Прожарка Tuist

Евтухов Александр
Teamlead iOS

открытие

План доклада

- Проблемы многомодульности на CocoaPods
- Альтернативные решения построения модульности
- Как выполнить миграцию на Tuist
- Как можно ускорить сборки без кеша
- Как работает кеш Tuist
- Какие у Tuist есть проблемы и как их решать
- Итоги

Кратко о проекте

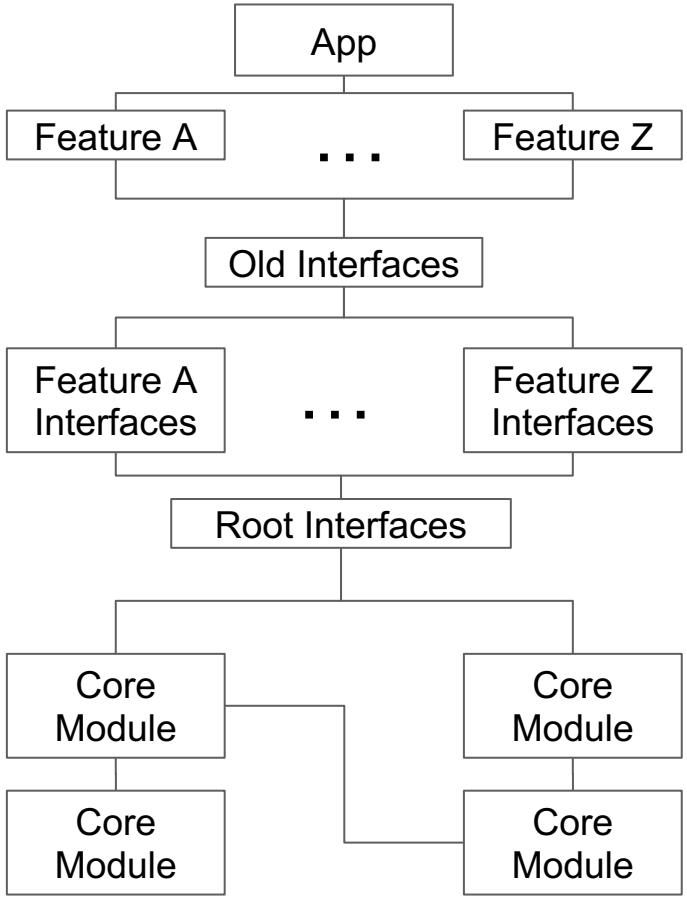
Многомодульность
- CocoaPods + LocalPods
- Глубокий патчинг CocoaPods

Внешние зависимости
- Предварительная сборка на базе Carthage

Около 40 продуктовых модулей

Всего 240 таргетов

Время холодной сборки 520 секунд



Мотивация к поиску альтернативы CocoaPods

1

Проблемы многомодульности на CocoaPods

- Огромное количество нечитаемого кода Ruby
- Долгая сборка проекта
- Еще более долгая сборка тестов
- Тестовые артефакты весят 30+ Гб что ставит крест на концепции распределенного тестирования
- Очень негибкий код, определяющий модульность проекта
- Нет инструментов кеширования
- Проблемы с копированием ресурсов

Доклад Авито где затрагивается тема патчинга CocoaPods

Наши хотелки в отношении новой системы сборки проекта

- Наличие механизмов, позволяющих писать какую-то автоматизацию при генерации таргетов
- Желательно наличие Кеша
- Нормальная работа с ресурсами (без их излишнего копирования)
- Быстрая генерация проекта

Главное - относительная простота и понятность разработчикам кор команды. А в идеале - среднестатистическому продуктовому разработчику.

Альтернативные инструменты для построения многомодульности

- Bazel
- Xcodegen
- SPM
- CocoaPods + Инструменты кеширования
  - Rugby
  - XCRemoteCache
- Tuist

Bazel

- Очень высокий порог вхождения
- Полный отказ от системы сборки Xcode
- Сторонняя утилита для генерации проекта (сейчас уже не актуально)
- Очень тяжело найти маленькие работающие примеры

XCodeGen

- Явное описание модулей и их зависимостей в формате YAML
- Не решает проблемы ускорения сборки
- Вообще не дает средств автоматизации

SPM

- Субъективный негативный опыт работы
- Проблемы с кешированием зависимостей
- Пока нет решений, которые поставляли бы кеш кода
- Очень нестабильная работа (добавление файла может запросто крашнуть XCode)
- Начальная загрузка всех пакетов, требующая подключения к инету
- До последнего момента отсутствие средств автоматизации и написания скриптов

CocoaPods + Сторонние утилиты кеширования

Rugby

- Отказался работать из-за пропатченных CocoaPods

XCRemoteCache

- Очень сырой инструмент
  (на момент когда мы его рассматривали)
- Завелся, но работал с ошибками

Tuist

- Написан на swift
- Описание проекта на swift
- Наличие механизмов focus и cache
- Очень низкий порог вхождения
- 1000+ примеров описания проекта и таргетов в оф. репозитории проекта
- Примеры компаний из РФ которые уже используют Tuist

Переход на Tuist
Выбор сложности

- **Легко**
- **Средний**
- **Сложный**
- **Я из Core команды Открытия**

Правила игры

- Не останавливать продуктовую разработку
- Менять схему модульности на ходу
- Работать сразу и с CocoaPods и Tuist
- Конфигурировать Tuist кодом CocoaPods

Организация модульности на Tuist

2

Выбор подхода к организации модульности

Полное описание модуля

```
let feature1 = Target(
        name: "Feature1",
        platform: .iOS,
        product: .framework,
        bundleId: "io.tuist.feature1",
        deploymentTarget: targetversion
        infoPlist: .default,
        sources: ["Targets/Feature1/Sources/**"]
)

let feature2 = Target(
        name: "Feature2",
        platform: .iOS,
        product: .framework,
        bundleId: "io.tuist.feature2",
        deploymentTarget: targetversion,
        infoPlist: .default,
        sources: ["Targets/Feature2/Sources/**"]
)
```

Использование прокси функций

```
func defFeature(name: String) -> Target {
    Target(
            name: name,
            platform: .iOS,
            product: .framework,
            bundleId: "io.tuist.\(name.lowercased())",
            deploymentTarget: targetversion,
            infoPlist: .default,
            sources: ["Targets/\(name)/Sources/**"]
    )
}


let feature1 = defFeature(name: "Feature1")
let feature2 = defFeature(name: "Feature2")
let feature3 = defFeature(name: "Feature3")
```

Полное описание модуля

Pros

- Никакой «магии»
- Максимальная прозрачность генерации

Cons

- Очень много кода
- 90% Этого кода – boilerplate
- Низкая толерантность к изменениям
- Полная противоположность концепции “динамической” генерации проекта

Вывод

Подходит для небольших проектов с фиксированной структурой

Использование вспомогательных функций

Pros

- Мало boilerplate кода

Cons

- Появление какой-то логики, в которой можно допустить ошибку
- Все еще не динамическая генерация

Вывод

Подходит для большинства проектов, так как уже в таком виде решает проблемы негибкости описания таргетов

Переход к динамической генерации проекта

Функция генерации таргета

```
func defModule(
 name: String,
 dependencies: [String],
 xcframeworks: [String]
) -> Target {
    ...
}
```

Podspec модуля

```
defcoremodule(
    name: "BaseInterfaceCode",
    summary: "Framework c Extensions",
    dependencies: [
        "TargetA", "TragetB"
    ],
    configs: { ... }
) do |m|
    m.vendored_frameworks = [
        ".../Calculator.xcframework"
...
    ]
end
```

Замечание

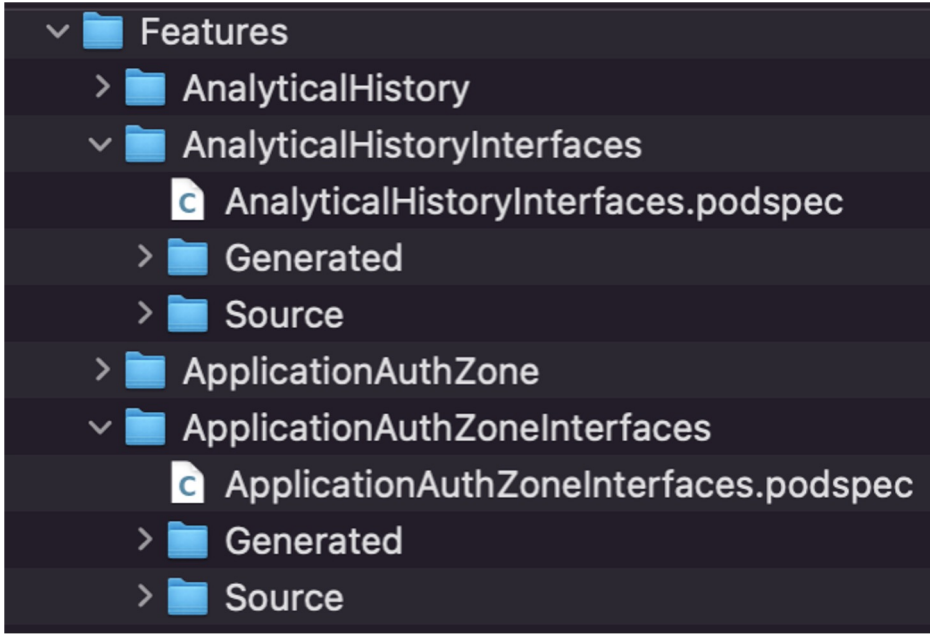
*.podspec выглядит так из-за наших патчей и оптимизаций. Но смысл от этого глобально не меняется. Данные достаются при помощи Regexp выражений

Динамическая генерация проекта
Получение списка модулей

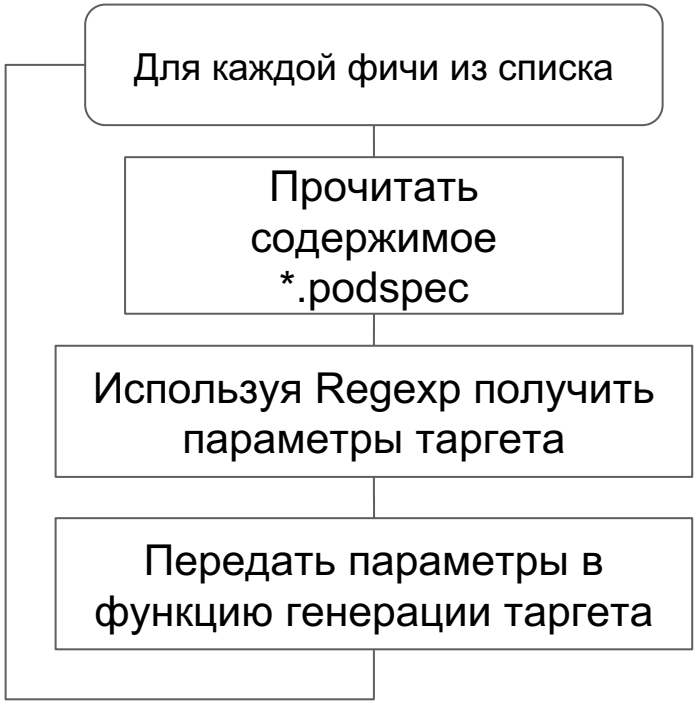
Взять все папки в директории Features

Отфильтровать папки, в корне которых есть *.podspec с тем же именем

Фактически так мы получаем список фичей + их директории



Динамическая генерация проекта Конвертация данных Podspec в описание модулей



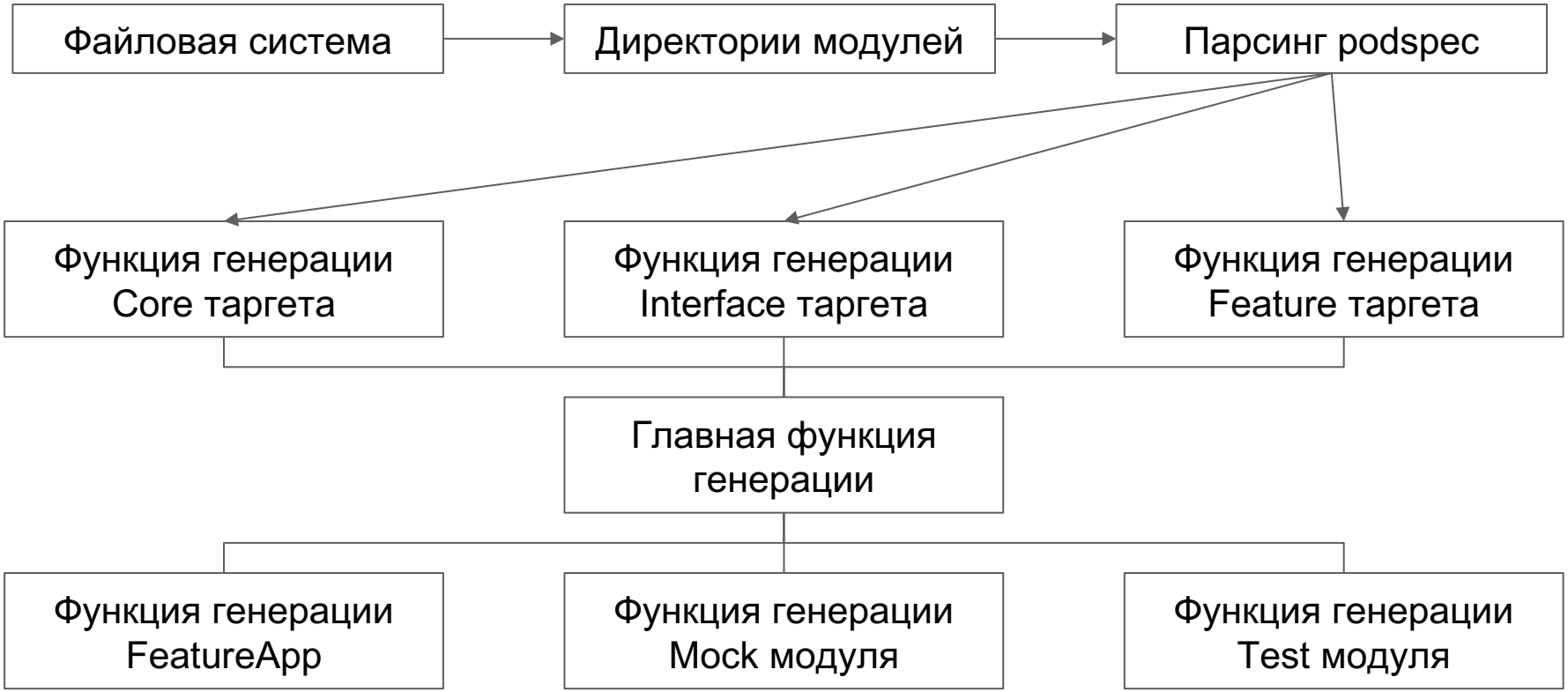
Podspec модуля

```
defcoremodule(
    name: "BaseInterfaceCode",
    summary: "Framework с Extensions",
    dependencies: [
        "TargetA", "TragetB"
    ],
    configs: { ... }
) do |m|
    m.vendored_frameworks = [
        ".../Calculator.xcframework"
...
    ]
end
```

Использование вспомогательных функций + динамическая генерация

- Pros
  - Минимум boilerplate кода
  - Максимальная гибкость описания таргета
  - Структура папок определяет проект
  - **Сборка проекта на базе конфигурации из CocoaPods**
- Cons
  - Очень много нетривиальной логики генерации
  - В случае ошибок, будет сложно их выявить

Иерархия вспомогательных функций генерации модулей



Управление генерацией проекта

- Переключение между whitelabel режимами
- Отключение от проекта лишних модулей
- Управление типом линковки

Пример

TUIST_CACHE=true \ - **проект собирается с использованием кеша**
TUIST_DEMO_MODE=ODW \ - **whitelabel режим - ODW**
TUIST_FORCE_DYNAMIC=true \ - **форсирование динамической линковки**
tuist generate

Whitelabel – приложение, которое имеет шаблонную структуру и поставляется разным заказчикам с минимальными изменениями дизайна, Apple Developer команды, логотипами.

Управление генерацией проекта

Файл констант

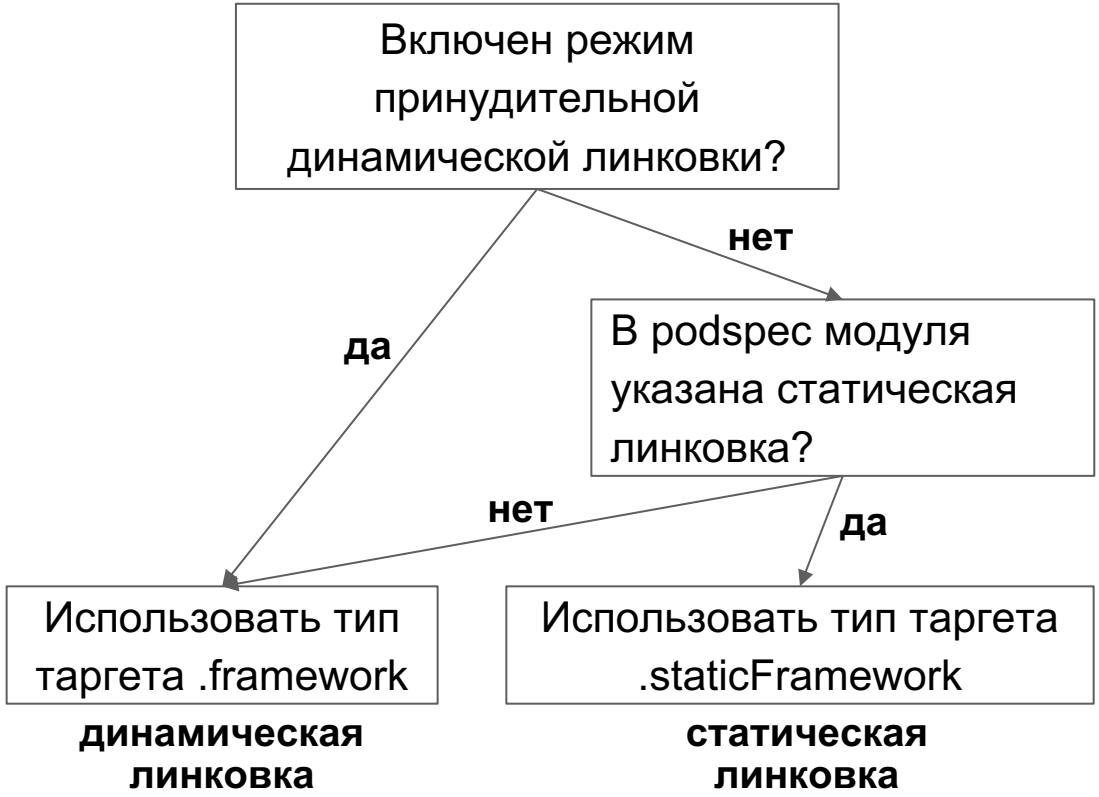
```
var forceDynamicLibGeneration: Bool = false //глобальный флаг
```

Парсинг параметров при запуске

```
private static func getForceDynamicLibsGenerationFlag() {
    let env = Environment.forceDynamic.getBoolean(default: false)
    forceDynamicLibGeneration = env
}
```

Управление генерацией проекта
Выбор типа линковки

```
let type: Product
if forceDynamicLibGeneration {
    type = .framework
} else {
    if isStatic {
        type = .staticFramework
    } else {
        type = .framework
    }
}
```



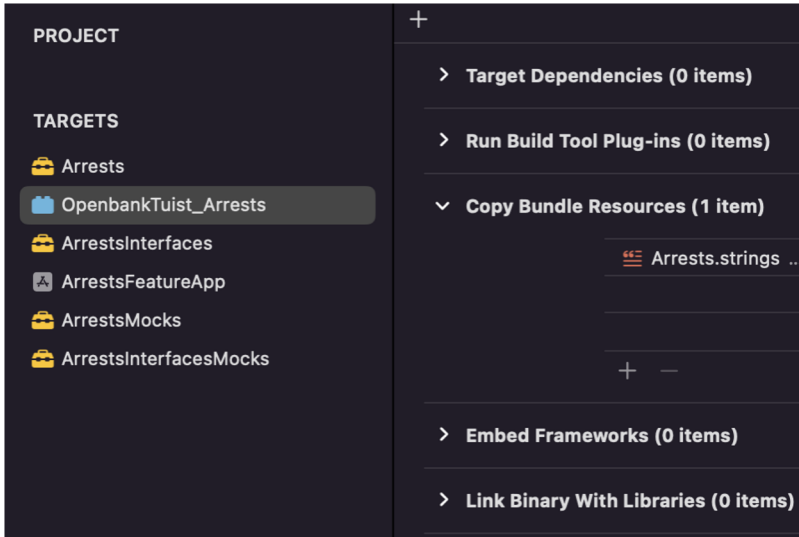
Работа с ресурсами
Переключение между бандлами

```
let bundle: Bundle
#if STATIC
    bundle = ArrestsResources.bundle
#else
    bundle = Bundle.module
#endif

let url = bundle.url(...)
```

- статическая линковка

- динамическая линковка



При изменении типов линковки модулей меняется бандл из которого необходимо извлекать ресурсы.

Работа с ресурсами
Пример swiftgen

```
private static func tr(_ table: String, _ key: String, _ args: CVarArg...) -> String {
   let bundle: Bundle

   #if STATIC
      bundle = {param.bundle}Resources.bundle
   #else
      bundle = Bundle.module
   #endif

   let format = bundle.localizedString(forKey: key, value: nil, table: table)
   return String(format: format, locale: Locale.current, arguments: args)
}
```

При работе со swiftgen необходимо использовать самописные кастомные шаблоны с поддержкой динамического выбора таргета

Внешние зависимости
Способы подключения к проекту

Доступные инструменты

- SPM
- Carthage

Пример

```
//Dependencies.swift

let dependencies = Dependencies(
    carthage: [
        .github(path: "...", requirement: .exact("0.20.0")),
    ],
    swiftPackageManager: [
        .remote(url: "...", requirement: .exact("0.20.0"))
    ],
    platforms: [.iOS]
)
```

Сравнение SPM и Carthage SPM

Pros

- Зависимости исходным кодом
- Возможность прогреть зависимости кешем

Cons

- Затраты на компиляцию кода
- Нестабильная работа инструмента
- Постоянная необходимость скачивать зависимости заново

Сравнение SPM и Carthage
Carthage

Pros

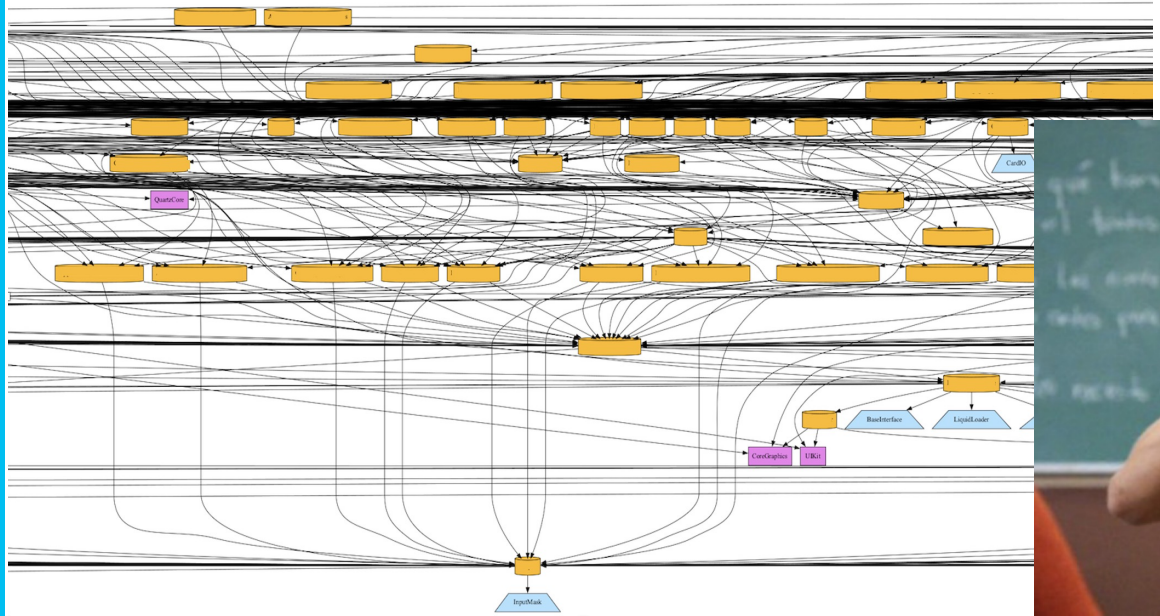
- Нет необходимости собирать код каждый билд

Cons

- При некоторых действиях зависимости нужно пересобирать

Генерация графа проекта

- Визуализация графа таргетов проекта
- Помощь в поиске ошибок в связях модулей
- Удобный инструмент для планирования рефакторинга связей

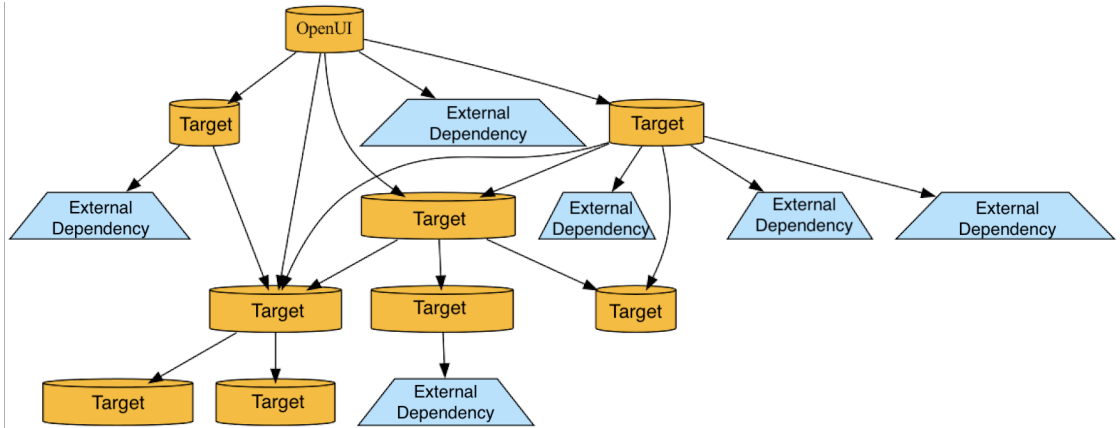


Генерация графа проекта
Способы ужать граф

tuist graph OpenUI - фокусировать на определенном таргете (эквивалентно указанию корня дерева)

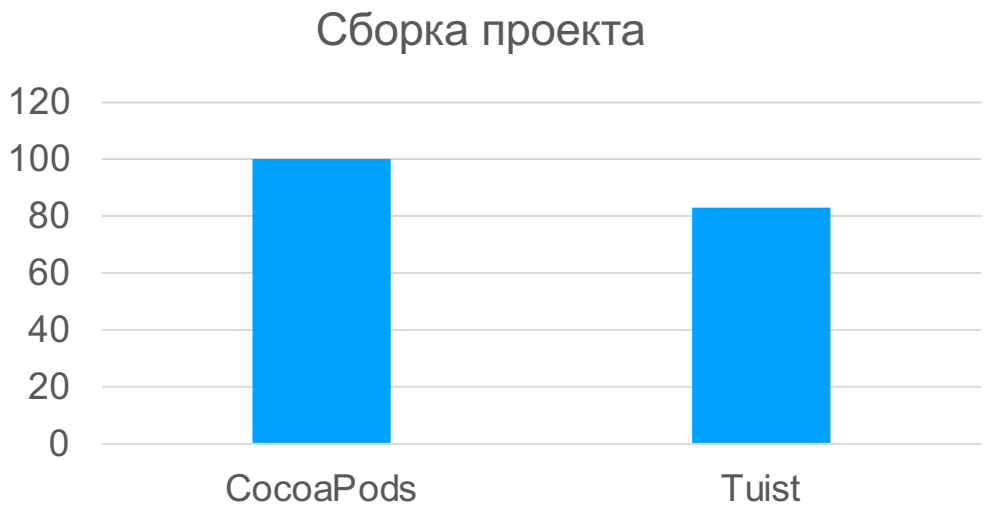
TUIST_NO_TEST= true tuist graph - управлять генерацией графа через кастомные параметры (не генерировать лишние таргеты на этапе сборки проекта)

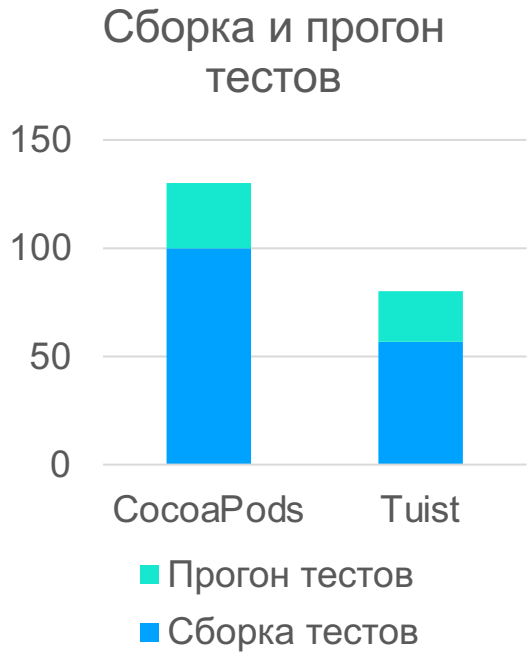
Использовать встроенные **флаги -t и -d** (отключение тест таргетов и зависимостей)



Миграция проекта на Tuist
Итоги

- Крайняя гибкость в генерации проекта
- Ускорение сборки проекта и тестов
- Режимы фокуса и постороения графов
- Решение проблемы распределенного тестирования





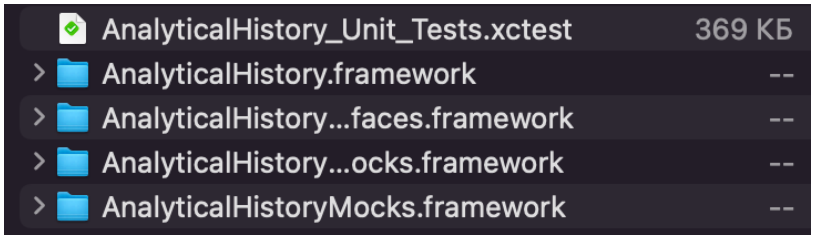
* графики приведены относительно времени сборки на CocoaPods

Как Tuist решает проблему распределённого тестирования

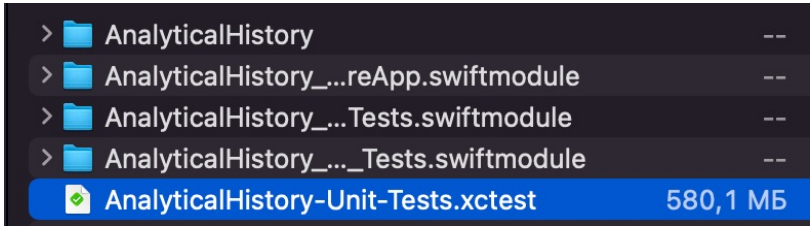
Tuist

Вес папки Products

Cocoa Pods



1.3 Gb Суммарно



30+ Gb Суммарно

Сборка тестов на **мощных** CI

Трансфер артефактов по сети

Тестирование на **слабых** CI

Тестирование на **слабых** CI

Ускорение сборки проекта без использования кеша

3

Ручная предварительная сборка зависимостей

Проблема

Чаще всего внешние зависимости поставляются в проект исходным кодом, что вынуждает разработчика тратить время на сборку этого кода

Решение

Все сторонние зависимости можно заранее собирать и сохранять в репозиторий, готовые XCFramework-и

Как можно предсобрать зависимости с помощью Carthage?

- Выгружаем зависимости (проекты)
- Патчим проекты
- Собираем в режиме xcframework
- Архивируем в zip
- Сохраняем в Git
- Подключаем xcframework к проекту

Carthage
Сборка кода в xcframeworks

xcframework

Git LFS
Хранение в Git

XCode
Подключение к проекту

Патчинг проектов carthage

Можно использовать простой ruby скрипт на базе библиотеки xcodeproj

- Отключить сборку лишних архитектур
- Включить флаги сборки ModuleStability для работы на разных XCode
  - BUILD_LIBRARY_FOR_DISTRIBUTION' = 'YES'
  - 'SKIP_INSTALL' = 'NO'
- Отключить CodeCoverage
- Включить статическую линковку
  - 'MACH_O_TYPE' = 'staticlib'

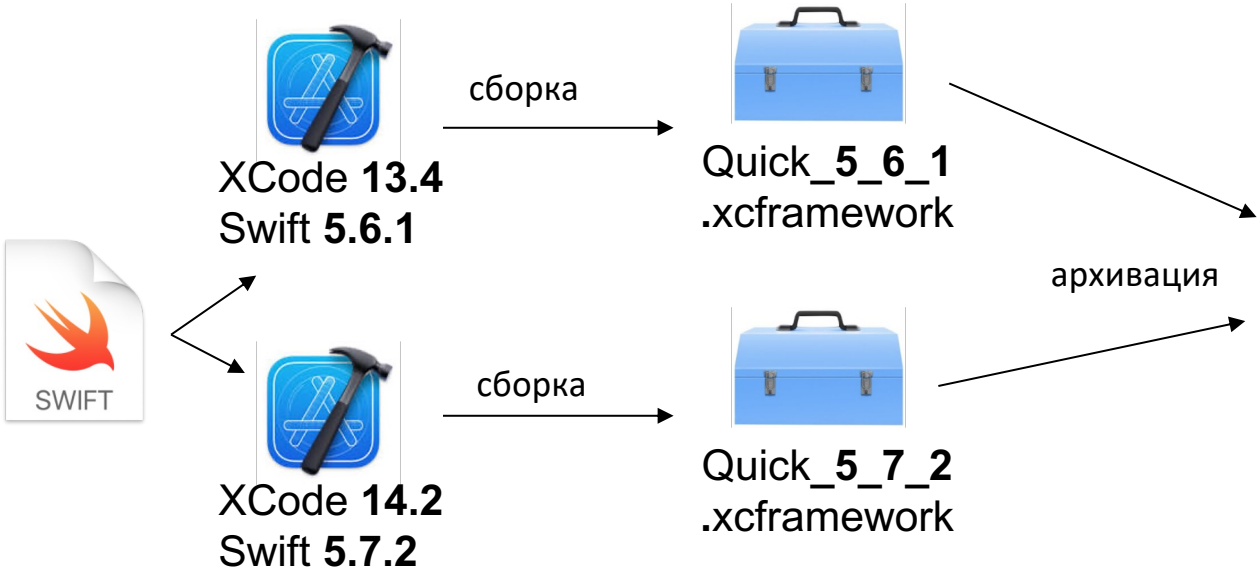
Тестовые утилиты

EnableTestability vs ModuleStability

EnableTestability не совместима с ModuleStability

Т.е. необходимо выбирать между возможностями

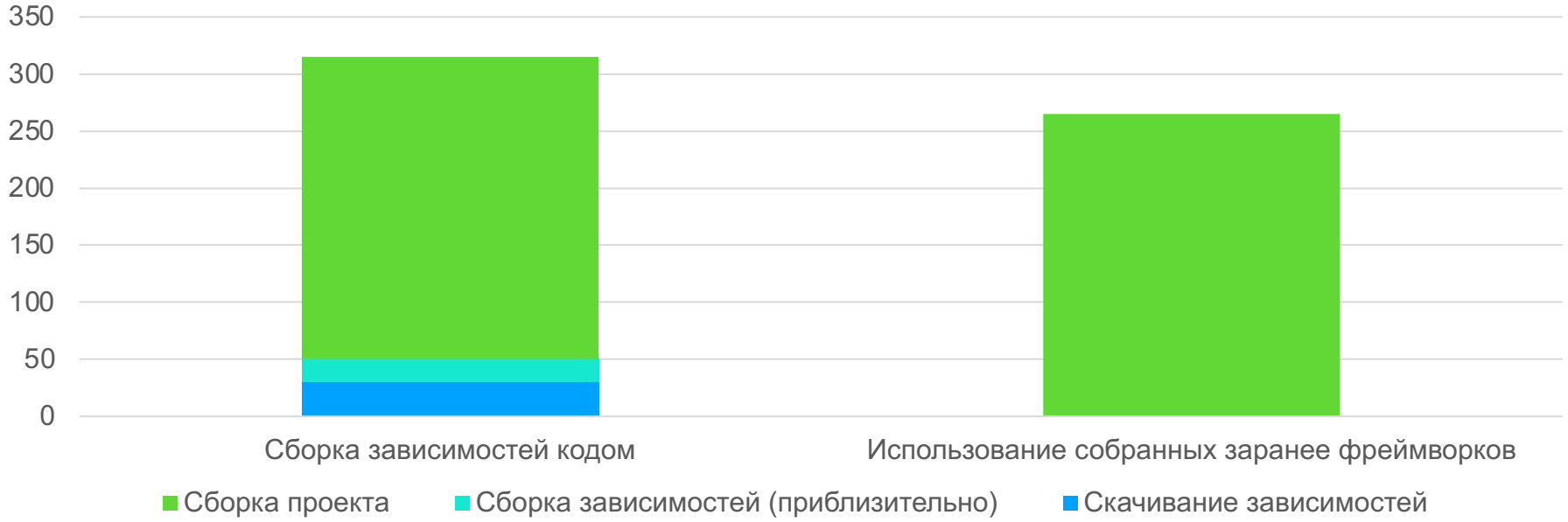
- или иметь ModuleStability
- или писать @Testable import ***



Git LFS
В git хранятся версии библиотек

Предварительная сборка зависимостей Бенчмарки

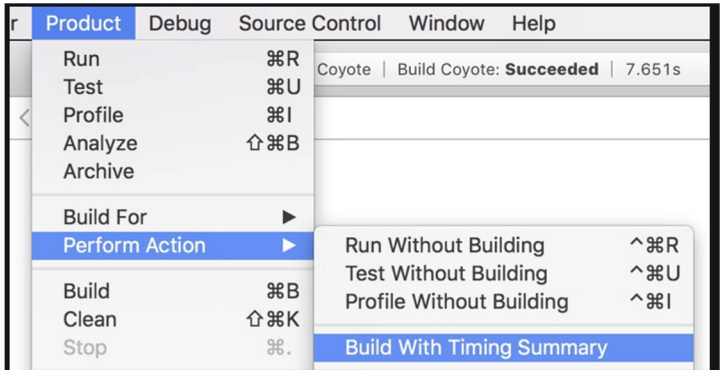
В среднем сокращение времени сборки составляет до 7% времени.

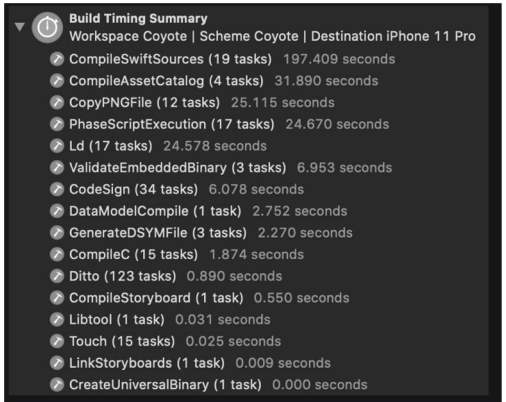


Оптимизация времени сборки
Типовые тяжелые операции

Операции, на которые постоянно тратятся ресурсы при сборке, но которые редко меняются

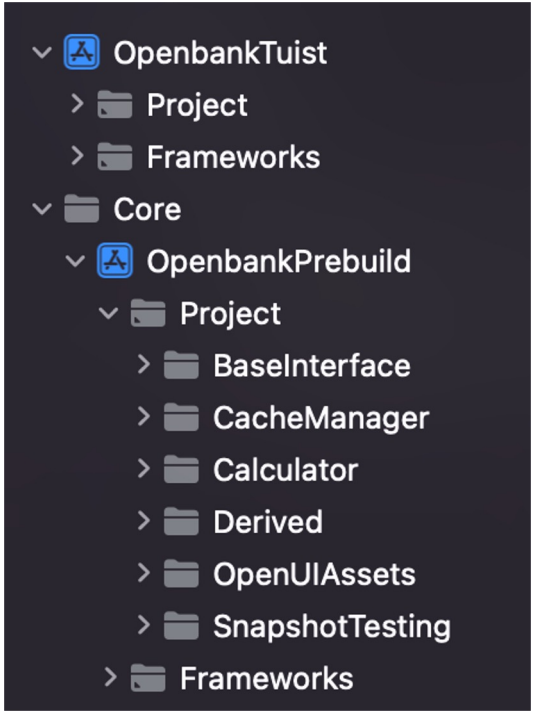
- Компиляция IBфайлов
- Компиляция XCassets (особенно если их много)
- Просто какой-то статический код библиотек, которые редко меняются





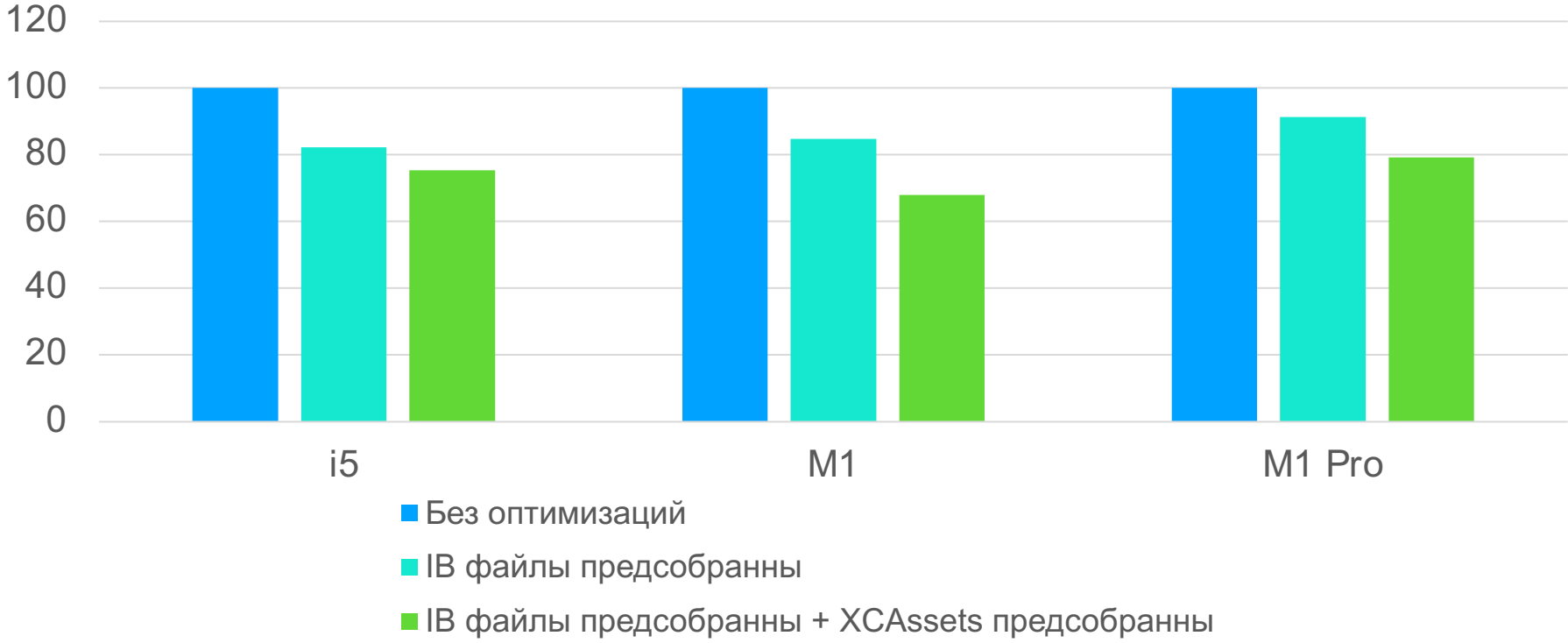
Предсборка таких таргетов
Выделение кода в отдельный проект

- BaseInterface
  - Все xib файлы без swift кода
- OpenUIAssets
  - xcassets
  - Swiftgen файл
- Статические библиотеки
  - Просто переносим таргеты в отдельный проект



Предсборка таргетов
Бенчмарки результатов

В среднем сокращение времени сборки составляет до 20-25% времени.



Команда focus

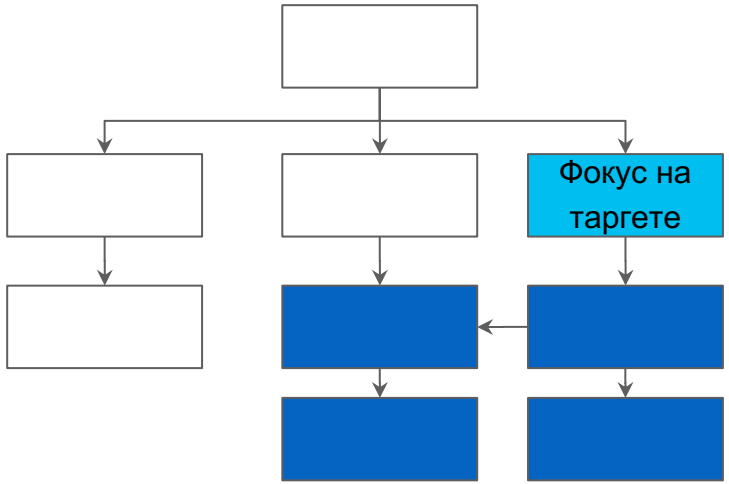
Фокусировка на 1м продуктовом таргете с фича апом дает очень весомое сокращение проекта

$$39 \text{ продуктовых модулей } * 5 \text{ таргетов (минимум)} = 195 \text{ таргетов}$$

$$\frac{195 \text{ лишних таргетов}}{240 \text{ кол} - \text{во таргетов в полном проекте}} = 81\% \text{ выброшенного кода}$$

Типовой состав одного продуктового модуля

- Код
- Моки модуля
- Моки интерфейсов
- Фича ап
- Таргет Юнит тестов



MonoTest таргет

Проблема

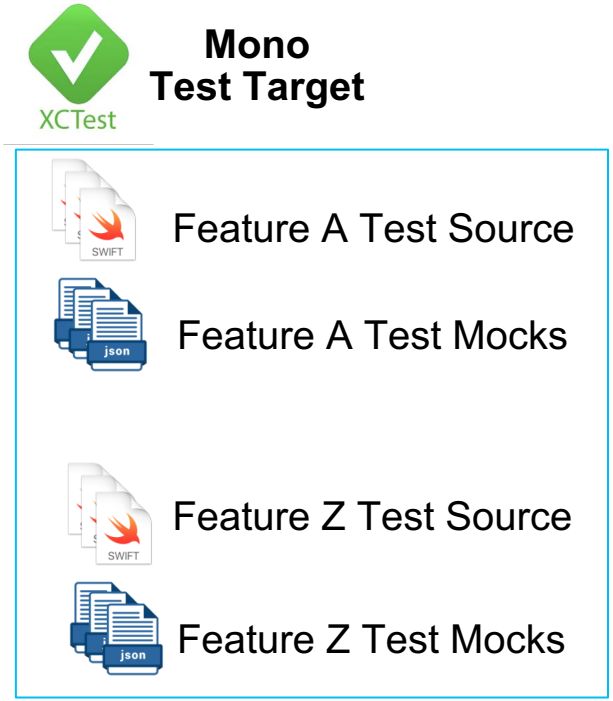
Большой вес xctest файла может замедлять их “установку” на симулятор и замедлять запуск теста

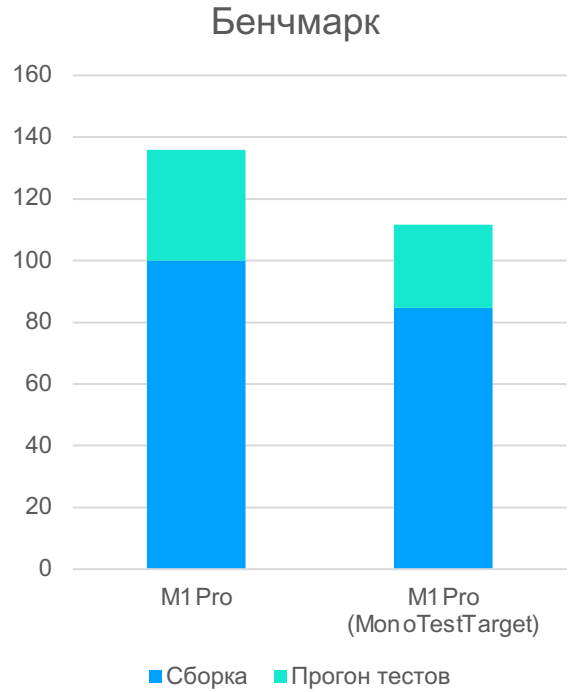
Tuist решает проблему веса тестовых артефактов, но переключение между тестовыми бандлами все еще может приводить к медленному исполнению тестов (в последовательном режиме)

MonoTest таргет

Решение

Для задачи прогона всех тестов в проекте можно прибегнуть к формированию единого тестового таргета





Ускорение сборок без использования кеша
Итоги

- Заранее собирайте внешние зависимости
- Выделяйте в отдельные таргеты редко-используемый “тяжелый” в компиляции код
- Может помочь использование MonoTest таргета для прогона всех тестов

Tuist cache

4

Локальный кеш

Последовательность команд при прогреве SPM зависимостей

- tuist fetch
- tuist cache warm -d
- tuist generate

SPM
Получение исходных кодов зависимостей

Tuist Cache
Кеширование зависимостей в framewok-и

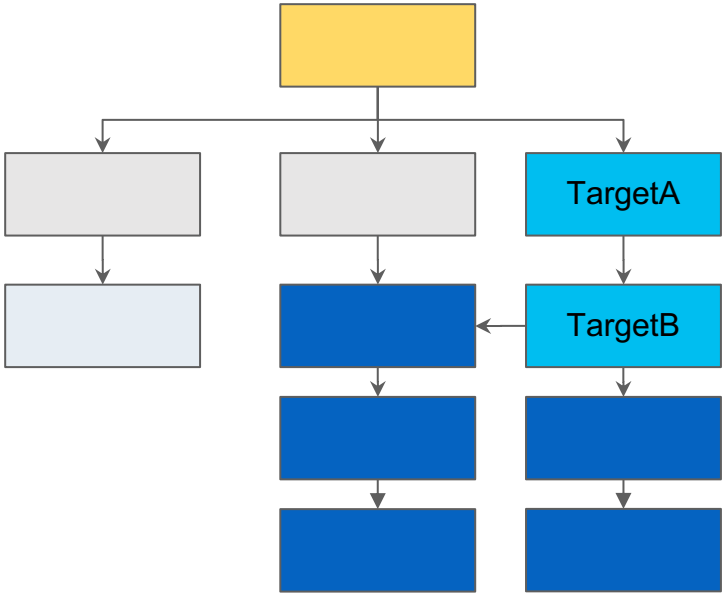
framework

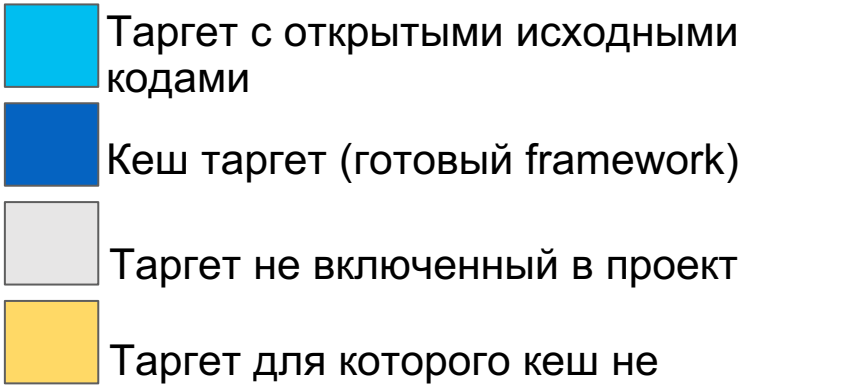
Tuist Generate
Подключение к проекту вместо исходников

Локальный кеш

Последовательность команд при прогреве фичей

- tuist cache warm
- tuist generate TargetA TargetB





Сетевой кеш Tusit

```
// Config.swift

return Config(
   cloud: .cloud(
projectId: "your-organization",
url: "http://127.0.0.1:8080",
options: []
   )
)
```

Pros

- Использование AWS под капотом в качестве хранилища данных
- Имеет полный набор утилит для своей работы
- Содержит красивый UI для авторизации

Сетевой кеш Tusit

Pros

- Использование AWS под капотом в качестве хранилища данных
- Имеет полный набор утилит для своей работы
- Содержит красивый UI для авторизации
- Все очень удобно

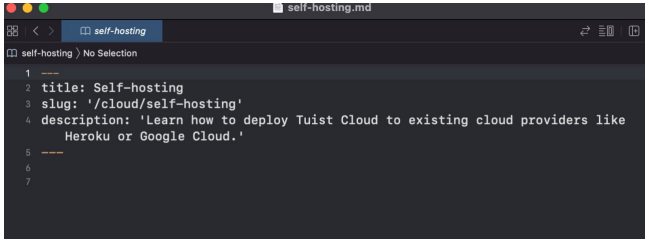
Cons

- **Ни один инфобез не позволит вам использовать AWS**

Directed by
ROBERT B. WEIDE

Почему мы не стали использовать self-hosted решения tuist cloud

На момент наших первых экспериментов с tuist cache инструкция по запуску self-hosted решения выглядела следующим образом:



Также мы опасались что

- это решение вдруг станет платным;
- его поддержку прекратят;
- оно перестанет иметь возможность самостоятельного хостинга.

Наш план по обходу использования Tuist Cloud + AWS

- Открыть исходники
- Происследовать как tuist работает с сервером
- Написать свой middleware backend который будет имитировать его поведение
- Profit…

Основные элементы в работе сетевого кеша Tuist

- Авторизация
  - Сложный набор действий, который в общем случае отвечает за получение аксес токенов
- Работа с middleware backend
  - Master server – управляющий сервер
  - Slave сервера для хранения файлов

Обход авторизации

```
let serverURL = URL(string: "http://127.0.0.1:8080")!

let keychain = Keychain(
    server: serverURL,
    protocolType: .https,
    authenticationType: .default
)
    .synchronizable(false)
    .label("\(serverURL.absoluteString)")

try! keychain.set("project-id", key: "LOL KEK")
```

Конфигурация кеша

```
// Config.swift

return Config(
   cloud: .cloud(
projectId: "your-organization",
url: "http://127.0.0.1:8080",
options: []
   )
)
```

Принятые упрощения

- Мы отказываемся от авторизации и не проверяем токен с которым происходит обращение к middleware бекенду.

Но это не значит что вы не можете развить эту идею и начать его использовать*

Принятые упрощения

- Мы отказываемся от авторизации и не проверяем токен с которым происходит обращение к middleware бекенду.
- Мы не будем решать коллизии
- Все файлы хранятся на 1м сервере
- Мы не будем использовать «сроки годности файлов»
- Наличие файла на Slave сервере проверяется HEAD запросом
- Мы будем считать, что в этой системе есть только один проект с одним ключом (идентификатором)

Подготовка master сервера

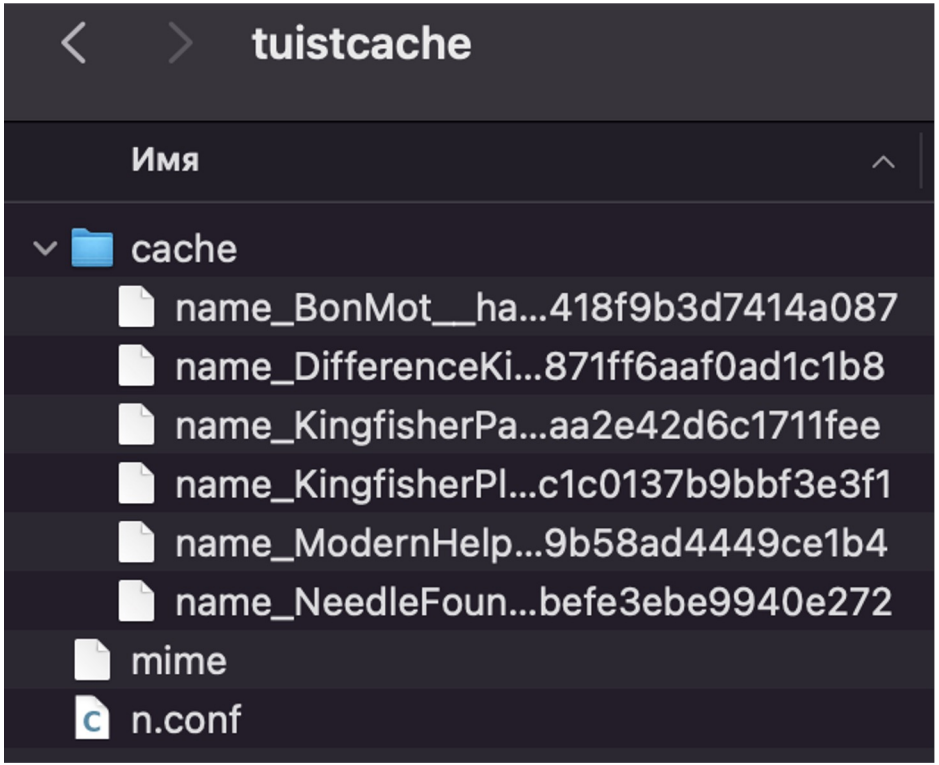
Требования

- Поддержка REST API

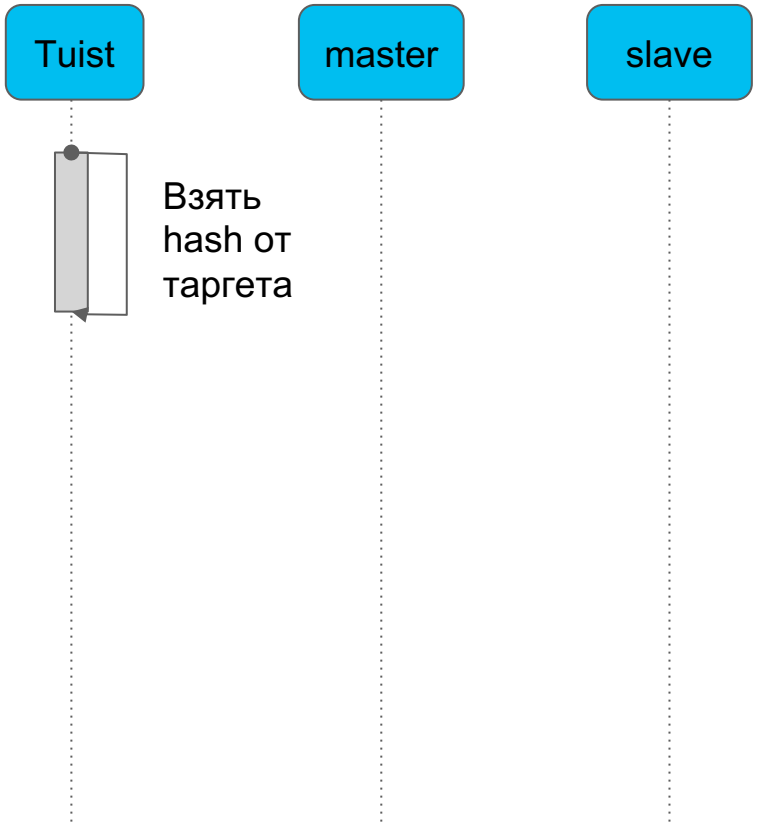
С учетом принятых упрощений большего не требуется

Подготовка slave сервера

- Установка nginx
- Формирование config файла
- Загрузка mime файла
- Запуск

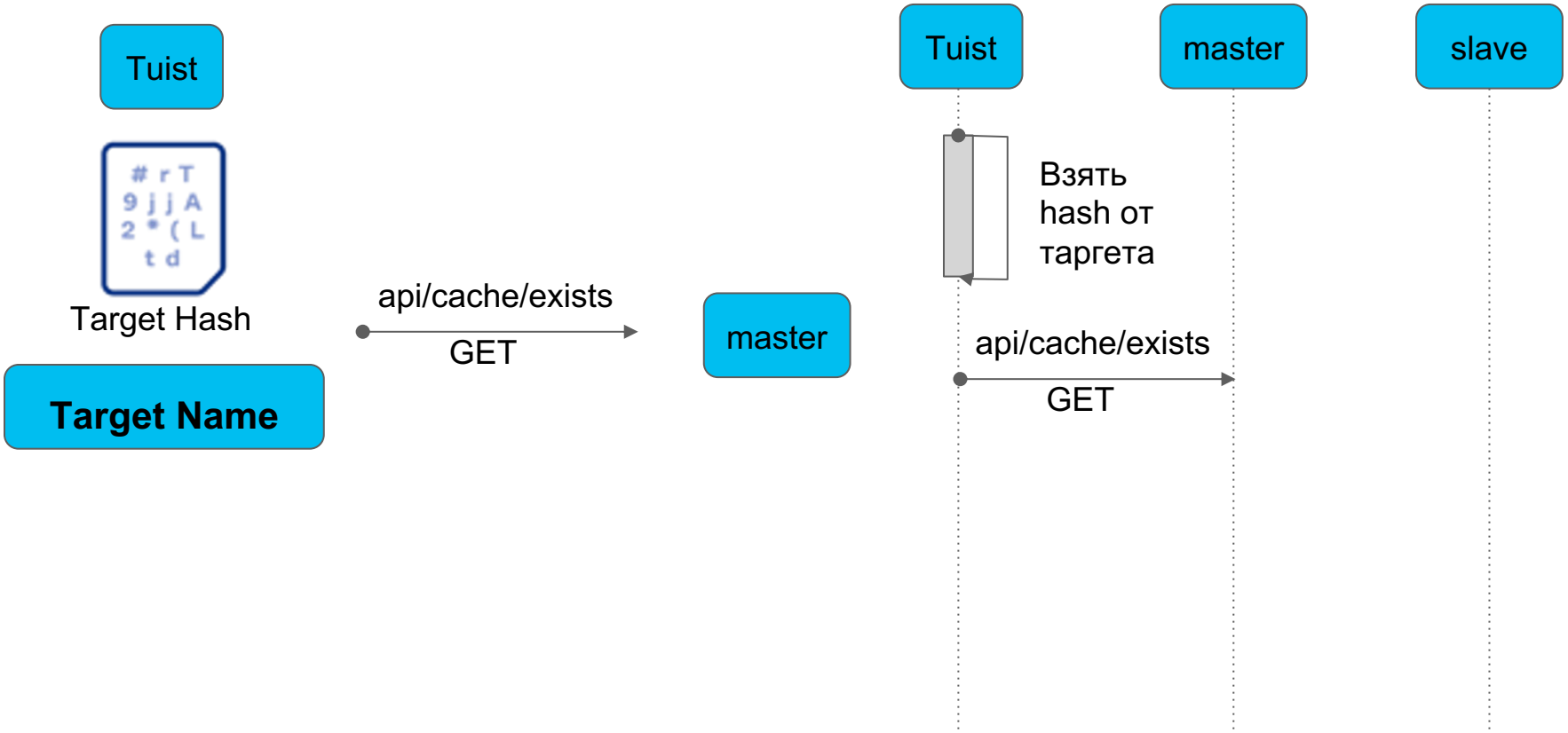


Прогрев кеша
Взятие hash от таргета

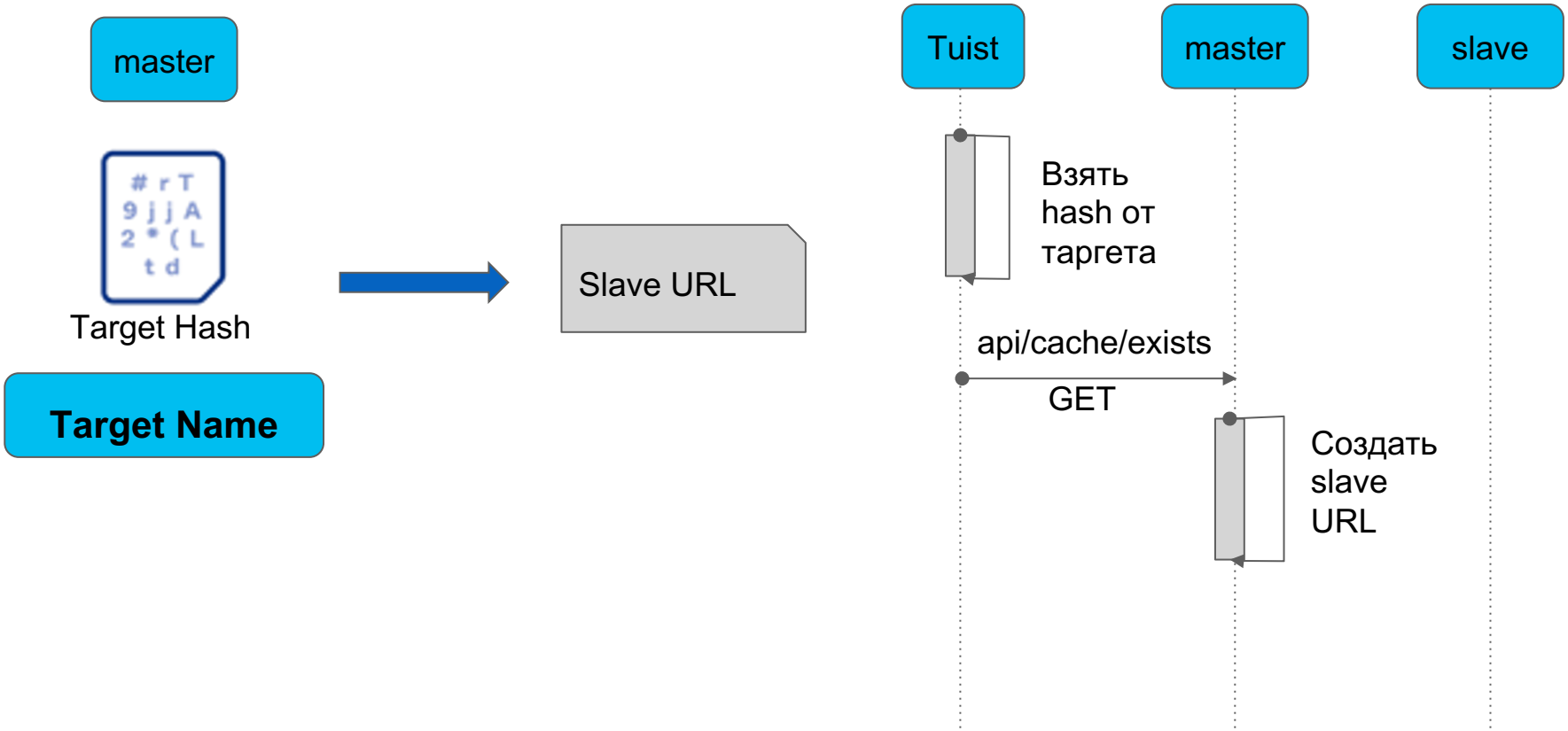


- Учитываются исходники
- Исходники
- Настройки таргета
- Версия XCode

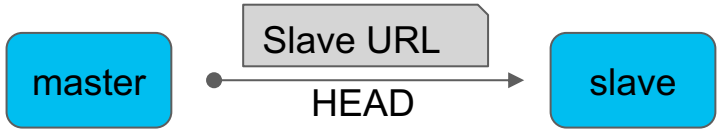
Прогрев кеша
Проверка наличия кеша для таргета

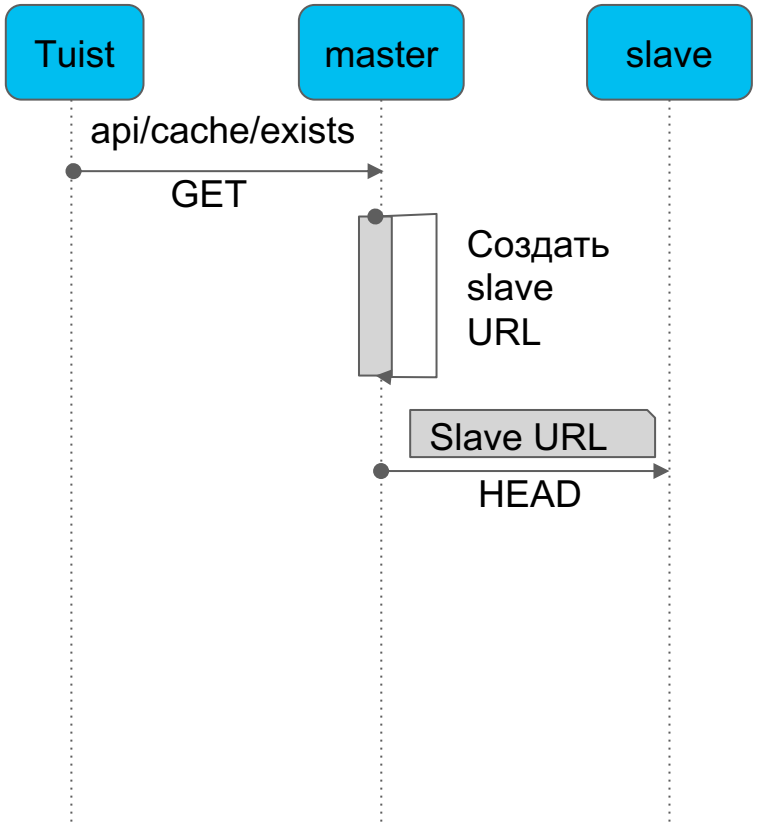


Прогрев кеша
Проверка наличия кеша для таргета

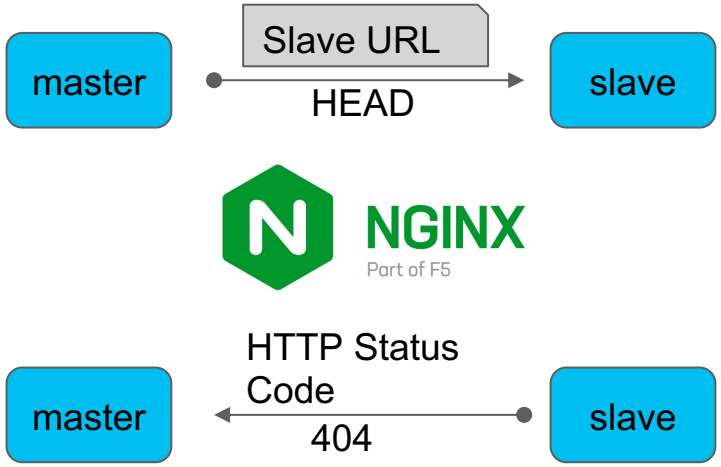


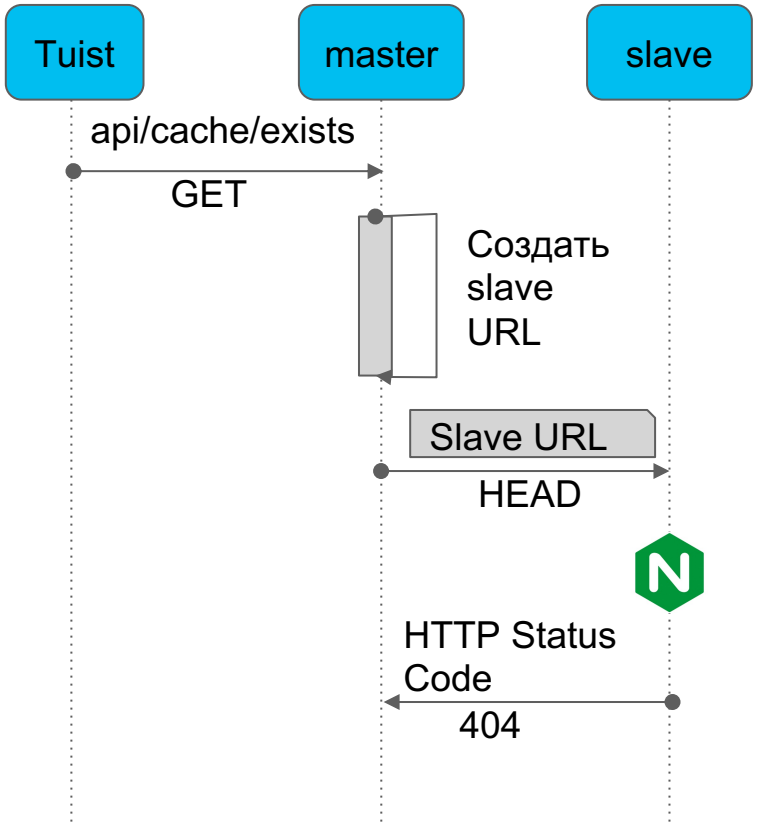
Прогрев кеша
Проверка наличия кеша для таргета



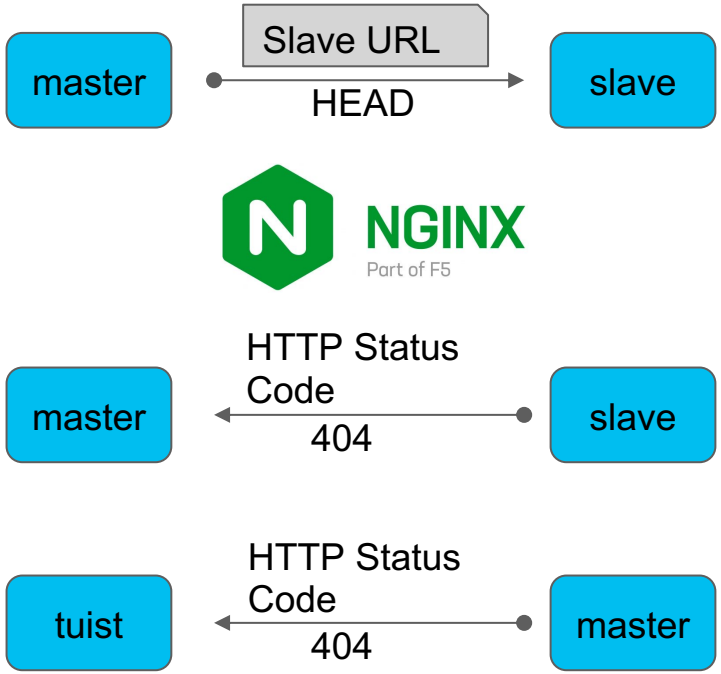


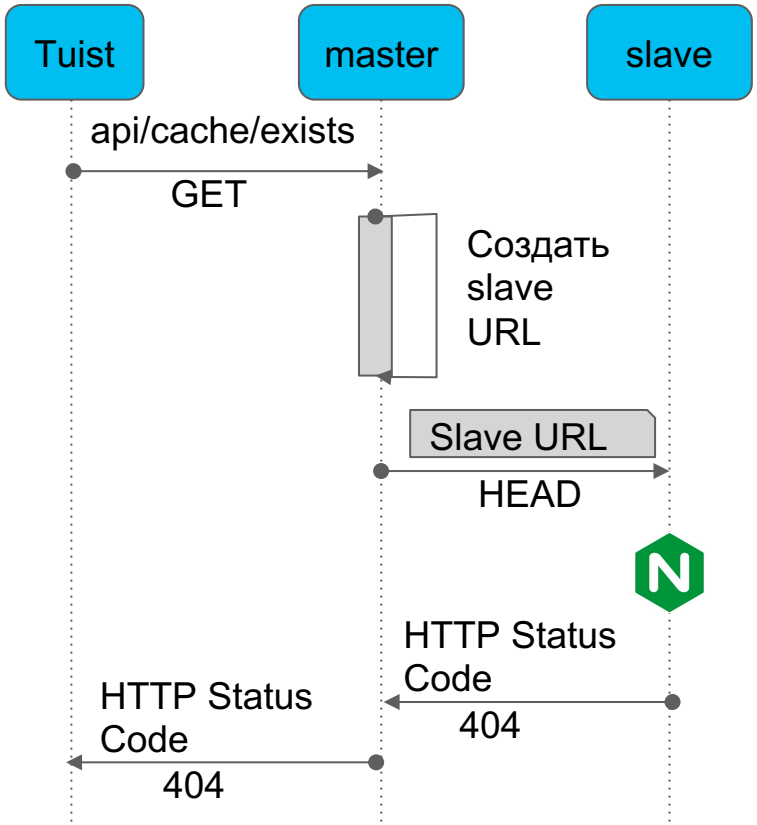
Прогрев кеша
Проверка наличия кеша для таргета





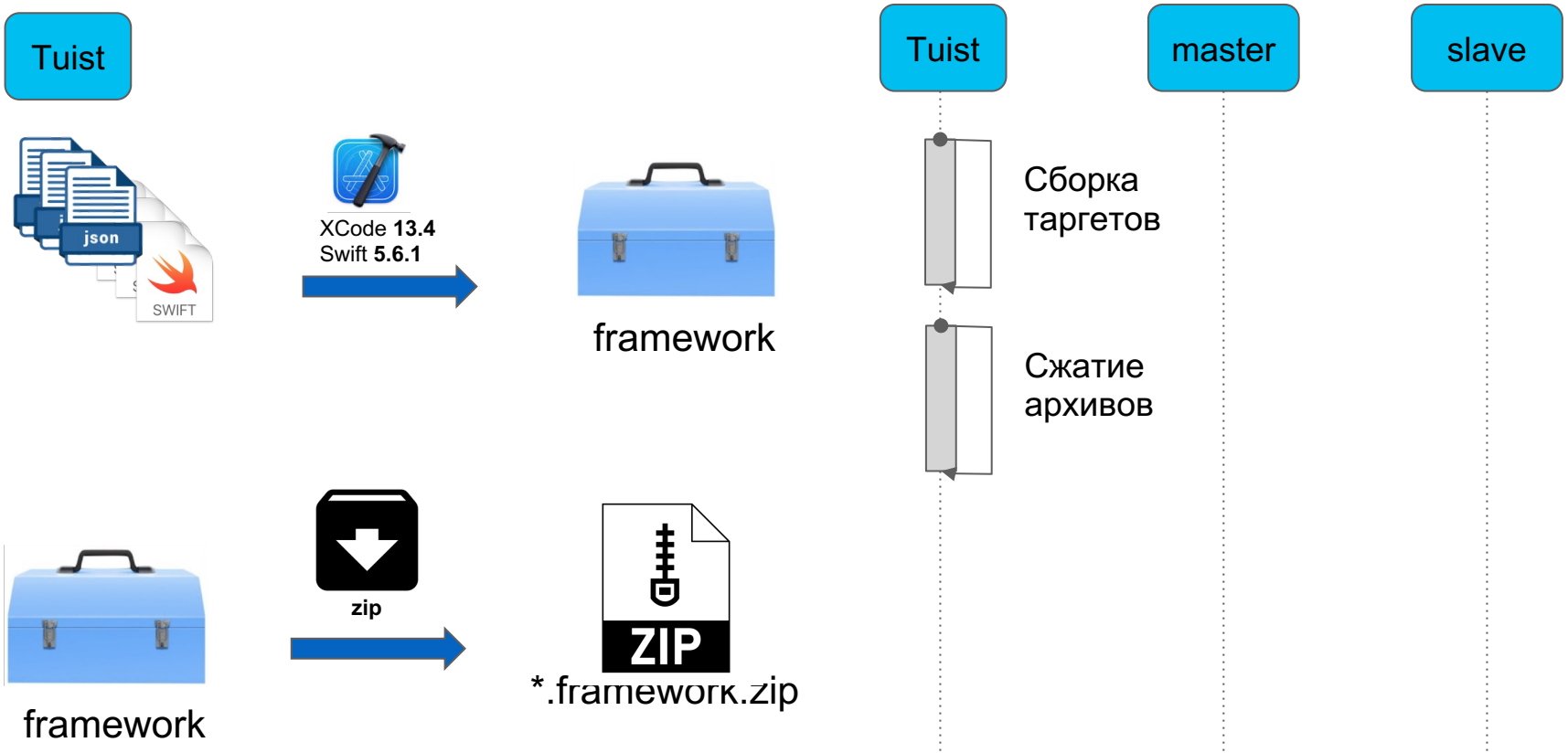
Прогрев кеша
Проверка наличия кеша для таргета



