Design Principles, Patterns & Architecture

This document describes the architecture of Apache Fineract CN.

Design Principles

Domain Driven Design

Based on Evans’ Domain Driven Design, 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 taken recent research by the World Bank into consideration identifying the core domains for digital financial services.

Services

Based on these core domains we have created a set of services 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 service 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.

CQRS

Every service employs the Command and Query Responsibility Segregation (CQRS) pattern and Event Sourcing to enable fast processing and data enhancement with a clean separation of concerns. CQRS segregates 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 behavior every services is stateless, not storing any data or holding context related information, and supports multi-tenancy by default.

Description of how CQRS is implemented for Fineract CN services

"Stateless" authentication

To maintain statelessness, and secure access at the same time, stateless authentication is performed via JWT bearer tokens which need to be regularly refreshed. The tokens are signed via a tenant-specific private key and long-lived tokens are transmitted as secure cookies, thus limiting the possibilities for interception.

Statelessness and Multi-tenancy

The services are multi-tenant capable. Data is placed in separate databases for each tenant. Which tenant is addressed by a request is transmitted via the request header, and via the bearer token. Because the services are stateless, no data is in the service which could be accidentally used to answer one tenant with data from another.

Containerization

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

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

Components

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 a MySQL 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 service. 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.