Apache Fineract: Architecture

White Paper

This white paper describes the architecture of Apache Fineract, an Application Framework for Digital Financial Services (Framework), a system to support nationwide financial transactions and to support the creation of an inclusive, interconnected digital economy for every nation in the world.

Executive Summary

With the rise of mobile devices and fast access to any type of data, consumers are seeking a move away from locked-in solutions to open and fast continuous innovation. We need to build systems that allows continuous innovation for any vendor to serve the needs of their customer. The sheer amount of user transaction requires a solution that is ephemeral, scalable, and agile. This solution needs to run in the cloud, support any client device, and connect to legacy data and processes via APIs.

This change of paradigms implies a cultural change within any organization. It is about right-sizing digital innovation and focusing on cycle time optimization. It is important to continuously deliver innovations, while lowering the impact of delivering new features on the overall solution. This can only be solved when boundaries are removed, bringing development and operations close together, allowing them to collaborate for the good of the user. To keep continuous innovation agile and focused, small teams with faster tools are needed to build solutions instead of fighting over responsibilities.

Containerized microservices running in a dynamically managed environment will allow the support and deployment of lean and agile innovations. Applications that want to utilize this approach need to respect some boundaries that are collectively called cloud native. According to the Cloud Native Computing Foundation[1] these properties are:

A. *Container packaged*. Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation. Improves overall developer experience, fosters code and component reuse and simplify operations for cloud native applications.

B. *Dynamically managed*. Actively scheduled and actively managed by a central orchestrating process. Radically improve machine efficiency and resource utilization while reducing the cost associated with maintenance and operations.

[1] https://www.cncf.io/about/charter

C. *Microservices oriented*. Loosely coupled with dependencies explicitly described (e.g. through service endpoints). Significantly increase the overall agility and maintainability of applications.

This has led us to the conclusion that, based on our knowledge and experience over the last decade, the time has come to create a cloud native, microservice oriented framework to build digital financial services.

Design Principles

Domain Driven Design

Based on Evans' Domain Driven Design[2], we've taken a close look at the solutions we have built, decomposed the functionality, and rearranged them based on bounded contexts we have identified. In addition we have take recent research by the World Bank[3] into consideration identifying the core domains for digital financial services:

Domain	Resource
1 Organization Management	1.1 Users
	1.2 Permissions
	1.3 Roles
	1.4 Offices
	1.5 Employees
2 Customer Information System	2.1 Customers
	2.2 Identities
	2.3 User-defined fields
3 Group Management	3.1 Groups
	3.2 Meetings
4 Loan Portfolio Tracking	4.1 Loans
	4.2 Fees
	4.3 Interest Calculations
	4.4 Repayment Strategies
5 Deposit Account Management	5.1 Deposit Accounts
	5.2 Fees
	5.3 Interest Calculations
	5.4 Line of Credit
	5.5 Payment Strategies
6 Transaction Processing	6.1 Customer Transactions

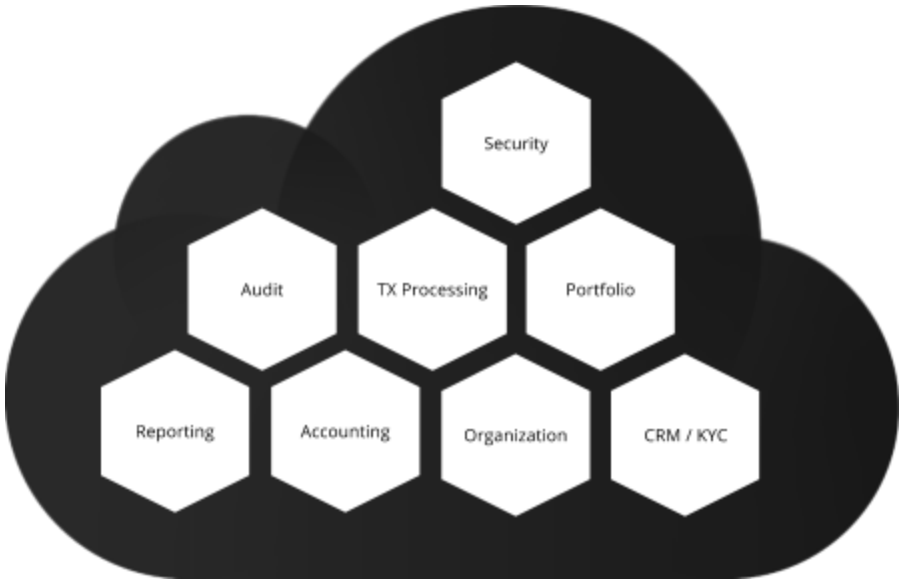
[2] https://en.wikipedia.org/wiki/Domain-driven_design
[3] https://openknowledge.worldbank.org/handle/10986/12272

	6.2 Group Transactions
	6.3 Teller Services
	6.4 Standing Orders
	6.5 Direct Debit Schemes
7 Accounting	7.1 General Ledger
	7.2 Journal Entries
	7.3 Trial Balance
	7.4 Bank Reconciliation
8 Audit Support	8.1 State changes
	8.2 Auth'n & Auth'z
9 Reporting	9.1 Customizable

Microservices

Based on these core domains we have created a set of microservices that allows any vendor to select, enhance, and extend the framework to build their own solution and provide continuous innovations on top of the framework.

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.[4]



[4] http://www.martinfowler.com/articles/microservices.html

CQRS

Every microservice is utilizing Command and Query Responsibility Segregation (CQRS)[5] and Event Sourcing[6] to enable fast processing and data enhancement with a clean separation of concerns. CQRS segregate operations that read data from operations that update data by using separate interfaces.

This pattern can maximize performance, scalability, and security; support evolution of the system over time through higher flexibility; and prevent update commands from causing merge conflicts at the domain level. To allow scalability and ephemeral behaviour every microservices is stateless, not storing any data or holding context related information, and supports multitenancy by default.

Containerization

The used patterns and specific implementation allows the containerization of any microservice provided by the framework. Because the bounded context provides a clean distinction between microservices, an available API supports seamless integration, and the stateless nature of every microservice, it is possible and preferable to run a microservice in an isolated unit.

We are reaching high availability, ephemeral behaviour, and scalability by utilizing containers and the ability to start and stop additional instances fast and without side effects to other microservices.

Dynamic Management

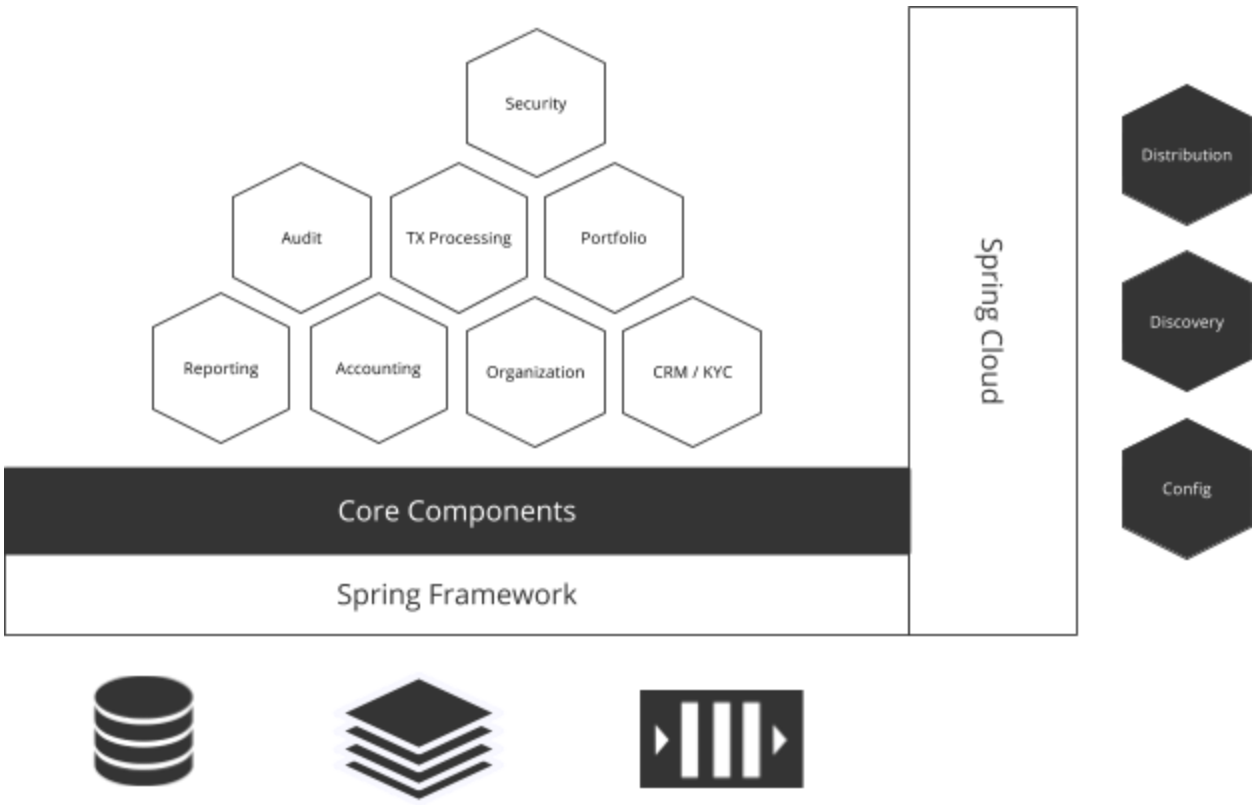
It is essential that containers are managed dynamically. A cloud native runtime will schedule containers, provide discovery services, allow the injection of configurations, and utilizes underlying infrastructure.

This orchestration is needed to automate the deployment of microservices to create a scalable and elastic runtime that can adjust to a growing number of consumers and transactions.

Components

The following diagram illustrates the used components:

[5] http://martinfowler.com/bliki/CQRS.html
[6] http://martinfowler.com/eaaDev/EventSourcing.html



Data Tier

RDBMS: A RDBMS is used to store relational data, providing a raw view. This data is optimized to be retrieved fast to create user based views, internal validation, reporting, or analytics. By default the framework comes with an MariaDB integration, that can be replaced with any Java Database Connectivity (JDBC) compatible RDBMS.

