Buck

📌 `Estudo` sobre o buck

Pre requisitos

ant

python

watchman

java8

xcode build tools

```
brew install ant python git watchman
```

🗝 Ensumo

Links de referência

site oficial: https://buck.build/setup/getting_started.html

Tutorial: https://yasminbenatti.com/2020/05/04/buck-getting-started-building-buck-locally-and-running-the-example-app/

Project Sample referência (melhor que a versão do facebook) https://github.com/airbnb/BuckSample

Medium: https://bobbyprabowo.com/post/buck_build_system_on_ios_tutorial/, https://medium.com/airbnb-engineering/building-mixed-language-ios-project-with-buck-8a903b0e3e56

ProgrammerSought: https://www.programmersought.com/article/21431025583/

📚 Revisão processo de build do iOS

https://bignerdranch.com/blog/manual-swift-understanding-the-swift-objective-c-build-pipeline/

In short, the build process includes these steps:

1. Passing bridging header files (maintained by the author) and Swift source files to the `swift` tool, generating two kinds of files:a) `.o`, machine code files for generating the final executable binary.b)`-Swift.h`, containing all classes and interfaces defined in Swift code. These can be imported explicitly in Objective-C files that want to use functionality defined in Swift code.
2. Passing all Objective-C source files and `-Swift.h` files to `clang` tool, it generating `.o` files for each Objective-C file.
3. Passing all `.o` files to `ld` command, which will link all machine code files and generate the final executable.

Xcode vs Buck

Xcode builds each module independently, producing dynamically-linked frameworks.

For a particular module M, the executable binary and related resources/assets end up under App.app/Framework/M.framework in the final app folder.

Buck treats these modules as static libraries, linking them all together and producing a single executable binary.

This approach can effectively reduce the binary size since:

a) If multiple modules are using the same resource/asset, it doesn't need to copy the same file to each *.framework folder.

b) It can strip more unused symbols, since all libraries are linked together statically.

📚 Key iOS files

📌 `Arquivos de configuração importante do buck`

ios/BUCK:

The build file is what makes Buck work. It defines all the build rules for your source code. A build rule may also include dependencies (via deps), which may be from other build files.

Arquivo de build que faz o Buck funcionar. Define todas as regras de build para o seu projeto. Uma regra de build também inclui as dependências, que pode ser de outros arquivos de build

.buckconfig:

A .buckconfig file allows for various flag and alias settings for any project (even beyond iOS) within the root directory.

https://buck.build/files-and-dirs/buckconfig.html

O arquivo que permite a configuração de flag e configurações de `alias` para qualquer projeto dentro da diretório raiz.

Esse arquivo deve estar na raiz do diretório.

> The .buckconfig file uses the INI file format. That is, it is divided into sections where each section contains a collection of key names and key values

For example, choosing swift and iOS versions, enabling python parser for BUCK files, choosing Xcode as the IDE and optimization flags. In the file, they would look more and less like this:

```
[swift]
  version = 5

[apple]
  iphonesimulator_target_sdk_version = 11.0
  iphoneos_target_sdk_version = 11.0

[parser]
  polyglot_parsing_enabled = true

[project]
  ide = xcode

[custom]
  optimization = -Onone
```

BUCK build files

Definem regras que o projeto irá seguir para os builds. Eram escritos em python mas agora é feito em skylark.

> You can still use Python by setting the config polyglot_parsing_enabled

Install Buck

```
brew update
brew tap facebook/fb
brew install --HEAD buck
```

BUCK Rules

apple_bundle

An apple_bundle() rule takes an Apple binary and all of the resources and asset catalogs in the rule's transitive dependencies and generates a bundle containing all of those files.

Optionally the generated bundle can also be signed using specified provisioning profiles.

▼ **Arguments**

- `name` (required)

  The *short name* for this build target.

- `deps` (defaults to `[]`)

  A list of dependencies of this bundle as build targets. You can embed application extensions by specifying the extension's bundle target. To include a WatchKit app, append the flavor `#watch` to the target specification. Buck will automatically substitute the appropriate platform flavor (either `watchsimulator` or `watchos`) based on the parent.

- `product_name` (required)

  The name of the resulting bundle and binary. The setting behaves like PRODUCT_NAME Xcode build setting. For example, if your rule is named "MyApp" and extension is "app", by default buck will generate MyApp.app bundle. But if you will set product name to "SuperApp", bundle will get "SuperApp.app" name.

- `extension` (required)

The extension of the generated bundle. For example `'app'` for an application bundle or `'appex'` for an application extension bundle.

- `binary` (required)

  A build target identifying an `apple_binary()` rule or an `apple_library()` rule whose output will be used as the main executable binary of the generated bundle. The required rule type depends on the value in the `extension` attribute. For example, application bundles expect a binary (e.g. `'//Apps/MyApp:MyApp'`), application extension bundles expect a shared library (e.g. `'//Libraries/MyLibrary:MyLibrary#shared'`).

- `info_plist` (required)

  A path to an `Info.plist` file that will be placed in the bundle. The specified file will be processed by substituting variable names with their values (see `info_plist_substitutions` for more information).

- `info_plist_substitutions` (defaults to `{}`)

  A dictionary that assigns variable names to their values. It is used for variable substitution when processing the file specified in `info_plist`. For example if this argument is set to `{'VAR': 'MyValue'}`, then each occurrence of `$(VAR)` or `${VAR}` in the file will be replaced by `MyValue`.

- `tests` (defaults to `[]`)

  List of build targets that identify the test rules that exercise this target. Note that non `apple_test` targets will not be included in generated projects due to Xcode's limitations but will still work with `buck test`.

- `asset_catalogs_compilation_options` (required)

  A dict holding parameters for asset catalogs compiler (actool). Its options include:

  - `notices` (defaults to `True`)
  - `warnings` (defaults to `True`)
  - `errors` (defaults to `True`)
  - `compress_pngs` (defaults to `True`)
  - `optimization` (defaults to `'space'`)

    - `output_format` (defaults to `'human-readable-text'`)
    - `extra_flags` (defaults to `[]`)
- `ibtool_flags` (defaults to `[]`)

  List of flags to be passed to ibtool during interface builder file compilation.

- `visibility` (defaults to `[]`)

  List of build target patterns that identify the build rules that can include this rule as a dependency, for example, by listing it in their `deps` or `exported_deps` attributes. For more information, see visibility.

- `licenses` (defaults to `[]`)

  Set of license files for this library. To get the list of license files for a given build rule and all of its dependencies, you can use `buck query`.

- `labels` (defaults to `[]`)

  Set of arbitrary strings which allow you to annotate a build rule with tags that can be searched for over an entire dependency tree using `buck query "attrfilter()"`.

apple_resource

An apple_resource() rule contains sets of resource directories, files and file variants that can be bundled in an application bundle.

https://buck.build/rule/apple_resource.html

▼ **Arguments**

- `name` (required)

  The *short name* for this build target.

- `dirs` (required)

  Set of paths of resource directories that should be placed in an application bundle.

- `files` (required)

Set of paths of resource files that should be placed in an application bundle.

- `variants` (defaults to `[]`)

  Set of paths of resource file variants that should be placed in an application bundle. The files mentioned here should be placed in a directory named `$VARIANT_NAME.lproj`, where `$VARIANT_NAME` is the name of the variant (e.g. `Base`, `en`). This argument makes it possible to use different resource files based on the active locale.

- `resources_from_deps` (defaults to `[]`)

  Set of build targets whose transitive `apple_resource`s should be considered as part of the current resource when collecting resources for bundles. Usually, an `apple_bundle` collects all `apple_resource` rules transitively reachable through apple_library rules. This field allows for resources which are not reachable using the above traversal strategy to be considered for inclusion in the bundle.

- `destination` (defaults to `resources`)

  Specifies the destination in the final application bundle where resource will be copied. Possible values: "resources", "frameworks", "executables", "plugins", "xpcservices".