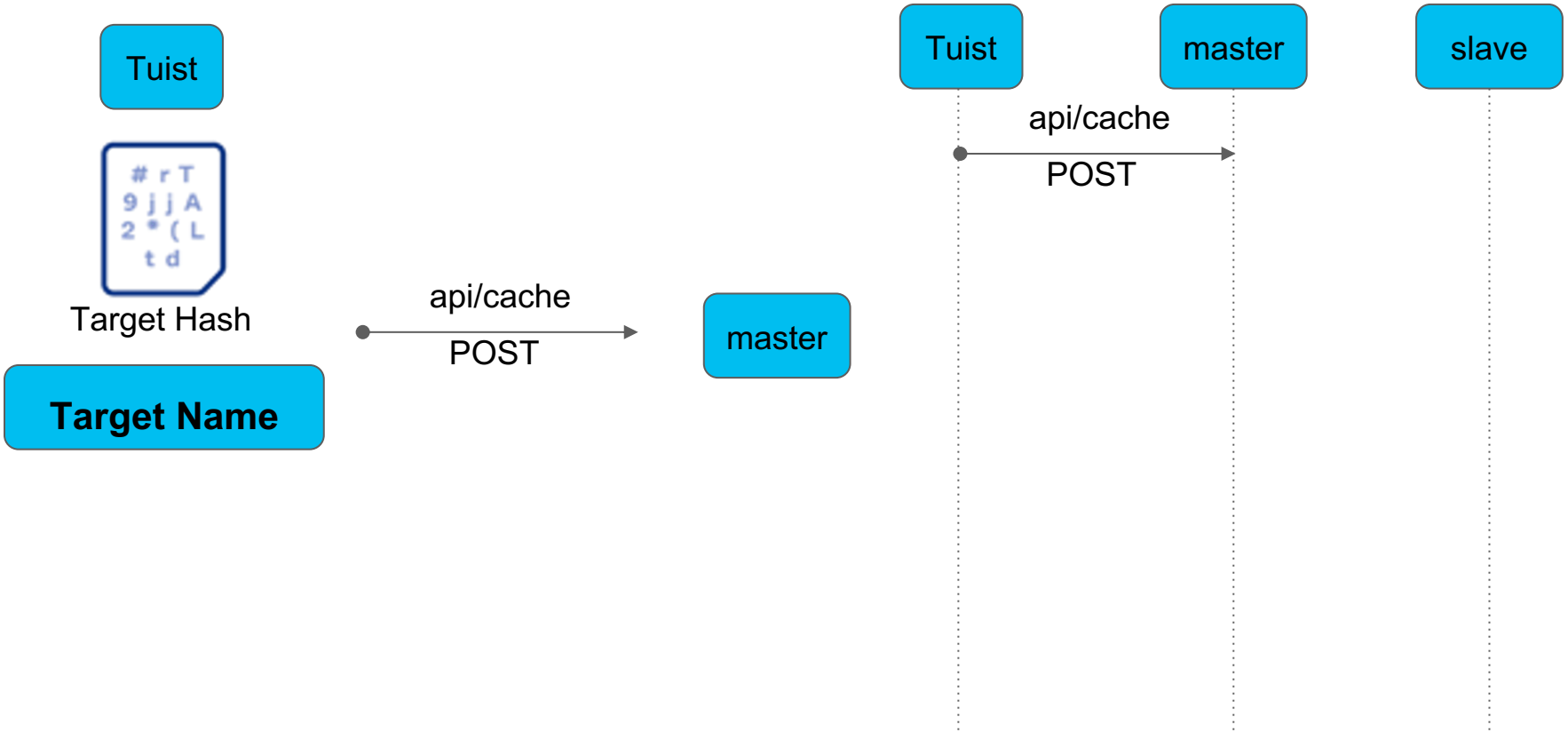


Прогрев кеша

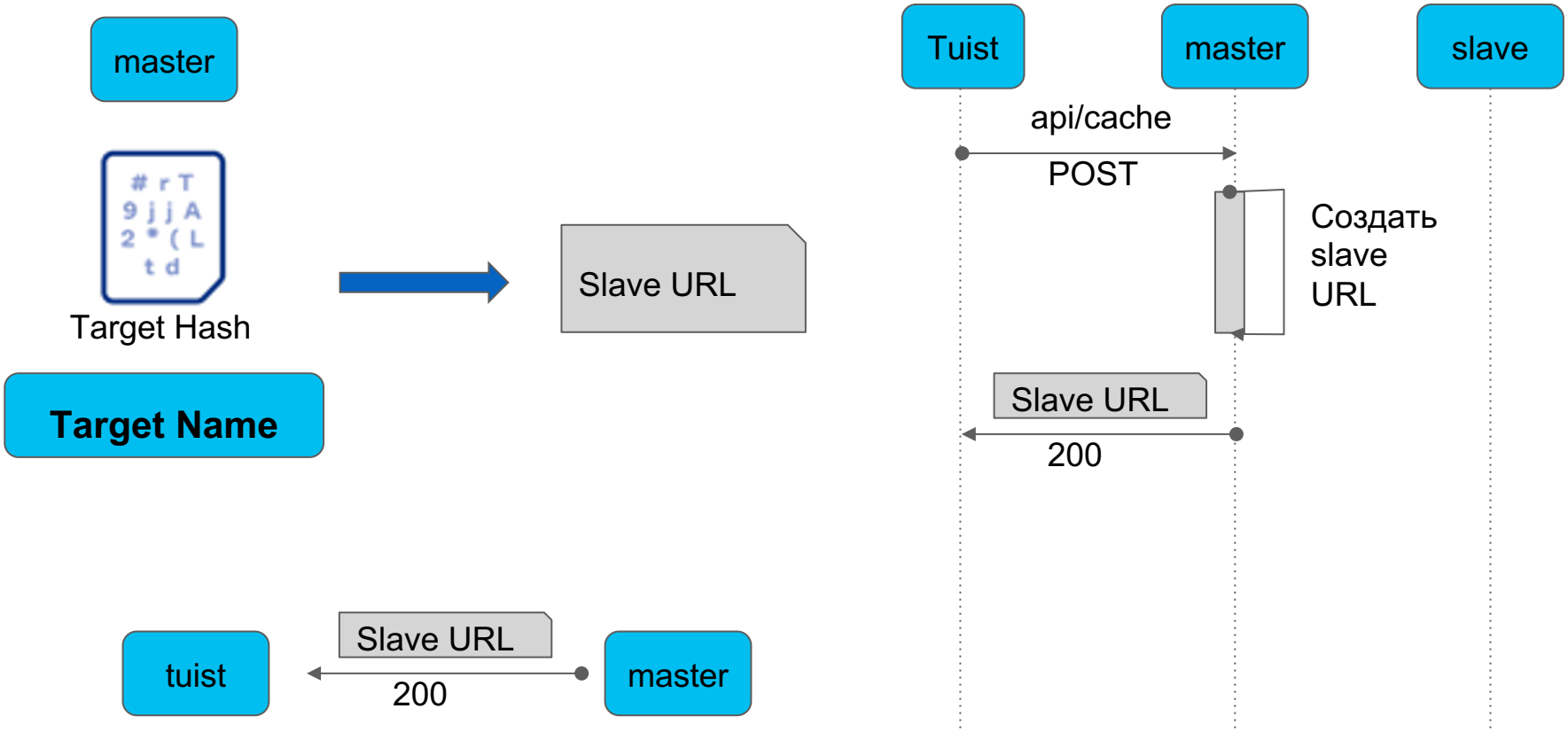
Сборка таргетов для которых не нашлось кеша



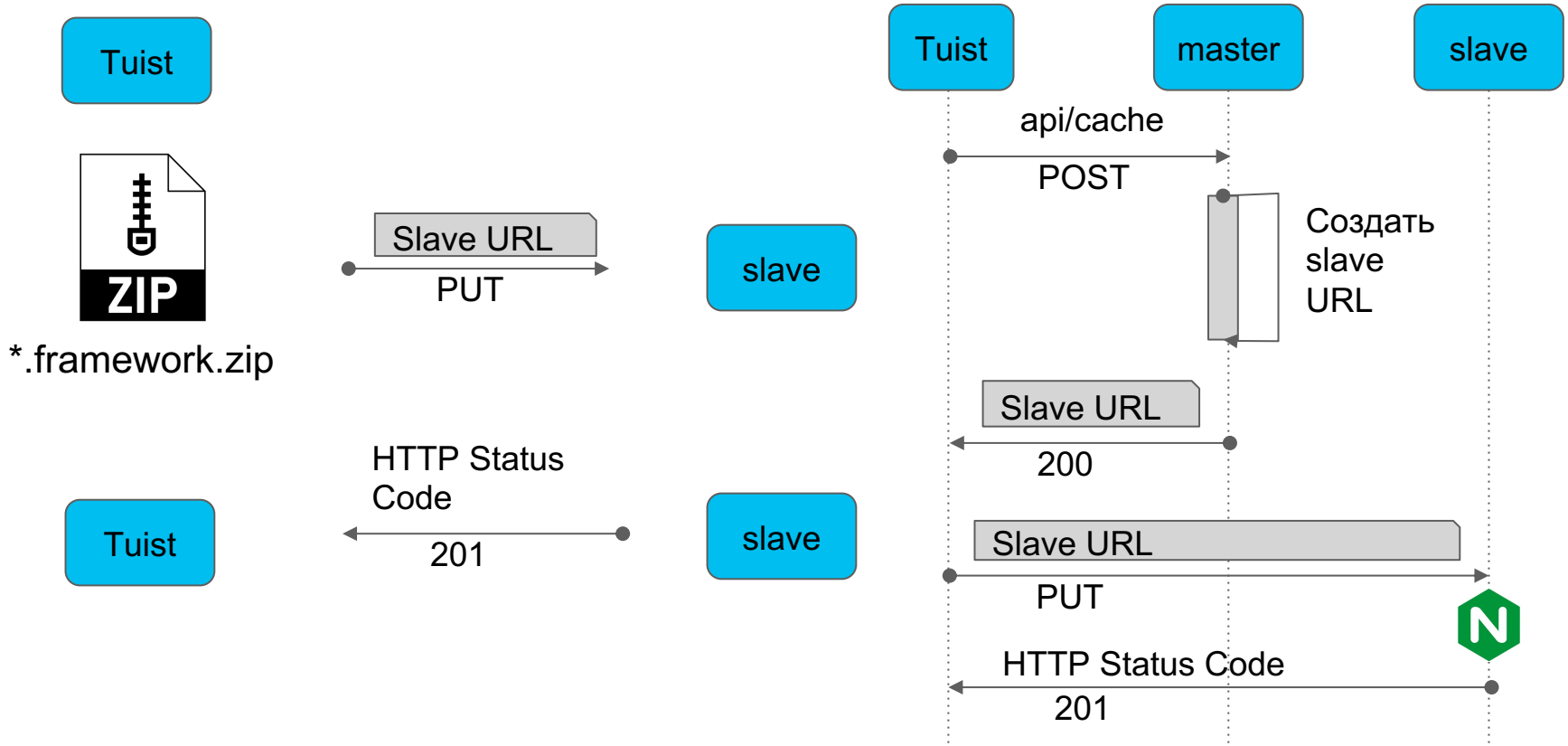
Прогрев кеша
Уточнение адреса выгрузки артефакта



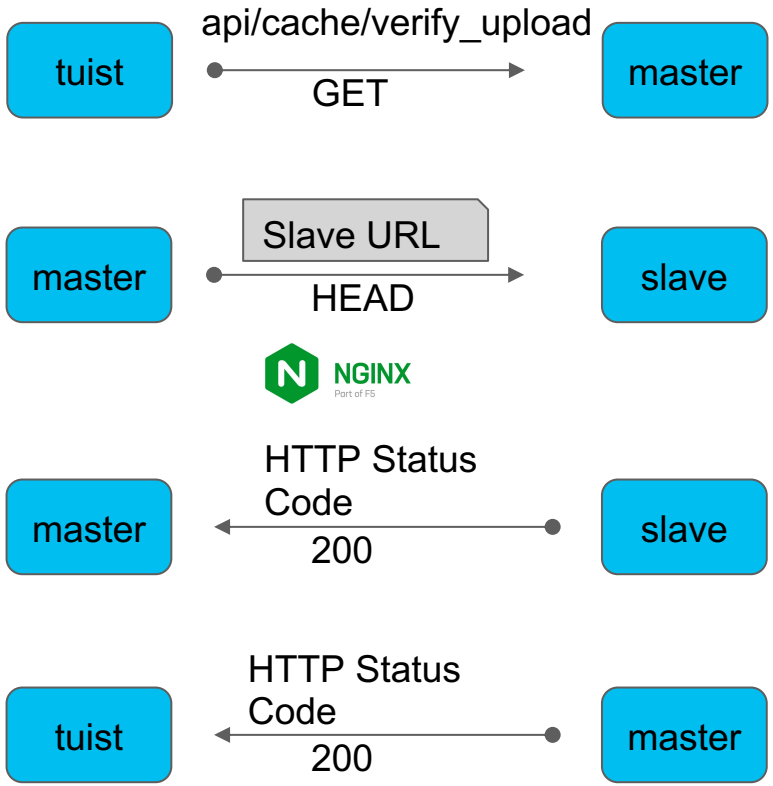
Прогрев кеша
Уточнение адреса выгрузки артефакта

