



Урок 1

Начало работы над проектом

Анализ технического задания. Создание нового проекта. Gitflow.

[Прием проекта](#)

[Техническое задание \(ТЗ\)](#)

[Анализ технического задания](#)

[Анализ макета](#)

[Анализ серверного API](#)

[Результаты анализа ТЗ](#)

[Создание проекта GBShop](#)

[Создание Git-репозитория. Первый commit](#)

[Gitflow](#)

[Организация файловой структуры проекта](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Прием проекта

Техническое задание (ТЗ)

Техническое задание (ТЗ) — это текстовый документ, в котором в мельчайших деталях описываются все технические подробности разработки будущего проекта. Хорошее ТЗ должно состоять из множества пунктов, описывающих различные аспекты проекта. Помните: чем глубже, подробнее прописаны пункты ТЗ, тем стабильнее будет разработка и успешнее — проект в целом.

Так ли необходимо ТЗ? Представим ситуацию: мы — команда разработчиков, и заказчик предлагает нам сделать мобильное приложение «ну, как такое-то (и приводит пример продукта)». После такой формулировки мы не знаем, что конкретно делать. Начинаются вопросы. Каковы основные действия пользователя? Какие версии операционной системы будут поддерживаться в мобильном приложении? Нужна ли поддержка альбомной ориентации? Какие анимации будут предусмотрены? И так далее. Поэтому все, что заказчик нам не рассказывает и держит «в голове», нужно расписать и зафиксировать в ТЗ. Иначе можем получить множество разногласий, которые приведут к:

- отрицательным отзывам;
- сорванными срокам;
- впустую потраченному времени, силам и нервам — как с нашей стороны, так и со стороны заказчика.

Анализ технического задания

Техническое задание должно быть написано на понятном языке.

Бывает, ТЗ приходит от заказчика, который хочет проявить свою техническую подкованность. Он формулирует в задании такие конструкции, что разработчики не могут понять, чего же от них хотят. Такие пункты надо прояснять и переформулировать. Также нужно избавляться от расплывчатых фраз: «Обычная страница/экран», «простая анимация», «форма, как у нас на сайте», «уведомление пользователей». ТЗ должно быть точным и однозначным.

Необходимо внимательно ознакомиться с ТЗ. Проверить, хватает ли данных для реализации проекта, все ли нюансы учтены. Если в ТЗ есть отсылки к другим приложениям — обязательно скачать и ознакомиться с ними.

Рассмотрим пример плохого ТЗ. Заказчик представил техзадание на разработку небольшого приложения:

1. Наименование работ.

Разработка iOS-приложения хранения паролей.

2. Общие сведения.

Приложение сохраняет пароли по имени ресурса. Отображает сохраненные пароли в списке. При выборе ресурса показывает сохраненный пароль.

3. Главный экран со списком имен ресурсов.

4. Окно добавления пароля.

5. Окно показа сохраненного пароля.

6. Требования к дальнейшей поддержке продукта.

В качестве последующей технической поддержки Подрядчик гарантирует качественную работу мобильного и серверного приложения на протяжении 12 месяцев. В случае обнаружения каких-либо технических сбоев, возникших по вине Подрядчика, он

обязуется устранить их в кратчайшие сроки.

7. Гарантийный срок.

Гарантийный срок составляет 12 (двенадцать) месяцев с момента подписания акта сдачи-приемки выполненных работ.

8. Приемка работ.

Приемка результатов выполненных работ осуществляется Заказчиком с участием представителя Подрядчика.

Казалось бы, всего три экрана. Но сколько осталось «за кадром». Нет ни одного макета. Как должен отображаться список имен ресурсов? Это табличное представление или коллекция? Нужен ли логотип ресурса? Нужны ли анимации перехода? Окна показа и добавления — модальные окна, или заказчик имел в виду обычные экраны? Какие типы устройств поддерживаются? Какие версии iOS? Нужна ли поддержка альбомной ориентации? И так далее.

Анализ макета

При разработке мобильного приложения у нас должен быть его дизайн, реализованный под все поддерживаемые типы устройств и ориентаций. Если по каким-то причинам заказчик не указал эти типы и ориентации, этот факт обязательно нужно вынести на обсуждение. Учтите, что на выполнение этих требований может уйти много времени, и срок разработки может увеличиться чуть ли не вдвое.

У нас должны быть макеты всех экранов приложения. Каждый макет нужно проверить на соответствие гайдлайнам (правилам размещения графических элементов). Обязательно посмотреть на размеры интерактивных элементов. Например, кнопка должна быть не меньше 40pt на 40pt — иначе ее будет сложно нажимать. Обратите внимание на присутствие нестандартных элементов, на повороты экрана... Все эти факторы добавляют дополнительное, а порой и очень ощутимое, время разработки.

В макетах должно присутствовать описание всех переходов. Требование к ним — быть логичными, чтобы пользователь мог легко в них ориентироваться. Необходимо проанализировать сложные анимации и обсудить все «узкие» моменты в них, чтобы убедиться, что мы говорим с заказчиком на одном языке.

Обратите внимание на выбранные заказчиком шрифты. На нестандартные iOS может выдавать предупреждения и требовать подключить другой шрифт. Приглядитесь к предоставленным наборам иконок. Требуйте иконки в формате pdf, чтобы в будущем не было проблем с поддержкой новых устройств с более высоким разрешением экрана.

Анализ серверного API

Просматриваем все предоставленное описание запросов и ответов от удаленного сервера. Обращаем внимание на форматы, типы возвращаемых данных от сервера.

Например, если при запросе стоимости товара с сервера сумма приходит в виде строки и далее участвует в арифметических операциях, данное действие приведет к ошибке.

```
{
  "id": 123,
  "name": "Ноутбук",
  "price": ",0",
  ...
}
```

Следующий ответ на запрос информации о пользователе, содержащий пол, может быть неоднозначным и при отсутствии документации привести к ошибке.

```
{
  "name": "Alex",
  "email" : "some@some.ru",
  "gender": true,
  ...
}
```

Задаем себе вопрос о полноте серверного API. Все ли данные, необходимые мобильному приложению в целом, возвращаются нам от сервера? Обращаем внимание на возможные ошибки и недочеты в серверном API. Если не проанализировать API и полностью положиться на серверных разработчиков, можно столкнуться с множеством «подводных камней». Например, серверное API для web-страниц не всегда в полной мере подходит для мобильного приложения. Пример — размер отображаемой информации, который для мобильного приложения может биться на несколько экранов. Это может повлиять на количество запросов к серверу, которые будут уникальны только для мобильного приложения, и так далее.

Результаты анализа ТЗ

На основе проведенного анализа ТЗ мы можем определиться с основными сценариями использования приложения. Будут понятны основные действия пользователя и переходы. Теперь мы приступаем непосредственно к архитектуре проекта. Выбираем паттерн проектирования (MVC, MVP или иной). Определяемся с уровнем гибкости проекта. Есть термин «**White label**», обозначающий концепцию, которая предусматривает создание продуктов для использования несколькими компаниями. То есть в приложении мы можем предусмотреть загрузку логотипов и описаний компаний с сервера, для каждой по-разному отображать набор графических компонентов, экраны и т.д. В подобном случае нужна большая гибкость приложения.

На этом этапе переходят к составлению плана работы над проектом. Обозначаются задачи, спринты.

Создание проекта GBShop

Приступим к созданию скелета проекта **GBShop**. Сначала откроем **Xcode** и создадим новое приложение по шаблону **Single View Application**. Зададим имя проекта — **GBShop**. Проставим галочки использования модульных (**Include Unit Tests**) и графических тестов (**Include UI Tests**). На следующих уроках мы подробно осветим тему тестирования.

На протяжении всего курса будем использовать Git.

Добавить в проект файл `.gitignore`

Зачастую в проектах находятся файлы, которые мы не хотим добавлять в репозиторий. К ним обычно относятся автоматически генерируемые файлы (например, **Xcode**), служебные файлы используемых инструментов и так далее.

Игнорировать подобные файлы можно, если создать файл **.gitignore**. Добавляя в него правила игнорирования, мы подсказываем Git, что файлы, соответствующие этим правилам, не надо добавлять в репозиторий.

Если подобные файлы уже были добавлены в Git-репозиторий, **.gitignore** не поможет. Придется самим удалять их и добавлять новые правила в **.gitignore**, чтобы не допустить повторного попадания этих файлов в Git-репозиторий. На этапе добавления проекта в Git можно полностью удалить репозиторий, воспользовавшись командой **rm -rf .git/**, и потом заново добавить проект в Git.

В идеале вы должны сами реализовать правила в **.gitignore**, но можно воспользоваться наработками **github.com**.

```
# Xcode
#
# gitignore contributors: remember to update Global/Xcode.gitignore,
Objective-C.gitignore & Swift.gitignore

## Build generated
build/
DerivedData/

## Various settings
*.pbxuser
!default.pbxuser
*.modelv3
!default.modelv3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3
xcuserdata/

## Other
*.moved-aside
*.xccheckout
*.xcscmblueprint
```

```

## Obj-C/Swift specific
*.hmap
*.ipa
*.dSYM.zip
*.dSYM

## Playgrounds
timeline.xctimeline
playground.xcworkspace

# Swift Package Manager
#
# Add this line if you want to avoid checking in source code from Swift Package
Manager dependencies.
# Packages/
# Package.pins
# Package.resolved
.build/

# CocoaPods
#
# We recommend against adding the Pods directory to your .gitignore. However
# you should judge for yourself, the pros and cons are mentioned at:
#
https://guides.cocoapods.org/using/using-cocoapods.html#should-i-check-the-pod-s-directory-into-source-control
#
Pods/

# Carthage
#
# Add this line if you want to avoid checking in source code from Carthage
dependencies.
# Carthage/Checkouts

Carthage/Build

# fastlane
#
# It is recommended to not store the screenshots in the git repo. Instead, use
fastlane to re-generate the
# screenshots whenever they are needed.
# For more information about the recommended setup visit:
# https://docs.fastlane.tools/best-practices/source-control/#source-control

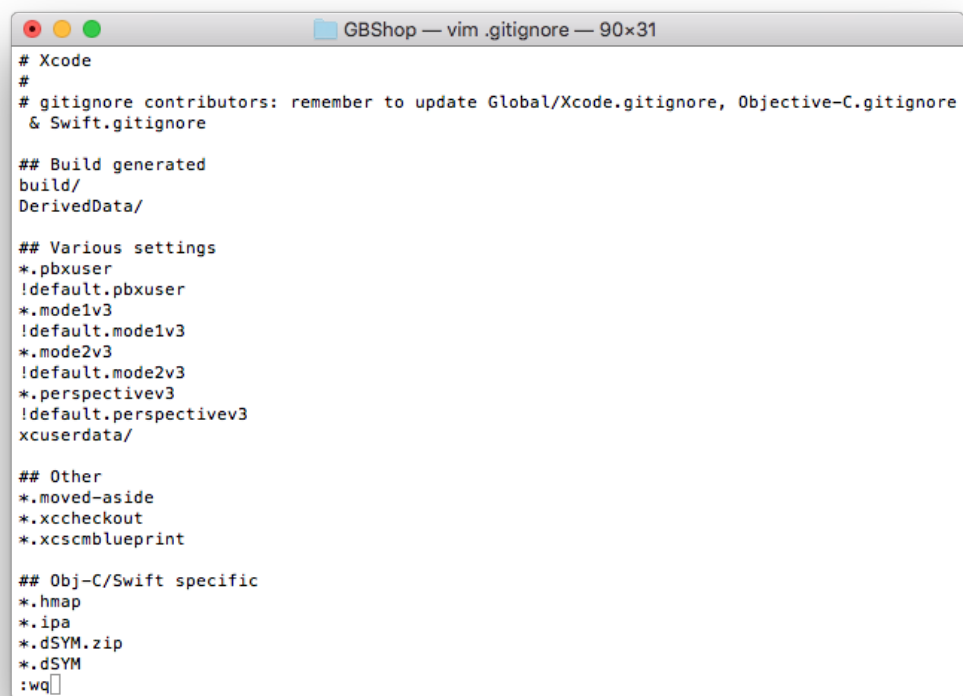
fastlane/report.xml
fastlane/Preview.html
fastlane/screenshots
fastlane/test_output

```


В этом примере предлагается игнорировать автоматически сгенерированные файлы **Xcode**, систем **Swift Package Manager**, **CocoaPods**, **Carthage**, **fastlane**.

Добавим файл **.gitignore** в наш проект. Откроем терминал (найти его можно в приложениях). Перейдем в корень проекта при помощи команды **cd**.

С помощью текстового редактора **vim** создадим файл **.gitignore** и вставим приведенные выше правила.



```
# Xcode
#
# gitignore contributors: remember to update Global/Xcode.gitignore, Objective-C.gitignore
# & Swift.gitignore

## Build generated
build/
DerivedData/

## Various settings
*.pbxuser
!default.pbxuser
*.mode1v3
!default.mode1v3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3
xcuserdata/

## Other
*.moved-aside
*.xccheckout
*.xcscmblueprint

## Obj-C/Swift specific
*.hmap
*.ipa
*.dSYM.zip
*.dSYM
:wq
```

Создание Git-репозитория. Первый commit

В терминале выполняем команду **git init** — создаем репозиторий в существующем каталоге.

Команда **git init** создает в текущем каталоге новый подкаталог с именем **.git**, содержащий все необходимые файлы для основы Git-репозитория. На этом этапе проект еще не находится под контролем Git. Добавляем все файлы (кроме тех, что попадают под правила **.gitignore**) с помощью команды **git add -A**.

Выполняем первый инициализирующий **commit** с помощью команды **git commit -m "init"** — добавление изменений в репозиторий. В данной команде после параметра **-m** задается описание добавляемых изменений. Всегда давайте осмысленные комментарии к **commit**, чтобы можно было оперативно найти изменения.

```
mac2:~ oivanov$ cd Documents/GBShop/
mac2:GBShop oivanov$ vim .gitignore
mac2:GBShop oivanov$ git init
Initialized empty Git repository in /Users/oivanov/Documents/GBShop/.git/
mac2:GBShop oivanov$ git add -A
warning: unable to access '/Users/oivanov/.config/git/attributes': Permission denied
mac2:GBShop oivanov$ git commit -m "init"
[master (root-commit) 7c53fc3] init
  Committer: OIvanov <oivanov@mac2.mcb.ru>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

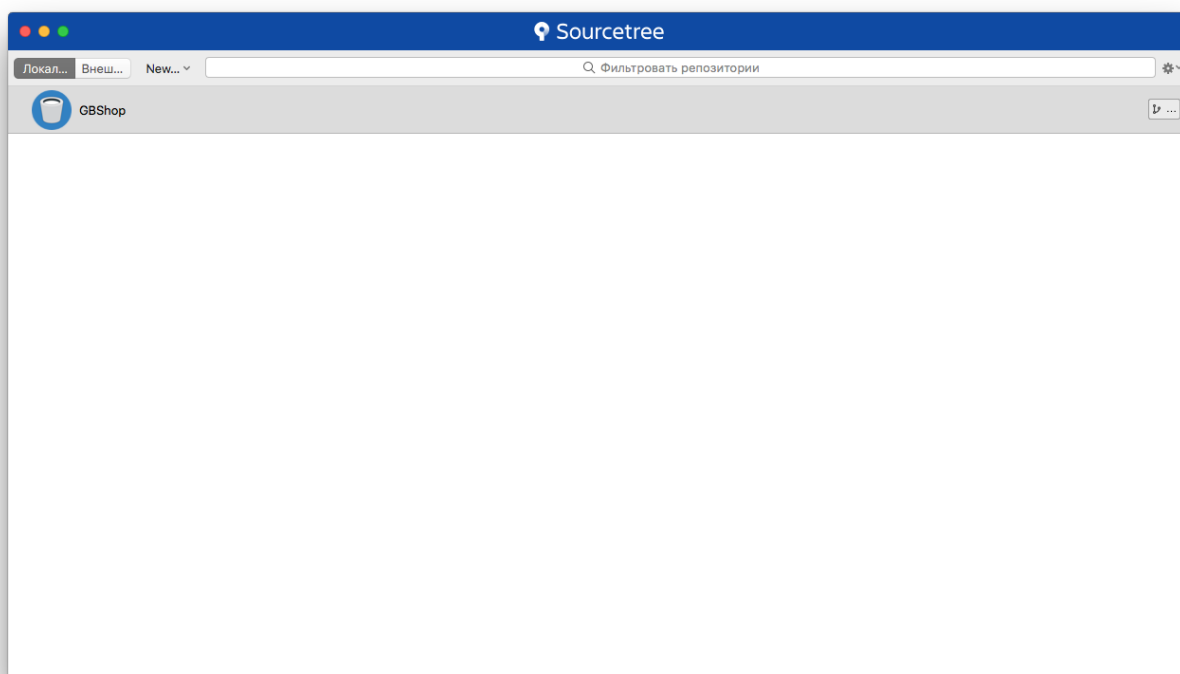
    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

warning: unable to access '/Users/oivanov/.config/git/attributes': Permission denied
13 files changed, 1012 insertions(+)
create mode 100644 .gitignore
create mode 100644 GBShop.xcodeproj/project.pbxproj
create mode 100644 GBShop.xcodeproj/project.xcworkspace/contents.xcworkspacedata
create mode 100644 GBShop/AppDelegate.swift
create mode 100644 GBShop/Assets.xcassets/AppIcon.appiconset/Contents.json
create mode 100644 GBShop/Base.lproj/LaunchScreen.storyboard
create mode 100644 GBShop/Base.lproj/Main.storyboard
create mode 100644 GBShop/Info.plist
create mode 100644 GBShop/ViewController.swift
create mode 100644 GBShopTests/GBShopTests.swift
create mode 100644 GBShopTests/Info.plist
create mode 100644 GBShopUITests/GBShopUITests.swift
create mode 100644 GBShopUITests/Info.plist
mac2:GBShop oivanov$
```

Для визуализации Git-репозитория и управления ими рекомендуется использовать [sourcetree](https://source-tree.com/). Это бесплатный Git-клиент с простым и понятным графическим интерфейсом. Откроем **sourcetree**, добавим туда наш репозиторий.



Gitflow

Рассмотрим работу с ветками на примере.

Допустим, мы находимся в основной ветке и хотим дописать функционал, основываясь на коде из нее. Для этого нужно написать команду для создания дополнительной ветки и провести всю работу там. А когда закончим, внесем реализованные изменения обратно в основную ветку (слить ветки).

Работать с ветвлением в Git просто, и у многих людей своеобразный подход к управлению ветками. Поэтому понадобилось соглашение о работе с ними, призванное стандартизировать модель веток.

Таким соглашением выступил **Gitflow**. Этот инструмент значительно упрощает работу для всех участников репозитория и позволяет вести понятную модель веток (**master** + **develop**), поддерживать версионирование релизов.

Gitflow предлагает организацию и поддержку определенного набора веток:

- **master** — главная ветвь для production-релизов;
- **develop** — главная ветвь разработки;
- **feature/*** — ветви с новыми функциями, которые потенциально будут слиты в **develop**;
- **hotfix/*** — ветви для исправлений/доработок в релизах, будут слиты в **master**;
- **bugfix/*** — ветки для исправлений ошибок.

Но **Gitflow** — всего лишь рекомендация, и в различных компаниях могут отсутствовать определенные ветви.

Откроем **sourcetree**. Выберем пункт **Initialize Repository**. И **sourcetree** предложит ввести названия всех веток в соответствии с **Gitflow**:

Организация файловой структуры проекта

Все разработчики хотят одного — чистоты кода и возможности легко добавлять новый. Проект будет разрастаться со временем, будут появляться новые классы, файлы... И с каждым разом навигация и вставка нового кода будет усложняться поиском места для него. Поэтому грамотная организация файловой структуры проекта — очень важный этап. Придерживайтесь одинаковой структуры файлов во всех проектах.

Приведенная ниже организация файловой структуры проекта также носит рекомендательный характер, и в различных проектах и компаниях она может отличаться. Но суть останется той же.

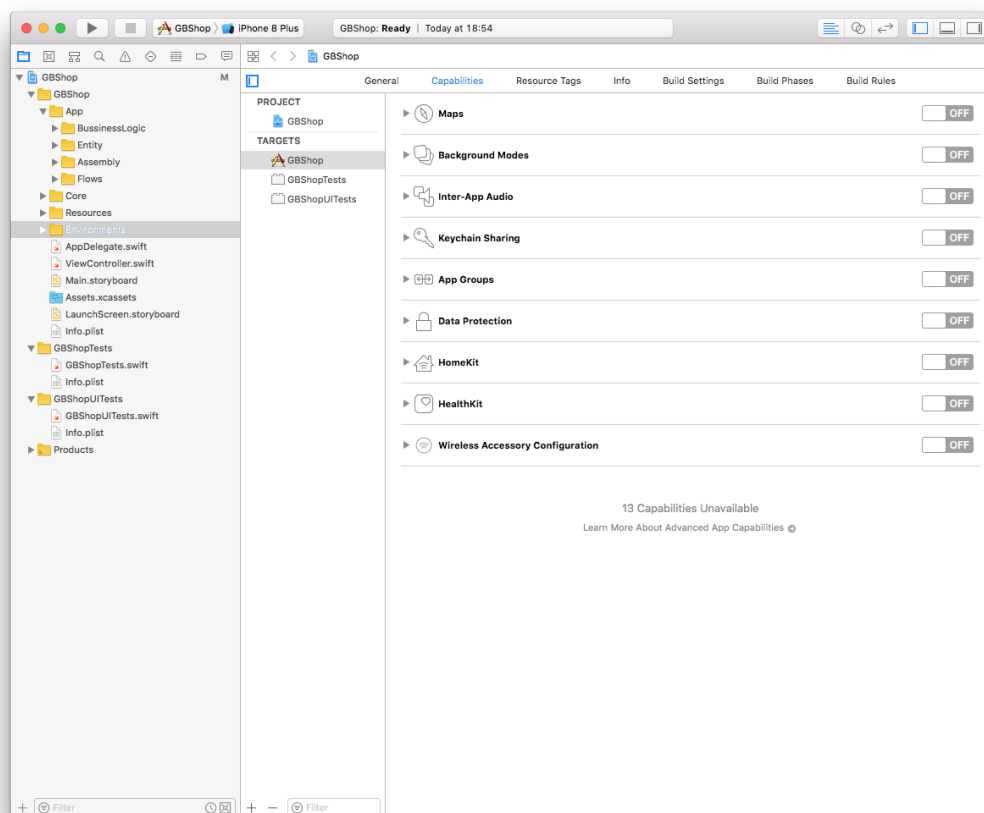
Структура проекта:

```
App
  BusinessLogic
  Entity
  Assembly
  Flows
Core
Resources
Environments
```

- **App** — бизнес-логика приложения, модели, экраны;
- **BusinessLogic** — здесь находятся все сервисы, работы с сетью, с хранилищами данных, с UserDefaults — все то, что работает с данными;
- **Entity** — здесь располагаются классы моделей данных. Например, классы информации о покупателе, о товаре;
- **Assembly** — сборщики приложения (об этом поговорим подробнее на уроках);

- **Flows** — здесь располагаются классы экранов приложения, классы ячеек, xib-файлы, storyboard со связанными контроллерами. Обратите внимание: классы экранов должны быть максимально «чистыми», то есть содержать минимум бизнес-логики. Например, логика работы с UserDefaults должна переехать в отдельный класс и располагаться в BusinessLogic;
- **Core** — здесь находятся различные helpers, вспомогательные инструменты, расширения. Например, классы расширений для форм с использованием атрибутов IBDesignable/IBInspectable, которые позволяют отображать произвольные, нестандартные view с кастомными полями класса в Interface Builder Xcode;
- **Resources** — файлы-картинки, файлы локализации, справочные файлы, media- и audio-файлы;
- **Environments** — файлы организации окружения приложения, файлы настроек.

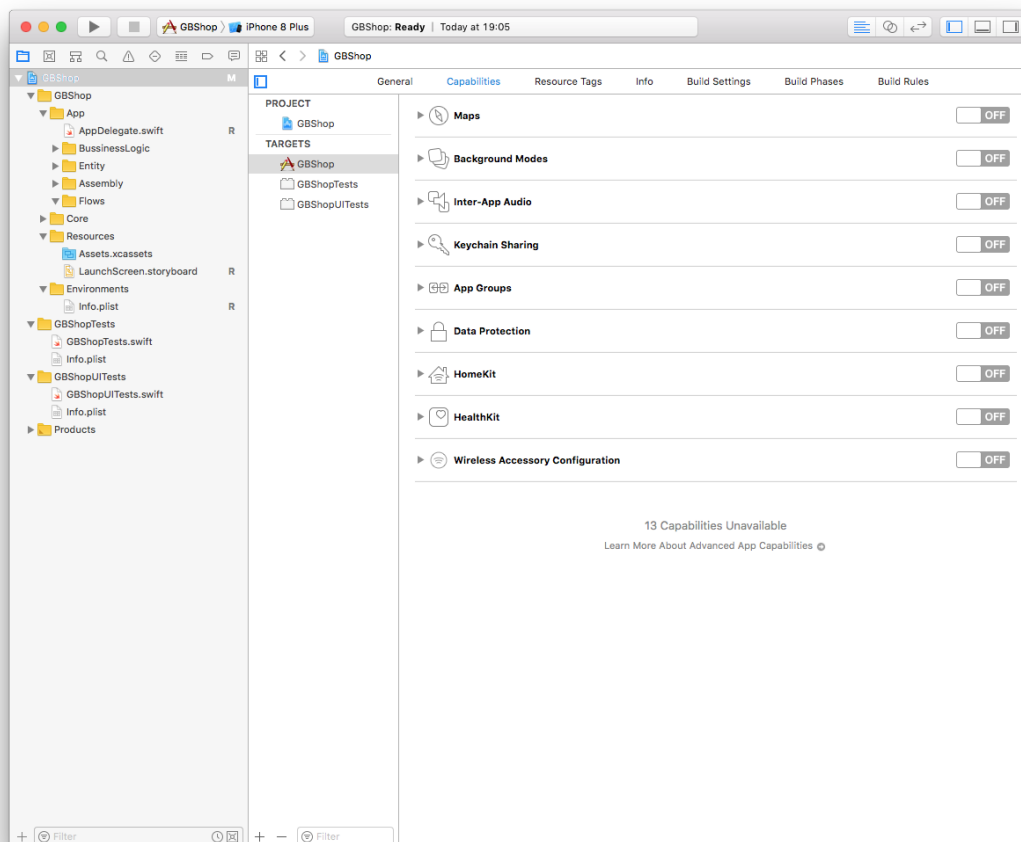
Перейдем в **Xcode** и применим описанную ниже структуру в проекте.