NoSQL: Apache Cassandra is used to handle the fast processing of state changes and financial transactions. To lower the impact use case specific models will have on the RDBMS, and to foster clean data separation, NoSQL is used to create use case specific view models out of raw data.

Message Queue: A Message queue is used to provide signals to parties which are interested in the change of data to build use case related sets of data. Data then will be retrieved using the API of a microservice. Apache ActiveMQ is used by default, but can be replaced with any Java Messaging Service (JMS) compatible message queue.

Application Tier

Spring Framework: To create light-weight modules, and focus on business functionality the Spring Framework is utilized. Cloud native requirements like context and dependency

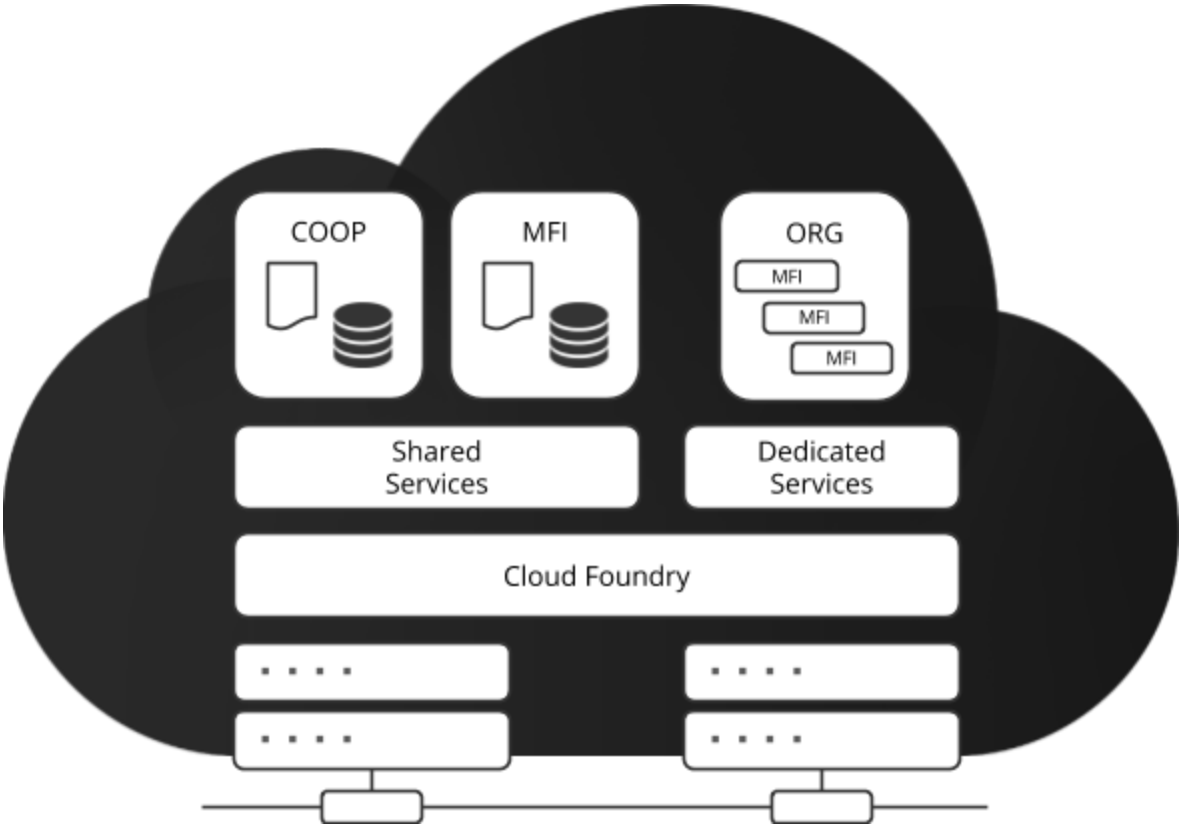
injection, transparent data access, or easy deployment are reached through the Spring Framework.

Spring Cloud: In a cloud native environment features like discovery, distribution, and configuration are essential. Spring Cloud offers us a toolset that simplifies the usage of these features, and allows us to choose the best solution based on the underlying infrastructure services.

Core Components: On top of the Spring Framework we have build additional libraries to ease the usage and configuration of CQRS, security, and multitenancy. These libraries are easy to use and we are working on integrating them into the Spring Framework themselves.

Runtime

We have chosen Pivotal's Cloud Foundry (PCF)[7] as our default runtime. PCF allows us to easily switch between infrastructure services, even deploying the framework on-premise in a data center. PCF is a complete platform to run cloud native applications. From configuration and distribution, via container scheduling, to security and logging, PCF is providing a full solution that is portable.



[7] https://www.cloudfoundry.org/

Disclaimer

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Mifos Initiative is approved by the Internal Revenue Service as a 501 (C) (3) tax-exempt organization, and all donations are tax deductible to the extent provided by law. The Mifos Initiative's Federal Identification Number (EIN) is 45-3613178.