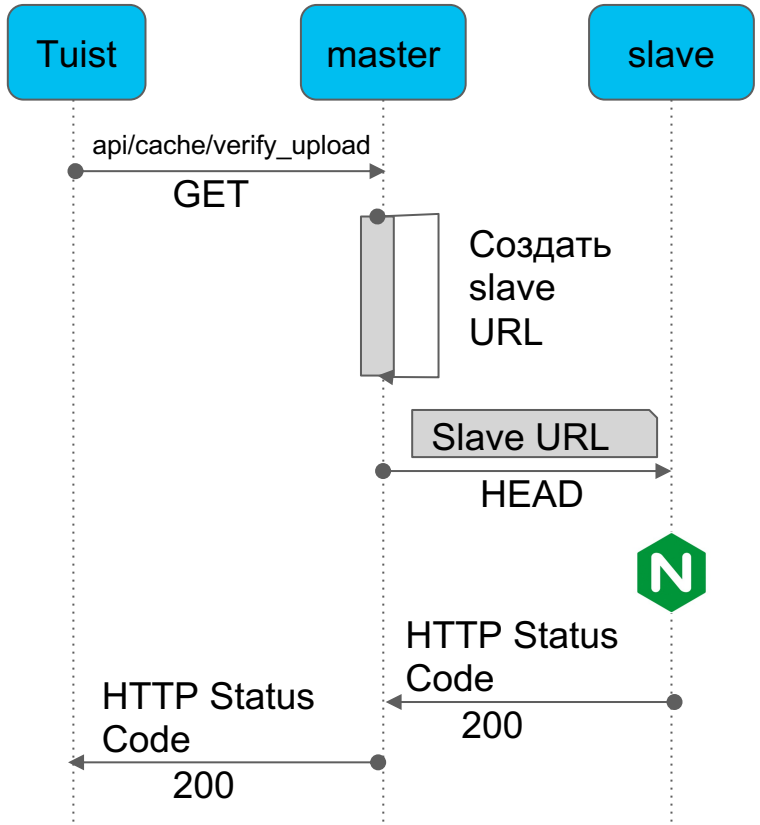


Прогрев кеша
Выгрузка артефактов на slave сервер

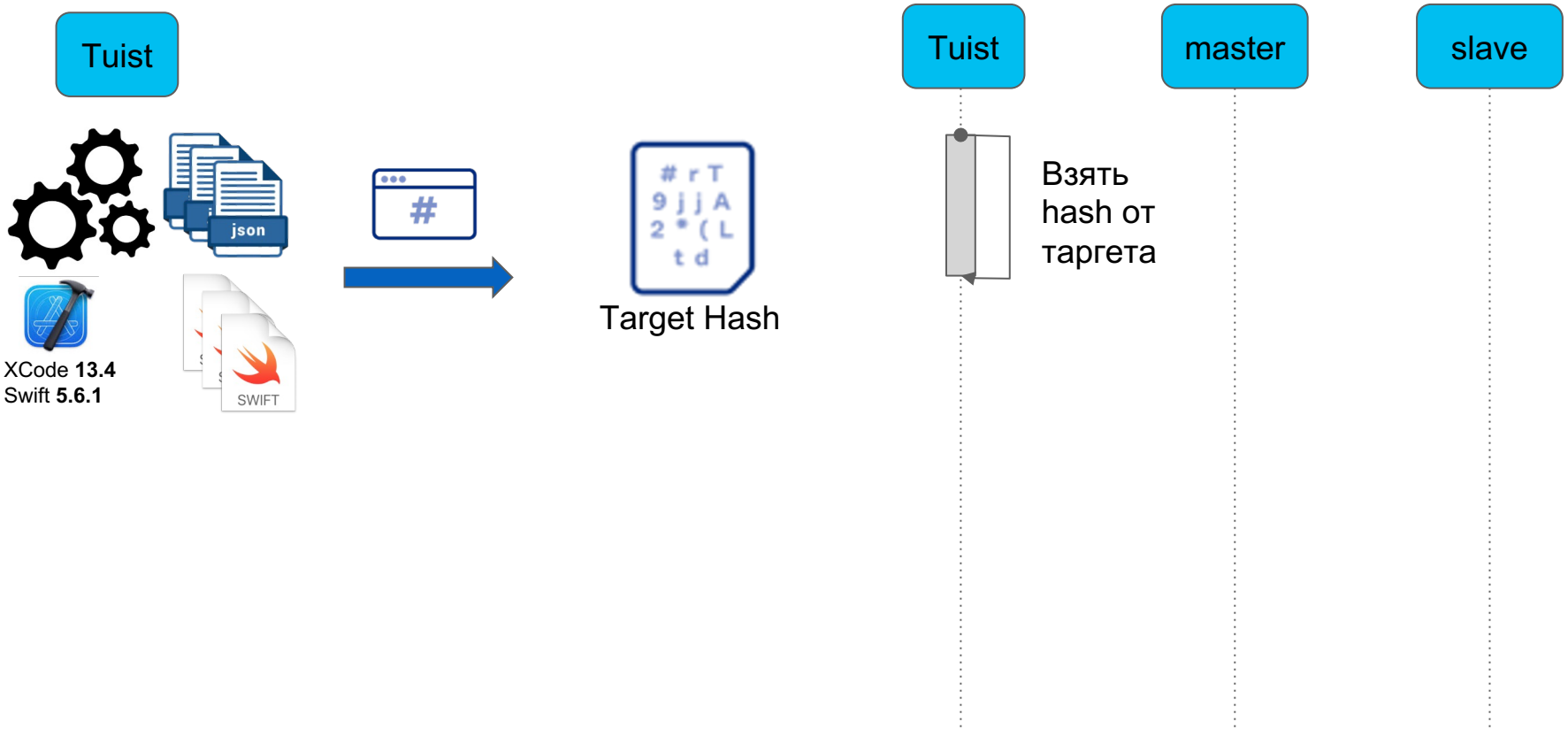


Прогрев кеша
Проверка успешности выгрузки

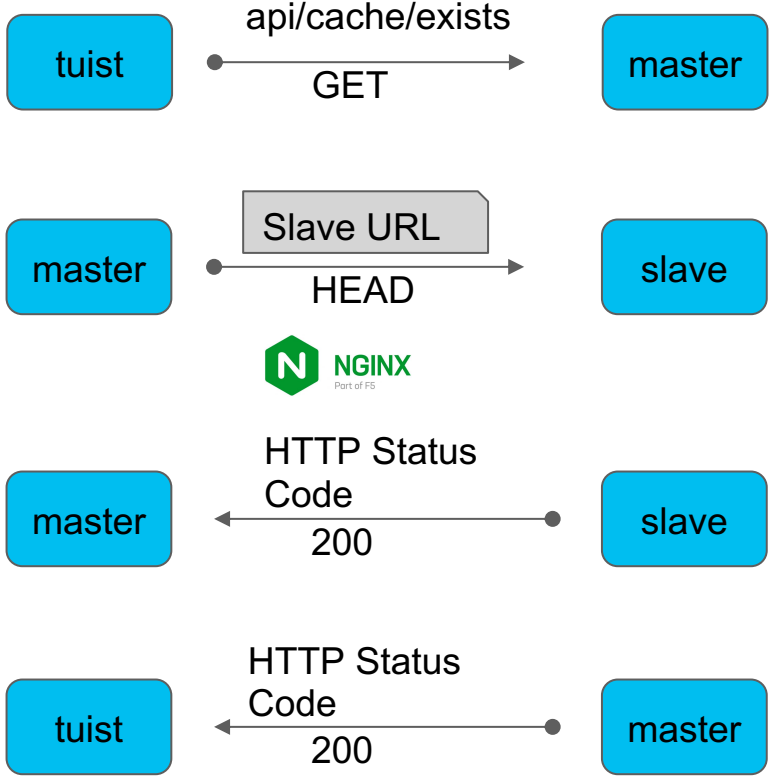


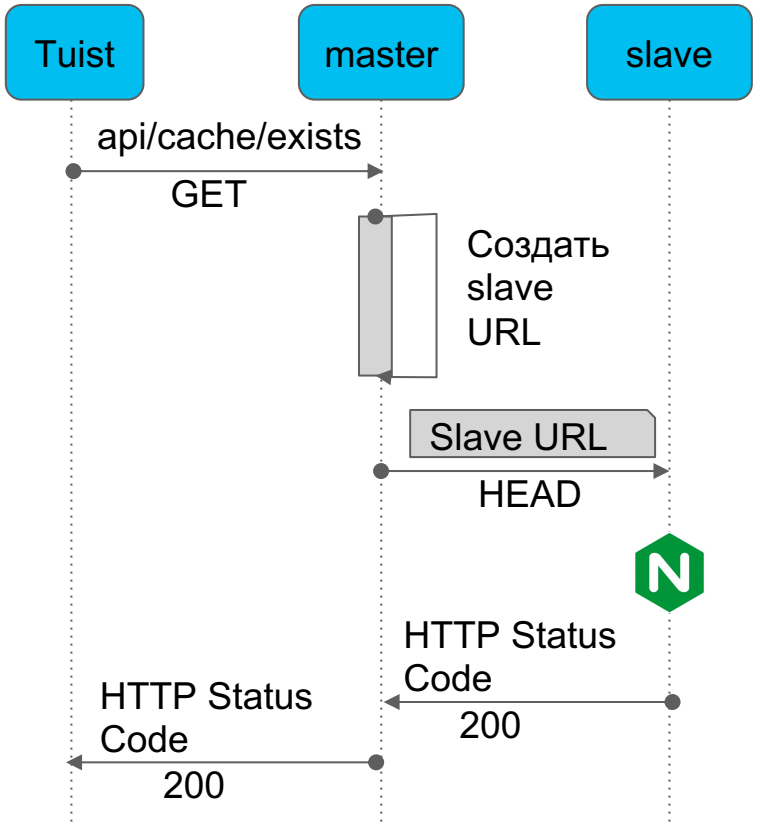


Генерация проекта у разработчика
Взятие hash от таргета

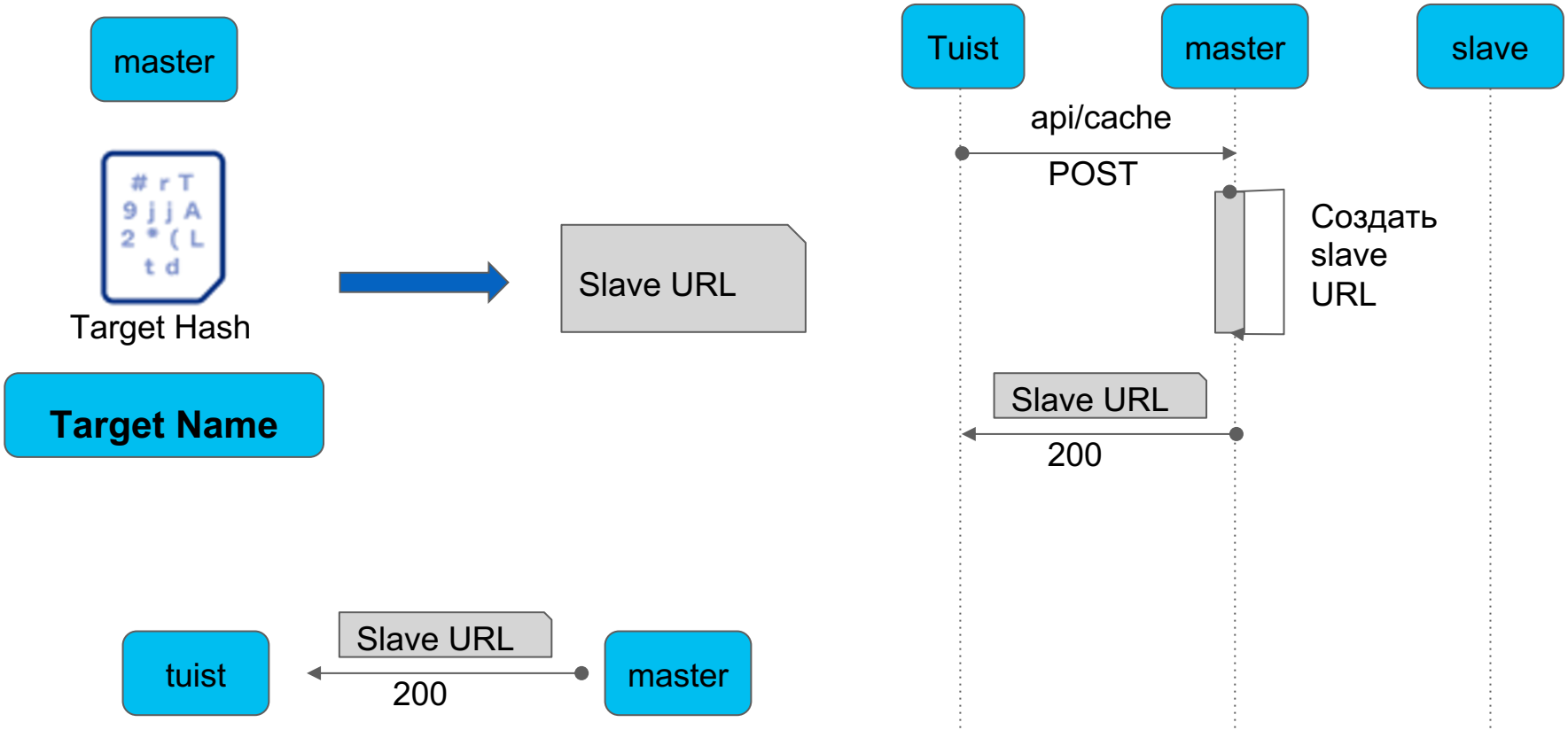


Генерация проекта у разработчика
Проверка наличия кеша

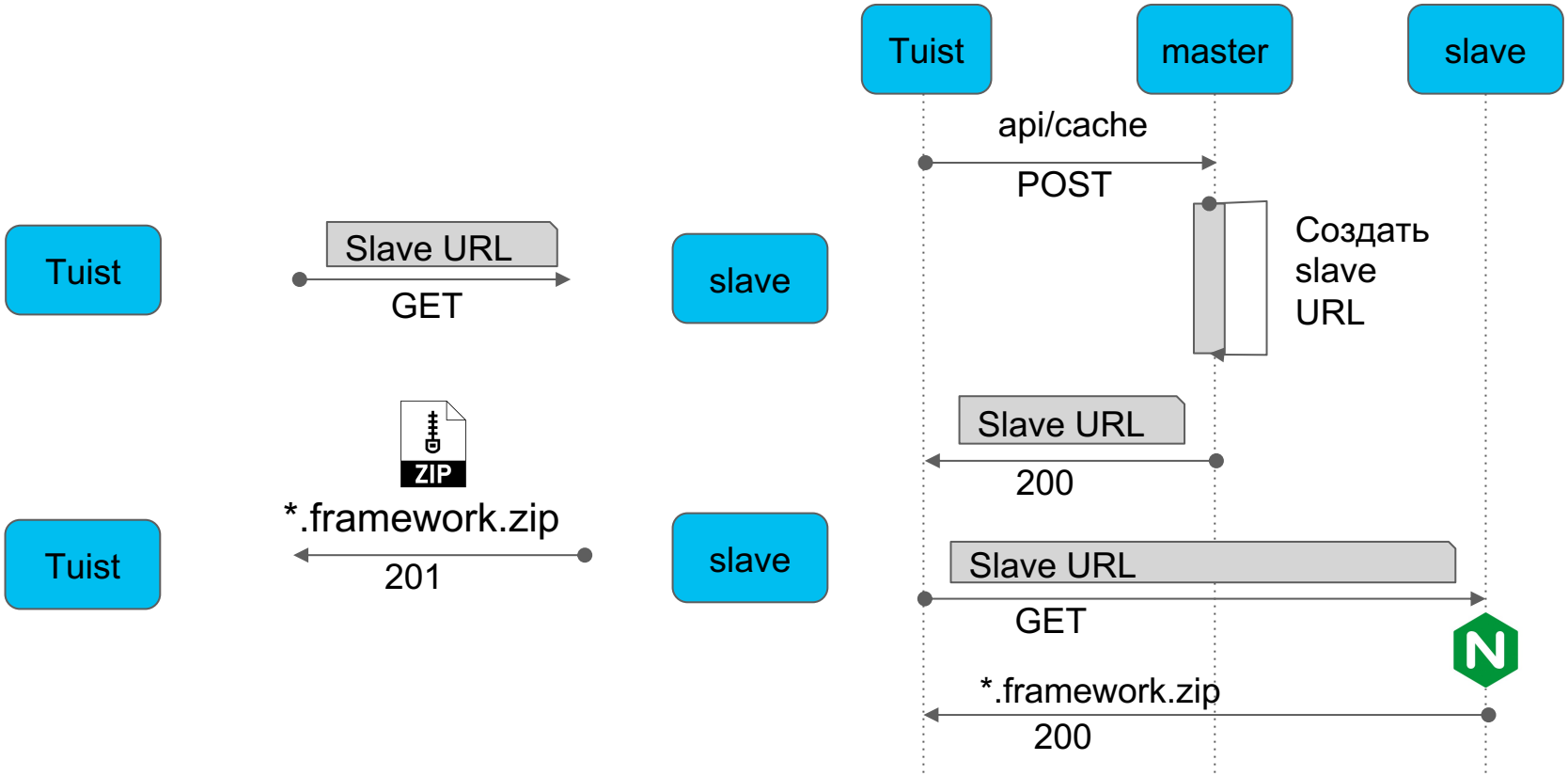




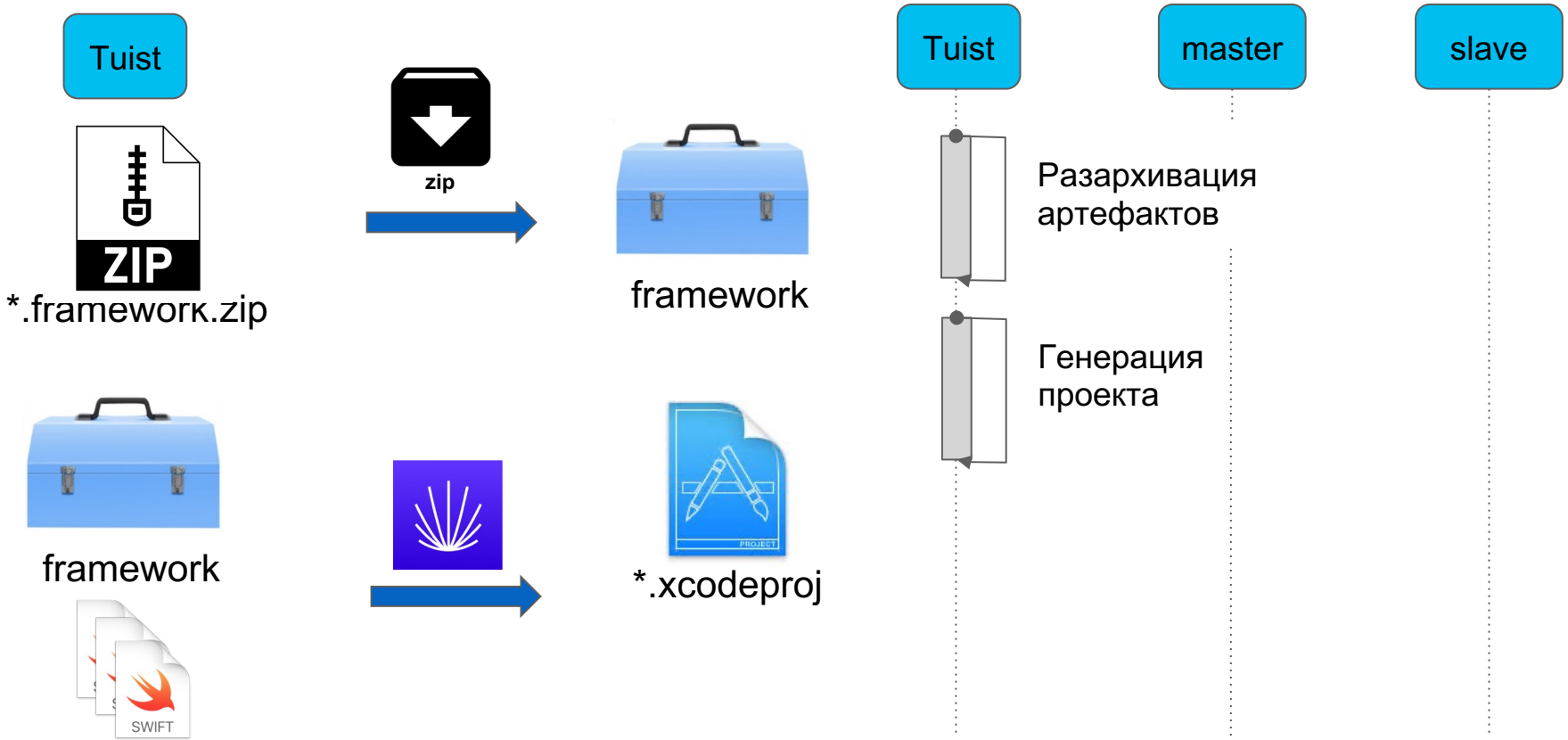
Генерация проекта у разработчика
Уточнение адреса скачивания артефакта



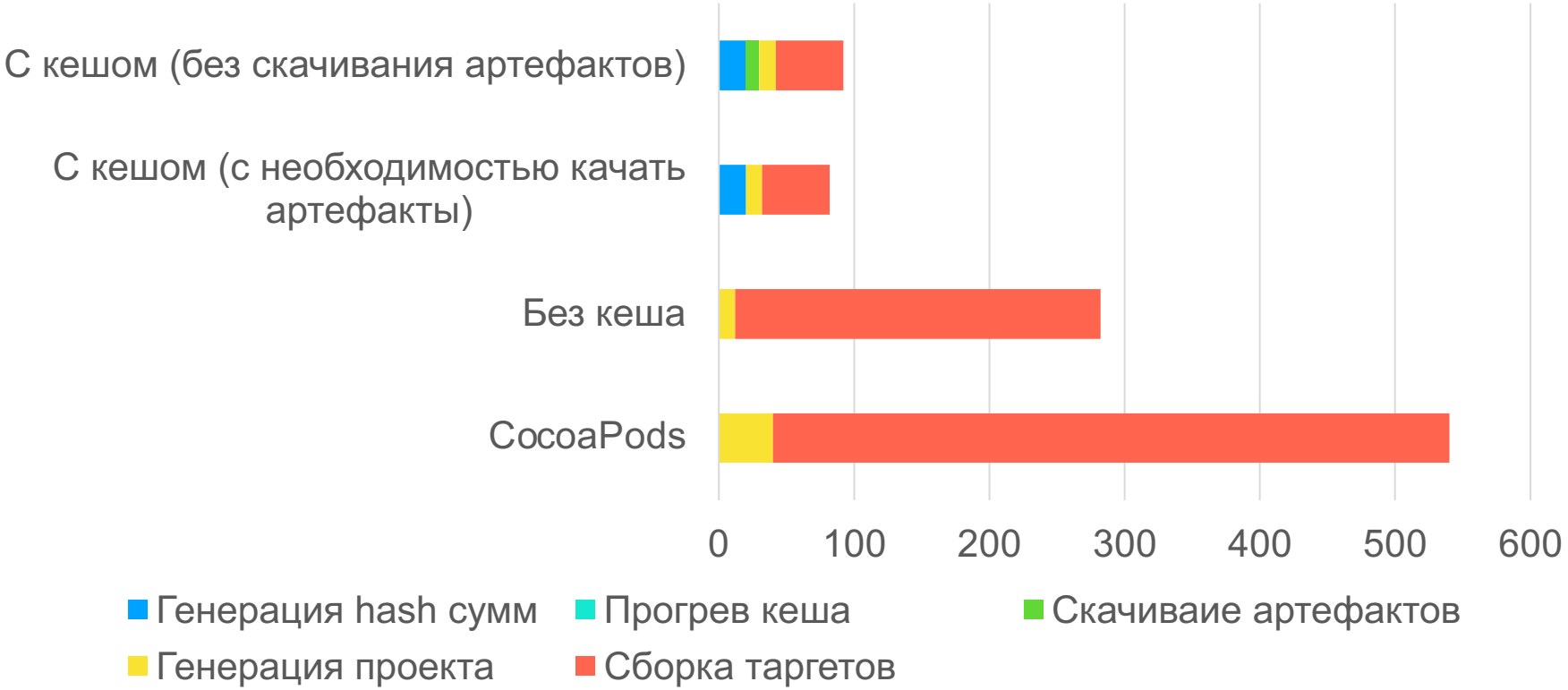
Генерация проекта у разработчика
Выгрузка артефактов на машину