- `codesign_on_copy` (defaults to `False`)

  Indicates whether the files specified in the files arg in this resource should be code signed with the identity used to sign the overall bundle. This is useful for e.g. dylibs or other additional binaries copied into the bundle. The caller is responsible to ensure that the file format is valid for codesigning.

- `visibility` (defaults to `[]`)

  List of build target patterns that identify the build rules that can include this rule as a dependency, for example, by listing it in their `deps` or `exported_deps` attributes. For more information, see visibility.

- `licenses` (defaults to `[]`)

Set of license files for this library. To get the list of license files for a given build rule and all of its dependencies, you can use `buck query`.

- `labels` (defaults to `[]`)

  Set of arbitrary strings which allow you to annotate a build rule with tags that can be searched for over an entire dependency tree using `buck query "attrfilter()"`.

apple_asset_catalog

https://buck.build/rule/apple_asset_catalog.html

▼ **Arguments**

- `name` (required)

  The *short name* for this build target.

- `dirs` (defaults to `[]`)

  Set of paths of Apple asset catalogs contained by this rule. All paths have to end with the `.xcassets` extension and be compatible with the asset catalog format used by Xcode.

- `app_icon` (defaults to `None`)

  An optional reference to a `.appiconset` containing a image set representing an application icon. (The extension itself should not be included.) This parameter may be specified at most once in a given `apple_bundle`'s transitive dependencies.

- `launch_image` (defaults to `None`)

  An optional reference to a `.launchimage` containing a image set representing an application launch image. (The extension itself should not be included.) This parameter may be specified at most once in a given `apple_bundle`'s transitive dependencies.

- `visibility` (defaults to `[]`)

  List of build target patterns that identify the build rules that can include this rule as a dependency, for example, by listing it in their `deps` or `exported_deps` attributes. For more information, see visibility.

- `licenses` (defaults to `[]`)

  Set of license files for this library. To get the list of license files for a given build rule and all of its dependencies, you can use `buck query`.

- `labels` (defaults to `[]`)

  Set of arbitrary strings which allow you to annotate a build rule with tags that can be searched for over an entire dependency tree using `buck query attrfilter()`.

```
apple_asset_catalog(
    name = "ExampleAppAssets",
    visibility = ["//App:"],
    app_icon = "AppIcon",
    dirs = ["Assets.xcassets"],
)
```

apple_binary

An apple_binary() rule builds a native executable—such as an iOS or OSX app—from the supplied set of Objective-C/C++ source files and dependencies.

https://buck.build/rule/apple_binary.html

▼ **Arguments**

- `name` (required)

  The *short name* for this build target.

- `srcs` (defaults to `[]`)

  The set of C, C++, Objective-C, Objective-C++, or assembly source files to be preprocessed, compiled, and assembled by this rule. We determine which stages to run on each input source based on its file extension. See the GCC documentation for more detail on how file extensions are interpreted. Each element can be either a string specifying a source file (e.g. `'foo/bar.m'`) or a tuple of a string specifying a source file and a list of compilation flags (e.g. `('foo/bar.m', ['-Wall', '-Werror'])`). In the latter case

the specified flags will be used in addition to the rule's other flags when preprocessing and compiling that file (if applicable).

- `platform_srcs` (defaults to `[]`)

  Platform specific source files. These should be specified as a list of pairs where the first element is an un-anchored regex (in java.util.regex.Pattern syntax) against which the platform name is matched, and the second element is either a list of source files or a list of tuples of source files and a list of compilation flags to be preprocessed, compiled and assembled if the platform matches the regex. See `srcs` for more information.

- `headers` (defaults to `[]`)

  The set of header files that are made available for inclusion to the source files in this target. These should be specified as either a list of header files or a dictionary of header names to header files. The header names can contain forward slashes (`/`). If a list of header files is specified, the headers can be imported with `#import "$HEADER_PATH_PREFIX/$HEADER_NAME"` or `#import "$HEADER_NAME"`, where `$HEADER_PATH_PREFIX` is the value of the target's `header_path_prefix` attribute, and `$HEADER_NAME` is the filename of the header file. If a dictionary is specified, each header can be imported with `#import "$HEADER_NAME"`, where `$HEADER_NAME` is the key corresponding to this file. In this case, the `header_path_prefix` attribute is ignored. In either case, quotes in the import statements can be replaced with angle brackets.

