖЕНИЕ А

В графическую часть выпускной квалификационной работы (5 листов формата А4) входят:

- Инфологическая модель (лист 1).
- Схема жизненного цикла запроса (лист 2).
- Архитектура серверного приложения (лист 3).
- UML-диаграмма главного сценария (лист 4).
- Блок-схема алгоритмов обработки запросов (лист 5).