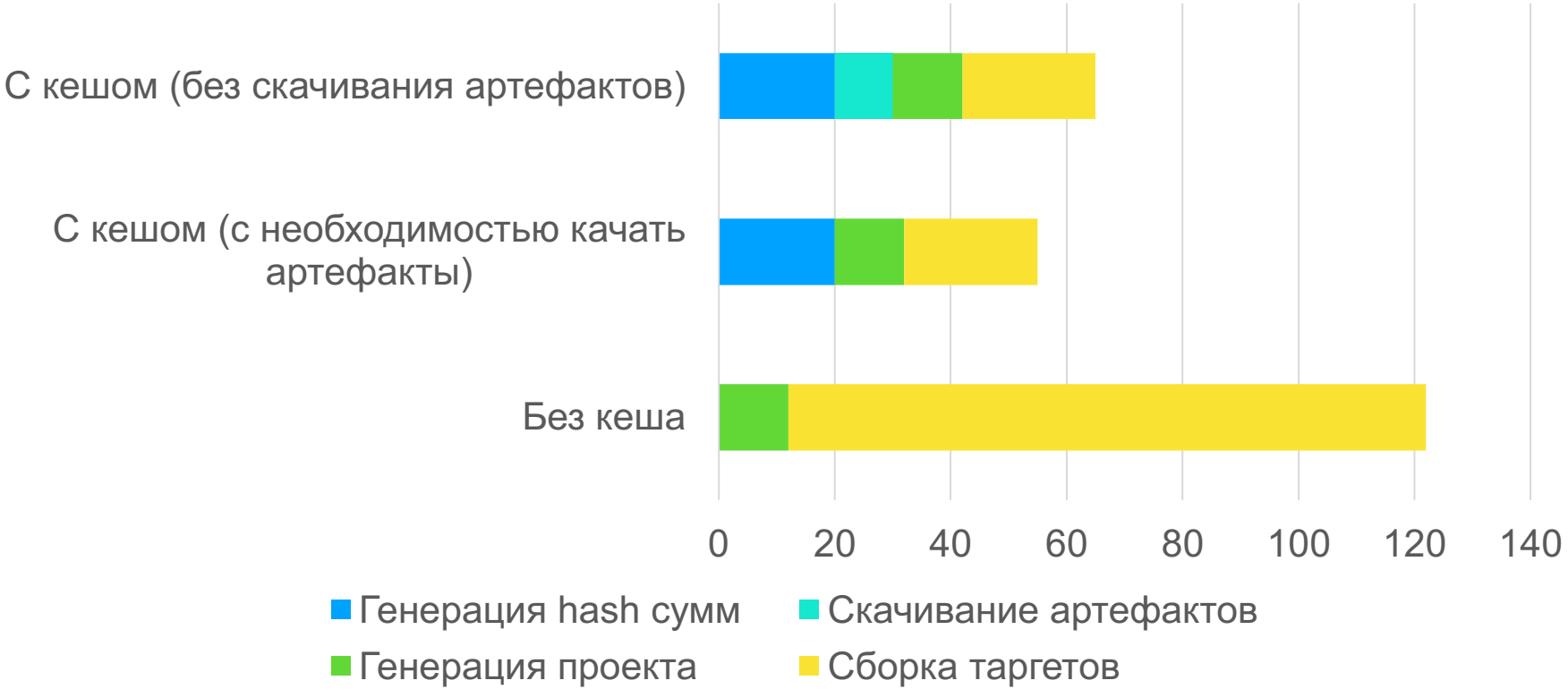
Генерация проекта у разработчика



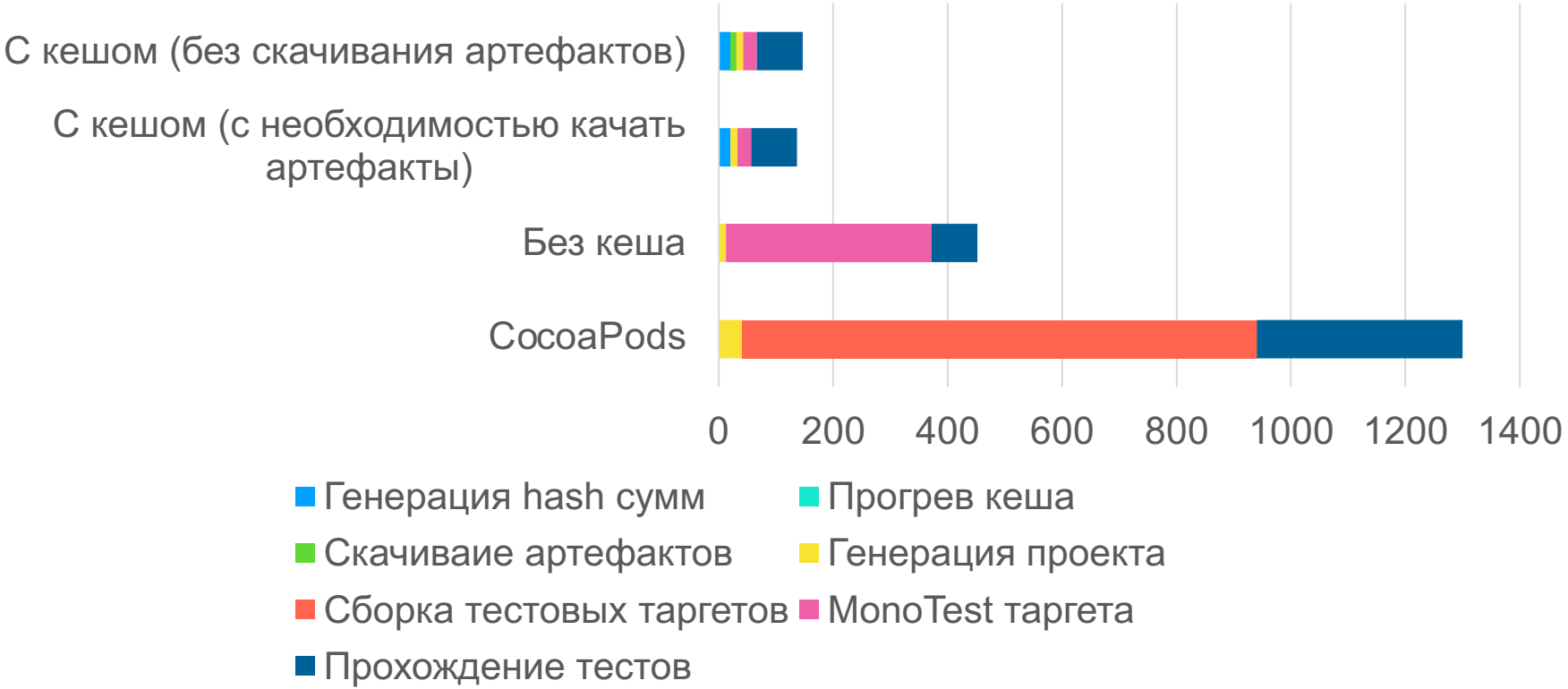
Бенчмарки
Основной таргет + 1 фича



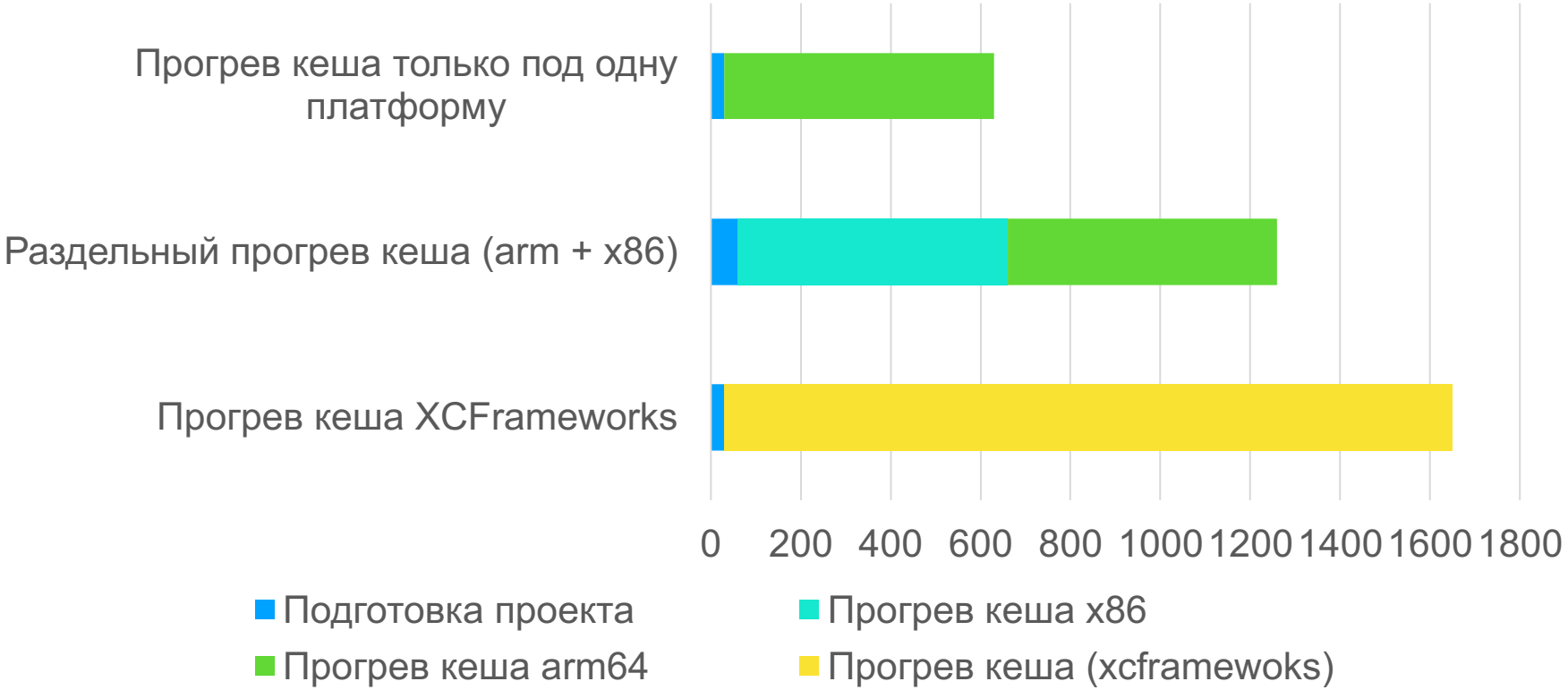
Бенчмарки
Feature-App + 1 фича



Бенчмарки
MonoTest таргет



Бенчмарки
Прогрев кеша



Типовые проблемы Tuist и их решения

5

Чистка манифестов
Кеш описания проекта

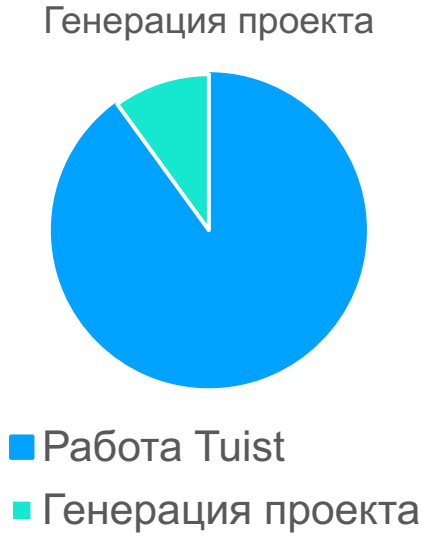
Проблема

Tuist кеширует описание проекта, поэтому использование в рамках генерации сторонних ресурсов, которые напрямую не видны в проекте, приводит к потере изменений

Решение

- Использовать swift файлы для описания таргетов и затягивать их в проект Tuist
- Перед каждой генерацией проекта чистить кеш манифестов

tuist clean manifests



Ипользование xcframeworks для хранения кеша

Проблема

Мнимая потеря кеша при использовании флагов **--xcframeworks**. Т.е. Tuist не применяет кеш при построении проекта хотя он есть

Решение

При использовании флага **--xcframeworks** его необходимо передавать и при прогреве кеша и при генерации проекта.

Поддержка обособленной сборки таргетов

Проблема

При динамической линковке возможно возникновение ситуаций когда один модуль импортирует другой, но при этом явно от него не зависит.

Решение

Необходимо тщательно проверять зависимости ваших модулей друг от друга. Можно использовать флаги отключения автолинковки или использовать статическую линковку

-Xfrontend -disable-autolink-framework

При возникновении проблем с импортами при прогреве кеша стоит построить граф проекта и проверить связи модуля.

Деградация перфоманса Tuist из-за сложного графа связей

Проблема

Рост количества связей в граффе может приводить (по субьективным ощущениям) к экспоненциальной деградации времени генерации проекта.

Решение

По возможности стоит держать число связей между модулями на минимальном уровне и максимально полагаться на транзитивный поиск зависимостей.

При помощи команды «граф» можно проверять, что один модуль дважды не линкуется в проект

Runtime скрипты в таргетах

Проблема

Наличие runtime скриптов делает таргеты некешируемыми. Это может особенно повлиять на проект с модульным DI, где каждый продуктовый модуль самостоятельно генерирует свой DI код при сборке.

Решение

При работе с кешом можно передавать специальный кастомный флаг, опираясь на который при генерации проекта все runtime скрипты будут выноситься в отдельный sh файл и запускаться перед сборкой проекта.

Оптимизация логов при прогреве кеша

Проблема

Может показаться, что использование xcpretty при генерации кеша тормозит процесс его сборки. После этого вы можете начать тратить время и ресурсы на поиск способов отключить логи.

Решение

Разбор исходников показывает, что Tuist:

- получает логи на отдельной очереди;
- собирает их в пачки;
- и передает их в xcpretty опять же на отдельной очереди.

Наши попытки отключить логи в конечном счете увенчаличь успехом, но прироста скорости сборки мы не увидели.

Проблема сортировки зависимостей

Проблема

Использование Set для операций над формированием зависимостей модуля из-за негарантированного порядка элементов в Set может привести к тому, что ваш модуль всегда будет иметь разные хеш суммы при генерации.

Решение

При формировании Target-а всегда необходимо сортировать его зависимости.

Стоит учитывыть, что некоторые зависимости содержат абсолютные пути, которые будут отличаться на разных машинах. Поэтому стоит аккуратно выбирать признак, по которому сортируются зависимости.

Логирование и дебагинг

Проблема

У Tuist довольно большие проблемы с дебагингом и логированием:

- Точки останова не работают;
- При критических ошибках принты не выводят данные в консоль.

Решение

Запись данных в лог файл вместо консоли позволяет решать проблему сбора логов. Это хоть как-то позволит вам отлаживать свой Tuist код

Итоги

6

Перед тем как начинать миграцию на Tuist взвешивайте риски и проверяйте свою мотивацию

- Что именно вас не устраивает в текущей системе сборки?
- Решит ли Tuist эти задачи?
- Даже если решит, то оправдаются ли затраты времени на переход?

Убедитесь, что все простые способы улучшить ситуацию закончились

Если вы используте модульность на базе CocoaPods, попробуйте сначала Rugby.

Эта утилита БЕЗ какого-либо изменения вашего кода даст вам гибкую систему кеширования таргетов.

Инструмент активно развивается и поддерживается.

Rugby

Миграция не будет легкой, но пройдите этот путь до конца

Tuist - это:

- Супергибкое описание проекта;
- Большой набор инструментов для удобных рефакторингов и управления модульностью;
- Понятный и простой код;
- Локальный и сетевой кеш;
- Весомое сокращение времени сборки проекта.

Важно помнить, что Tuist это не серебрянная пуля на все случаи жизни и имеет собственные недостатки

Оно того стоило!!!

Спасибо за внимание

Материалы

- https://github.com/tuist/tuist
- https://docs.tuist.io/tutorial/get-started