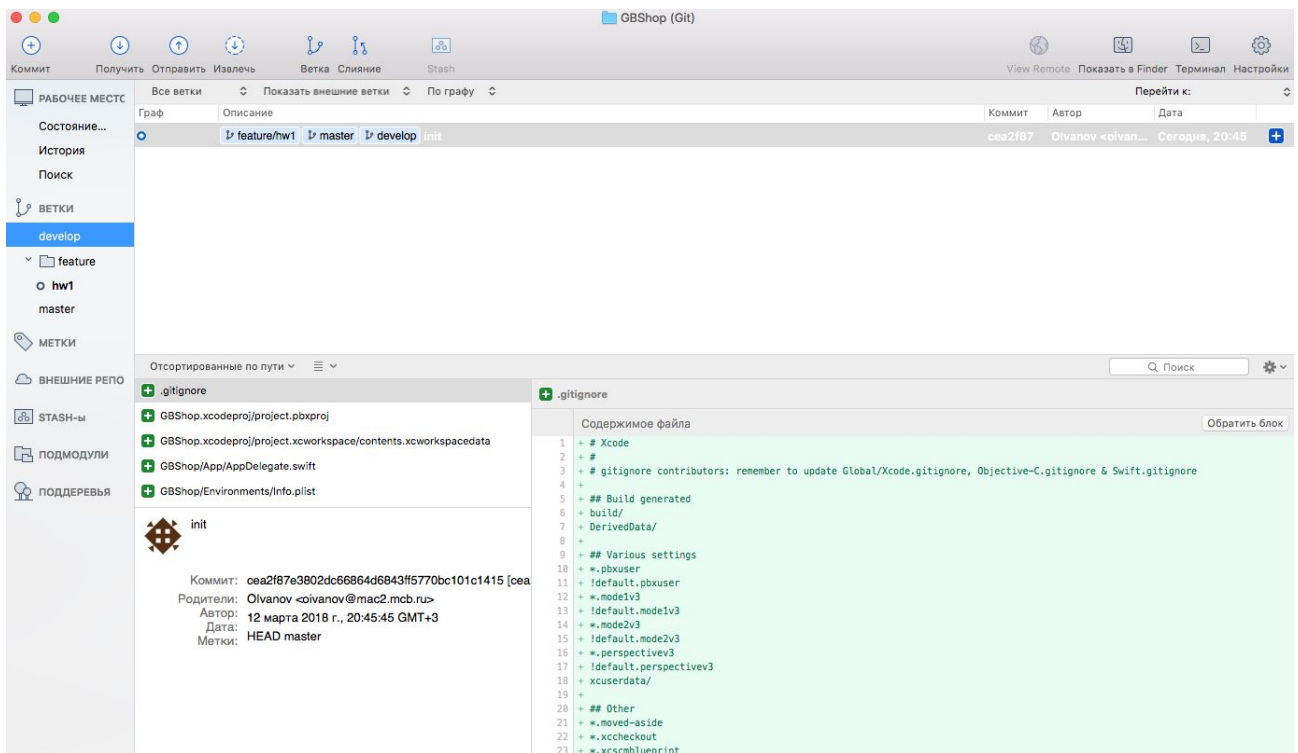
В проекте уже присутствуют файлы. Распределим их по созданным папкам.

Файл **assets.xcassets** идеально подходит папке **Resources**, перенесем его. Туда же отправим файл **launch.storyboard**, который отражает статичный экран загрузки приложения. Файл **info.plist** также подходит папке **Environments**, перенесем. Файл **AppDelegate.swift**, описывающий жизненный цикл приложения, копируем в корень папки **App**.

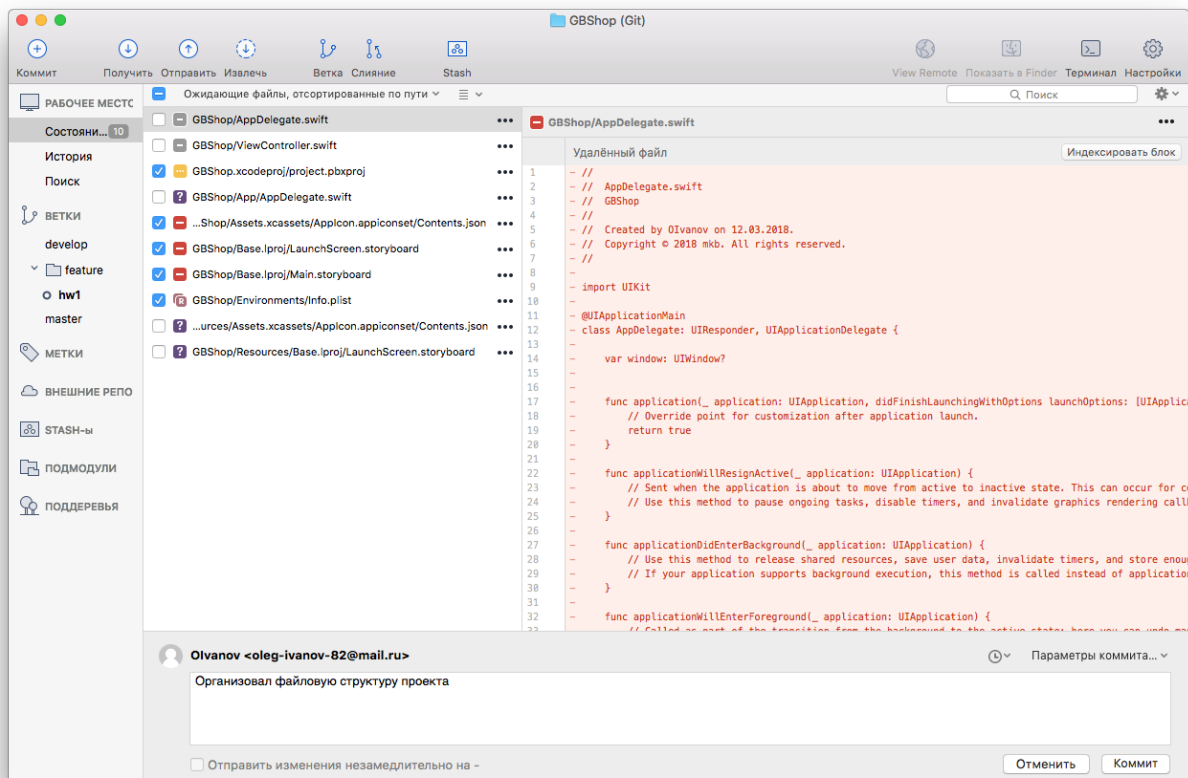
Удалим автоматически созданные файлы **ViewController.swift** и **Main.storyboard**. Позже, мы создадим новые файлы экранов в зависимости от нашей бизнес-логики. Из **Info.plist** необходимо не забыть удалить **Main** из строки **Main Interface**.

