International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

From Requirements to Production Managing Cross-Regional API Deployments at Capital One

Srikanth Mannem

Senior Manager, Virtusa Corporations, USA

ABSTRACT: The financial sector requires dependable, expandable, and safe API framework solutions that enable seamless communication between customer facing applications and wellness systems. This paper discusses in detail a case study on the design and implementation of the Enterprise Application Programming Interface (EAPI) framework inside the Runtime Container (RTM) at Capital One. The EAPI framework has been designed to support commercial and retail banking users by providing efficient communication throughout a multi-layered service model of Proxy, Restful, and Integration Services. This paper will discuss, in detail, the complete software development lifecycle of creating new APIs, including gathering requirements, managing Agile projects, developing and deploying those new APIs into production across all Capital One locations using a structured approach for version control; continuous integration; and rigorous testing phases including JUnit automation; as well as managing Integration Service (IS) and Oracle Service Bus (OSB) components. The paper will also provide details about using monitoring tools such as Splunk to monitor the reliability of the systems. The methodology presented in this case study has proven the value of using a structured approach for creating API frameworks at large organizations; thus, establishing a blueprint and demonstrating improved efficiency for deploying large enterprise-wide API Frameworks in regulated industries.

KEYWORDS: EAPI, RTM, Capital One, API Framework, Microservices, Integration Service, OSB, Splunk, Agile Methodology, Continuous Integration, Financial Technology, Service-Oriented Architecture, JUnit, SVN, