Living and Breathing the Apache Way

Contributing upstream with competitors, customers, and investors breathing down your neck

APACHECON
NORTH AMERICA
OCTOBER 3 - 6, 2022
NEW ORLEANS
LOUISIANA
WWW.APACHECON.COM

By Aleksander Vidakovic

Intro

We all know that **Fineract** as a **mission-critical core banking system** provides a difficult balance to strike when rapidly building a differentiated product on top of an open source project. While challenging to balance the need for **secrecy** and **time to market** for one’s solution versus the transparency and contribution to the open source project, when **upstream contribution** is done right it unlocks enormous economic value for not just you the innovator but the entire ecosystem as a whole.

This session will explore at a practical level the **culture**, **processes**, and **standards** that need to be in place at both an org level and a community level to ensure that individuals can effectively contribute to upstream open source codebase while efficiently maintaining their downstream solution. We know that putting into practice **the Apache Way** can be difficult and will share our firsthand experiences so others in the community can do the same and reap the economic benefits both internally as well as externally across the entire community.

Who’s working with Fineract’s Source Code

Audience:

- Fineract Committers
- Freelancers
- Commercial integrators
- Larger companies with own development resources
- Small companies
- DevOps people who deploy Fineract as-is, no source changes
- … a combination of all of the above

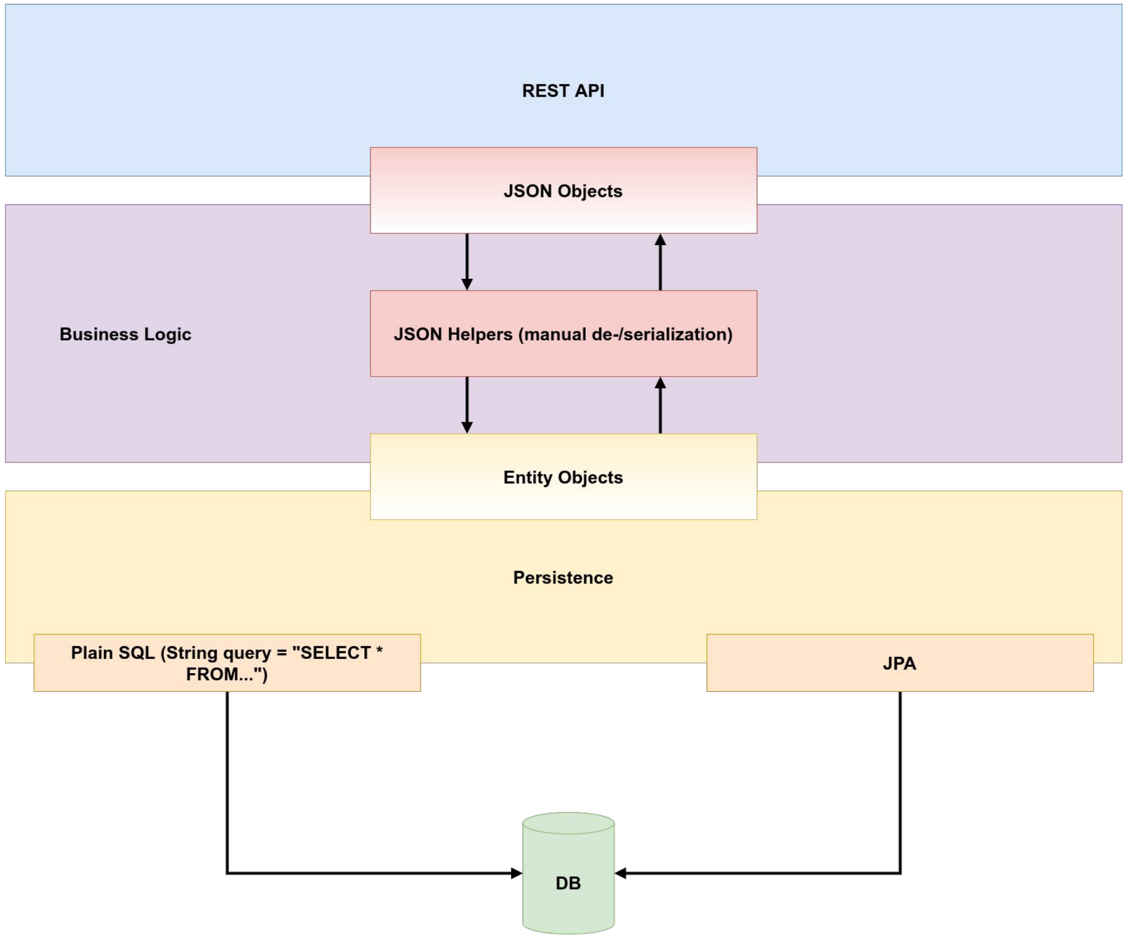
… with different **requirements**:

- **New users** that need help to set up their projects
- **Long-time users** that might be stuck with a very old version
- … with preferences for more **stable production releases**
- … or who want the **latest and greatest** and accept a certain risk that things might break

Where we are…

- **Monolithic** codebase
- No modules, libraries on **Maven Central**
- Need to **fork** upstream Git repository for private customizations
- Infrequent **releases** (once per year)
- No way to fully/partially **replace** existing **business logic** without refactoring (modules, “auto-configuration”)
- **Good package structure**, but **not** always following best practices for **clean architecture**
- Too many **dependencies** between **business logic** and **API** layer
- Two database query technologies (plain **SQL** in strings, JPA/**OQL**)
- Handcrafted SQL database independence tools (MySQL, PostgreSQL)
- This worked pretty well so far…

Fineract
Current Architecture (simplified)


REST API
JSON Objects
Business Logic
JSON Helpers (manual de-/serialization)
Entity Objects
Persistence
Plain SQL (String query = "SELECT * FROM...")
JPA
DB

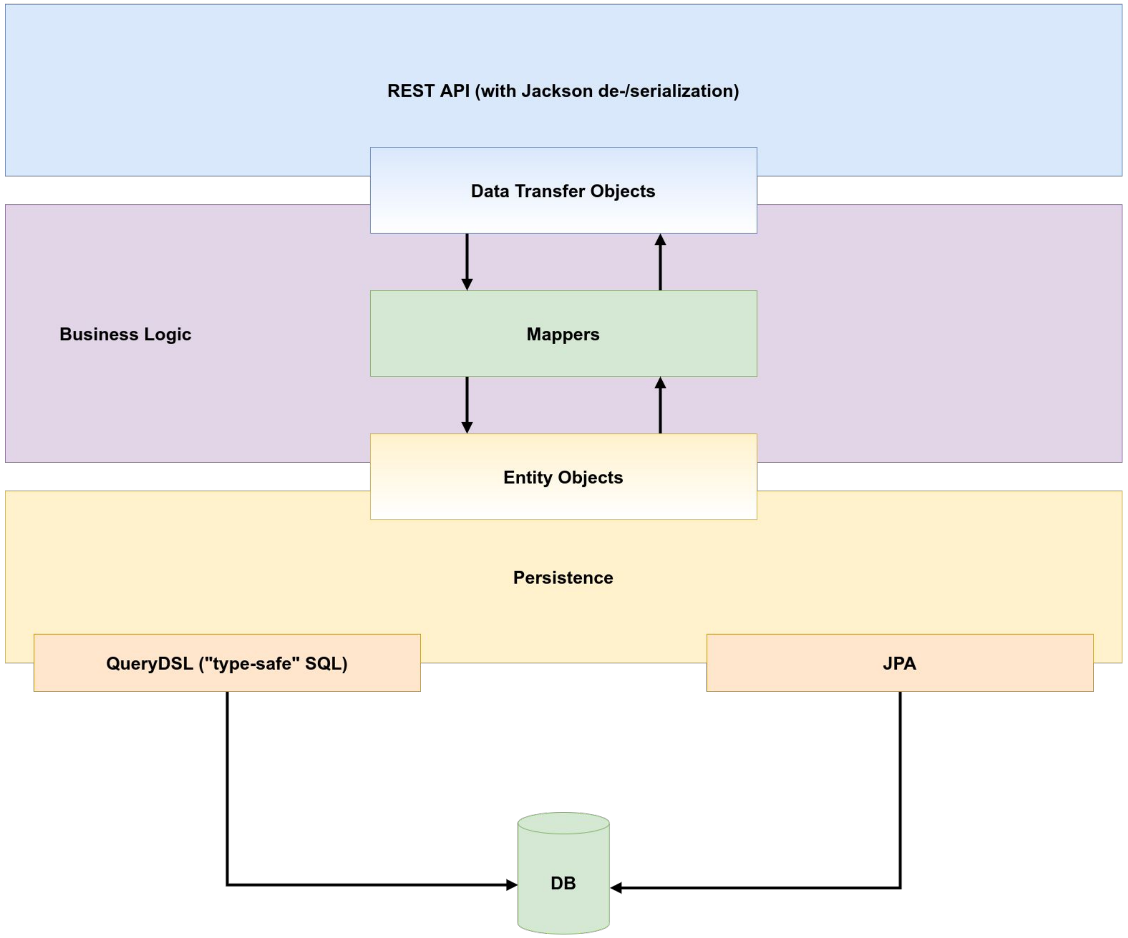
Resulting Anti-Patterns

- Create a **fork** based on the current **develop branch**, to get all recent fixes as soon as possible downstream risks downstream production instability
- Customization of original upstream business logic (“**Open Heart Surgery**”) risks **Git conflicts** when synchronizing downstream with latest changes from upstream
- **No clear separation** between private **customizations** (aka “secret sauce”) and **generally useful fixes/improvements** effectively slows down upstream development
- **Waiting too long to synchronize** downstream with upstream changes effectively risks a downstream hard fork

Where we are heading…

- Multiple **major** (feature-)**releases** per year
- **Hotfix**/maintenance releases whenever necessary (security, bugs; field test with Manoj and a large company)
- Maintenance **support** for the latest **two minor releases**
- In discussion: **long term support**
- Recommend to new users to base their **forks** on a **release tag**