- `entitlements_file` (required)

  An optional name of a plist file to be embedded in the binary. Some platforms like `iphonesimulator` require this to run properly.

- `exported_headers` (defaults to `[]`)

  The set of header files that are made available for inclusion to the source files in this target and all targets that transitively depend on this one. These should be specified as either a list of header files or a dictionary of header names to header files. The header names can contain forward slashes (`/`). If a list of header files is specified, the headers can be imported with `#import "$HEADER_PATH_PREFIX/$HEADER_NAME"` or, if a header file

that belongs to the same rule is being imported, with `#import "$HEADER_NAME"`, where `$HEADER_PATH_PREFIX` is the value of the target's `header_path_prefix` attribute, and `$HEADER_NAME` is the filename of the header file. If a dictionary is specified, each header can be imported with `#import "$HEADER_NAME"`, where `$HEADER_NAME` is the key corresponding to this file. In this case, the `header_path_prefix` attribute is ignored. In either case, quotes in the import statements can be replaced with angle brackets.

- `header_path_prefix` (defaults to `name`)can be imported using following mappingDefaults to the short name of the target. Can contain forward slashes (`/`), but cannot start with one. See `headers` for more information.

  A path prefix when including headers of this target. For example, headers from a library defined using

  ```
  apple_library(
      name = "Library",
      headers = glob(["**/*.h"]),
      header_path_prefix = "Lib",
  )
  ```

  ```
  Library/SubDir/Header1.h -> Lib/Header1.h
  Library/Header2.h -> Lib/Header2.h
  ```

- `frameworks` (defaults to `[]`)

  A list of system frameworks that the code in this target uses. Each entry should be a path starting with `$SDKROOT` or `$PLATFORM_DIR` to denote that the rest of the path is relative to the root of the SDK used for the build or to the platform toolchain directory.

- `preprocessor_flags` (defaults to `[]`)

  Flags to use when preprocessing any of the above sources (which require preprocessing).

- `exported_preprocessor_flags` (defaults to `[]`)

Just as `preprocessor_flags`, flags to use when preprocessing any of the above sources (which require preprocessing). However, unlike `preprocessor_flags`, these preprocessor flags are also used by rules that transitively depend on this rule when preprocessing their own sources.

- `compiler_flags` (defaults to `[]`)

  Flags to use when compiling any of the above sources (which require compilation).

- `platform_compiler_flags` (defaults to `[]`)

  Platform specific compiler flags. These should be specified as a list of pairs where the first element is an un-anchored regex (in java.util.regex.Pattern syntax) against which the platform name is matched, and the second element is a list of flags to use when compiling the target's sources. See `compiler_flags` for more information.

- `linker_extra_outputs` (defaults to `[]`)

  Declares extra outputs that the linker emits. These identifiers can be used in `$(output ...)` macros in `linker_flags` to interpolate the output path into the linker command line. Useful for custom linkers that emit extra output files.

- `linker_flags` (defaults to `[]`)

  Flags to add to the linker command line whenever the output from this rule is used in a link operation, such as linked into an executable or a shared library.

- `exported_linker_flags` (defaults to `[]`)

  Flags to add to the linker command line when the output from this rule, or the output from any rule that transitively depends on this rule, is used in a link operation.

- `platform_linker_flags` (defaults to `[]`)

  Platform-specific linker flags. This argument is specified as a list of pairs where the first element in each pair is an un-anchored regex against which the platform name is matched. The regex should

use `java.util.regex.Pattern` syntax. The second element in each pair is a list of linker flags. If the regex matches the platform, these flags are added to the linker command line when the output from this rule is used in a link operation.

- `link_style` (defaults to `static`)

  Determines whether to build and link this rule's dependencies statically or dynamically. Can be either `static`, `static_pic` or `shared`.