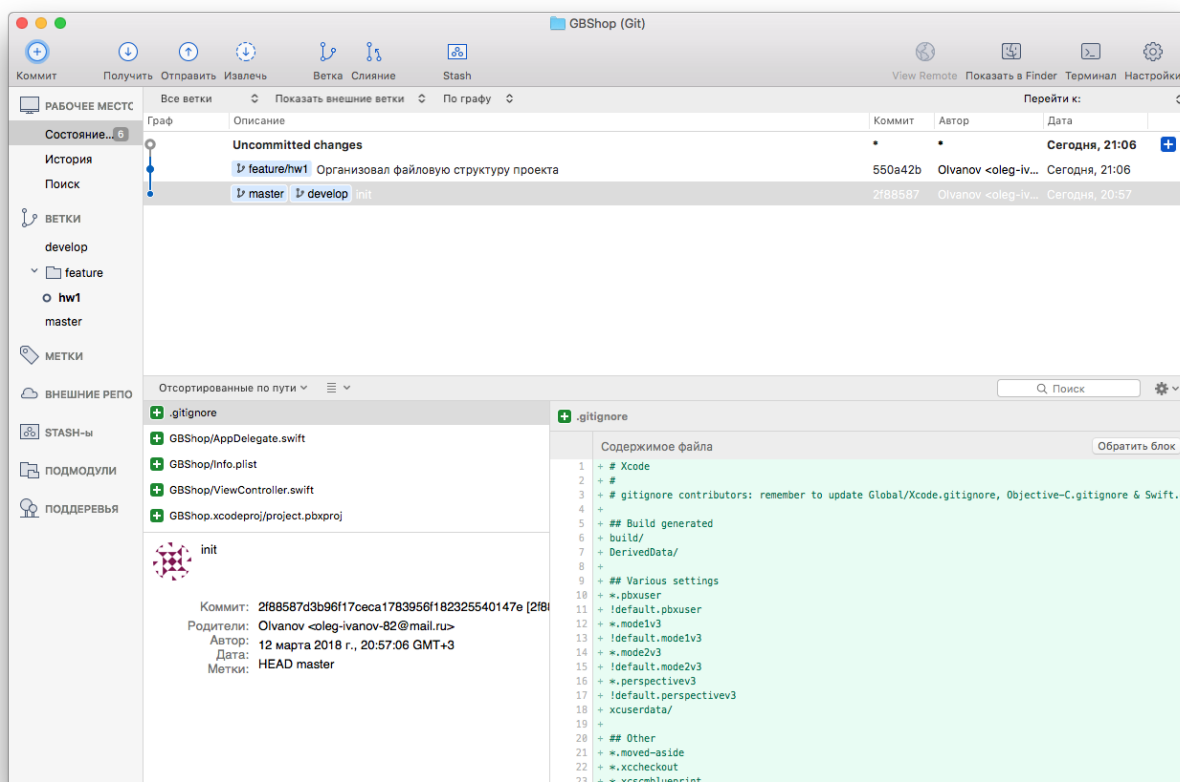


Осталось внести изменения в Git-репозиторий с использованием **sourcetree**. Сначала создадим новую ветвь в **feature** с именем **hw1** — впоследствии именно туда добавим изменения.



Нажмем на кнопку «Коммит» и добавим изменения в репозиторий. Уделите внимание комментарию к коммиту — всегда давайте осмысленное описание, какие именно изменения добавляете. Это важно для определения места вставки тех или иных изменений для возможной их модификации, отката на них или иных действий.





Практическое задание

1. Создать пустой проект.
2. Создать Git-репозиторий.
3. Добавить **.gitignore**.
4. Создать репозиторий на сервере **gitlab**.
5. Закоммитить локальный проект на сервер.
6. Создать локально ветку новой фичи (feature) по номеру урока.
7. Перейти в новую ветку, выполнить организацию структуры проекта.
8. Отправить изменения в Git-репозиторий на сервере.
9. Открыть **merge request** в ветку **develop**.
10. Назначить **request** на преподавателя.

Дополнительные материалы

1. [Git](#).
2. [sourcetree](#).
3. [Гайдлайны](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>.