- Proper **separation of architectural layers** (especially proper use of DTOs, JSON de-/serialization)
- **Modularization** will help in the future to avoid forks altogether
- Standardized way to fully/partially **replace** existing **business logic** ("default implementations") with custom logic
- **Recommendations** how to set up projects (especially features that don't exist upstream)
- Avoid **SQL string "blobs"** in the persistence layer (QueryDSL, JOOQ)
- **Performance** improvements in **JPA** by keeping up with latest standards (EclipseLink, soon version 3.x)

Fineract
Future Architecture (simplified)



Howto: Downstream Fork (recommended)

- **EVERY** downstream **fork** is based on a supported **release tag**
- **NEVER** follow the **upstream develop** branch in your downstream fork; things can break!
- If the latest release is 1.8.0 then a partner can choose between 1.8.0 and 1.7.0 (last two minor releases are supported)
- Hotfixes (e. g. 1.7.2) supercede the last previous releases (e. g. 1.7.0 or 1.7.1)
- Create your own **main** development **branch downstream**, e. g. “acme-develop” based on release tag 1.7.0 (“git checkout -b acme-develop 1.7.0”)
- If a **hotfix** becomes available for your fork then it is recommended to **upgrade** as soon as possible (within days? a week?)
- **Important**: pay attention to upstream development and releases; avoid getting stuck with a version that is EOL

Git Commands: Downstream Fork (recommended)

- Create a for from upstream Fineract in your Github organization via Github UI
- git clone git@github.com:acme/fineract.git && cd fineract
- git remote add upstream git@github.com:apache/fineract
- git fetch upstream && git checkout develop && git merge upstream/develop
- git checkout -b acme-develop 1.7.0
- Set your downstream main branch in Github settings to “acme-develop”
- Continue developing on “acme-develop”
- When a new release (e. g. 1.8.0) appears upstream…
- git fetch upstream && git checkout develop && git merge upstream/develop
- git checkout -b upgrade-1.8.0 acme-develop
- git checkout upgrade-1.8.0
- git merge 1.8.0
- Fix all conflicts, when done create downstream pull request, build, test, approve, merge into “acme-develop”

Howto: Downstream Fork (latest, greatest)

- **Important**: only for those who really want to follow the latest changes as close as possible and are aware of the potential **risks** that things might **break**
- Downstream fork based on **upstream develop branch**
- Pull changes from upstream as pull request merges happen or schedule a task and **sync every hour** (every day… depends on you)
- If your use case is e. g. to update a **demo installation** with the latest version then besides a full build and deployment nothing else is required
- If your use case is to provide the latest version with some degree of stability then maybe you want to run some **additional** manual and/or custom (integration) **tests** before deploying
- If you find an issue create immediately a **JIRA ticket** and (if possible) an associated **pull request**
- Only private customizations are developed downstream; everything else (bug fixes, general improvements) is contributed immediately upstream

Git Commands: Downstream Fork (latest, greatest)

- Create a for from upstream Fineract in your Github organization via Github UI
- git clone git@github.com:acme/fineract.git && cd fineract
- git remote add upstream git@github.com:apache/fineract
- git fetch upstream && git checkout develop && git merge upstream/develop
- git checkout -b acme-develop develop
- Set your downstream main branch in Github settings to “acme-develop”
- Continue developing on “acme-develop”
- When a change appears upstream…
- git fetch upstream && git checkout develop && git merge upstream/develop
- git checkout -b merge-upstream acme-develop
- git checkout merge-upstream
- git merge develop
- Fix all conflicts, when done create downstream pull request, build, test, approve, merge into “acme-develop”

Howto: Hotfix

- **Create JIRA ticket** requesting a new maintenance version (e. g. 1.7.1)
- If you anticipate multiple pull requests then create one **sub-task** per pull request
- Maximum of **3 pull requests** per hotfix (the number is somewhat arbitrary)
- **Create pull requests** based on targeted version tag ("git checkout -b fix-xyz 1.7.0")
- No major discussion discussions are needed as hotfixes have strict rules and contain only fixes and improvements of existing functionality (backwards compatibility!)
- Final vote by the PMC follows usual rules (min. 3 PMC member votes or 72h)
- **Important**: **EVERYONE** can request a hotfix and prepare the JIRA ticket and pull request(s)

Other Scenarios

- diverging too far from upstream
- new feature in older version
- version upgrade with private customizations
- customize existing services/business logic

Scenario: diverging too far from upstream

- Fineract had **no recommendations** how to fork the upstream repository
- A lot of (most?) long-time users **forked** based on the **upstream develop** branch
- … and continued to add their **customizations** (and the eventual bug fix) in the **downstream develop** branch
- This makes it immediately very **difficult to synchronize** with upstream, very likely to have Git conflicts
- Because synchronization is difficult developers start avoiding this task and fall further behind upstream changes

Solution: diverging too far from upstream

- If you continued your customizations downstream on the develop branch then create a **new branch** (“acme-customizations”) and move all your changes from downstream develop branch
- **Reset** your **downstream develop** branch (should contain the same as upstream)
- Do a full **synchronization** of your downstream repository with upstream
- **Create** a new downstream **branch** for your future **main development**; e. g. “acme-develop”; this branch should be based on the version you want to upgrade to (“git checkout -b acme-develop 1.7.0”)
- **Create** a new **branch** that will be used to merge your **changes**; e. g. “upgrade-1.7.0” based on your future main development branch (“git checkout -b upgrade-1.7.0 acme-develop”)
- **Merge** your changes to the upgrade branch (“git checkout upgrade-1.7.0 && git merge acme-customizations”)
- **Fix** all Git **conflicts** and take the opportunity to rearrange/refactor code according to our recommendations to avoid/minimize future conflicts; this might take a while…
- Once you are done, create a **downstream pull request** based on “upgrade-1.7.0”, build, test, approve and finally merge the pull request with your downstream main development branch “acme-develop”

Scenario: new feature in older version

- Partner is working on version 1.7.0
- Upstream development “fast” (aka multiple releases per year), version 1.8.0 is latest
- Partner wants to contribute new feature in 1.7.0, but hotfix rules forbid to add non-backward compatible features to older releases

Solution: new feature in older version

- At the moment not covered by recommendations and hotfix rules
- Needs more discussion with community

Scenario: version upgrade with private customizations

- Partner is working on version 1.7.0
- Upstream development “fast” (aka multiple releases per year), version 1.8.0 is latest
- Internal APIs (service interfaces etc.) might have changed significantly
- Partner has extensive private customizations (“secret sauce”)
- How can we ensure to make the migration as smooth as possible (similar to database migrations)?