- `target_sdk_version` (defaults to `None`)

  The minimum OS version that the library target should support, overriding the minimum set in `.buckconfig`. When set, Buck will automatically add flags to both Objective-C and Swift compilation that will allow the use of the new APIs without guarding code inside availability checks.

- `tests` (defaults to `[]`)

  List of build targets that identify the test rules that exercise this target. Note that non `apple_test` targets will not be included in generated projects due to Xcode's limitations but will still work with `buck test`.

- `extra_xcode_sources` (defaults to `[]`)

  When the project is generated, this is the list of files that will added to the build phase "Compile Sources" of the given target.

- `extra_xcode_files` (defaults to `[]`)

  When the project is generated, this is the list of files that will added to the project. Those files won't be added to the build phase "Compile Sources".

- `visibility` (defaults to `[]`)

  List of build target patterns that identify the build rules that can include this rule as a dependency, for example, by listing it in their `deps` or `exported_deps` attributes. For more information, see visibility.

- `licenses` (defaults to `[]`)

  Set of license files for this library. To get the list of license files for a given build rule and all of its dependencies, you can use `buck query`.

- `labels` (defaults to `[]`)

> Set of arbitrary strings which allow you to annotate a build rule with tags that can be searched for over an entire dependency tree using `buck query attrfilter()`.

First build

Depois de configurar o arquivo BUCK, que fica dentro da pasta do projeto.

Rode o comando abaixo para ajustar o xcode de acordo com o buck

To adjust xcode properly with buck, go to root folder and type

```
buck project --ide xcode
```

Info.plist

Quando rodamos o buck pela primeira vez ele tenta ler o info.plist porem esse arquivo possui variáveis dinâmicas. Desta forma podemos deixar elas fixas ou criar uma forma do buck ler e substituir elas quando for fazer o build.

Unrecognized plist variable: $(PRODUCT_BUNDLE_IDENTIFIER)

External dependencies

Para adicionar o carthage ou cocoapods podemos definir na parte de prebuild_frameworks

```
prebuilt_frameworks = [
    "//Carthage:AFNetworking",
    ...
]
```

When a pod only provides a compiled binary (e.g. libSample.a), we use the prebuilt_cxx_library build rule.

When a pod provides a framework file, we use the prebuilt_apple_framework build rule. More examples of this build rule can be found here.

Build scripts

Scripts de build como Swiftgen ou. xcodegen. Pre e post build scripts podem ser declarados usando o xcode_prebuild_script and xcode_postbuild_script

```
xcode_prebuild_script(
    name = "Hello_World",
    cmd = '"${SRCROOT}/../scripts/sample.sh"',
    ...
)
```

The App

Defining the app has a few steps. Defining the library, bundle, binary, and the package.

```
apple_library(
    name = "ExampleAppLibrary",
    ...
)

apple_bundle(
    name = "ExampleApp",
    ...
)

apple_binary(
    name = "ExampleAppBinary",
    ...
)

apple_package(
    name = "ExampleAppPackage",
    bundle = ":ExampleApp",
)
```

The workspace

To use Xcode, we need to define a workspace that will later be called when running make project

```
xcode_workspace_config(
    name = "workspace",
    workspace_name = "ExampleApp",
    src_target = ":ExampleApp",
    ...
)
```

Airbnb Sample App

O projeto de example do airbnb é mais atual que o do facebook (2015).

Além disso o iFood usou os arquivos de configuração do projeto deles para usar na empresa.

https://github.com/airbnb/BuckSample

> All this scripts are under /Makefile. Check it out for more details.

Run project from Xcode

Para criar um projeto do Xcode use o comando abaixo:

At my company, we mostly use this option, except for CI. We still get the benefits of using the IDE and it's easier for developers.

make project

🏆 Próximos passos

- ☑️ ~~Resolver problemas de dependências (ruby gem e java) que podem estar dando erro~~

- [x] ~~Criar um projeto simples e fazer o build com o buck~~
- [ ] Problema de leitura do info.plist
- [ ] Criar um projeto com cocoapods e fazer o build com o buck
- [ ] Validar funcionamento com pod que geram framework