Solution: version upgrade with private customizations

- Recommended **folder structure**
- Keep **customizations separate** from Fineract internals (small PoC contained upstream)
- Source migrations (similar to Angular “ng update”)

Scenario: customize existing services/business logic

- Partner intends to add small changes to existing business logic (95% is fine, 5% tweaked)
- At the moment not possible
- Services are automatically picked up by Spring because of “@Service” annotation

Solution: customize existing services/business logic

- Spring Boot “auto-configuration”
- conditional configuration
- Small PoC available upstream
- Note: easiest approach given the current architecture, but might be very coarse grained (aka services with “a lot” of functions); organizing in use cases and workflows would be better

Scenario: sponsor feature development upstream

- Partner wants to be as close as possible to the **latest developments**
- … but also wants to use only **proper releases** for production deployments
- Fineract is currently collaborating with a **major sponsor** who assigned own and external development resources to **actively drive upstream development**

Solution: sponsor feature development upstream

- Regular **meetings** with committers, other participating developers, project managers, PMC members to ensure the contributions are in line with the community goals
- More **resources**, more **features** and **fixes**, more **releases**

Summary

- Why are hotfixes such a big deal?
  - Enable downstream forks to contribute back upstream
  - Faster response to security and other issues
  - Smooth upgrades
  - Don’t force everyone to include latest developments
- Why is it important to do the code clean-up?
  - Developer experience
  - Work towards more modular architecture
  - Makes customizations easier
  - Keep upgrades as simple as possible
- Make sure the community as a whole doesn’t diverge too much from the supported releases
- Minimize production deployments that are EOL

APACHECON
NORTH AMERICA
OCTOBER 3 - 6, 2022
NEW ORLEANS
LOUISIANA
WWW.APACHECON.COM

Thank you

Questions?

Ed Cable
edcable@mifos.org
https://mifos.org

Aleksandar Vidakovic
aleks@apache.org

Appendix

Roadmap (boilerplate, modularity)

- ✓ Increase release frequency
- ✓ Hotfix/maintenance releases
- ✓ Partial/full replacement of business logic (“auto-configuration”)
- ❏ Reduce boilerplate code
  - ✓ Introduce Lombok for DTO and entity classes
  - ✓ Dependency injection via Lombok’s “@RequiredArgsConstructor”
  - ❏ Introduce Mapstruct to map DTOs to entities and back
  - ❏ Cleanup Spring Data repositories (get rid of unnecessary wrapper classes)
  - ❏ Introduce type safety in REST API layer, remove manual JSON de-/serialization
  - ❏ **Important: from this point on we need to refactor internally service interfaces and are technically not backward compatible in terms of code, but the REST API would stay the same**
  - ❏ Replace any JSON related classes in business logic services with DTOs
  - ❏ Remove handcrafted OpenAPI helper classes
  - ❏ Replace handcrafted SQL queries and JPA with QueryDSL
  - ❏ Add Java Bean Validation annotations to DTO classes
- ❏ Enforce clean architecture (primarily no JSON de-/serialization in business logic)
- ❏ Split monolithic codebase into independent modules (“JARs”)
- ❏ Publish JARs to Maven Central
- ❏ Fineract distributions: pick and choose only features/modules you need for a project
- ❏ Future:
  - ❏ automatic code migration (“OpenRewrite”)?
  - ❏ Self-contained modules/features à la Unix tools?
  - ❏ non-blocking/reactive from top (REST/Webflux) to bottom (database driver/R2DBC)?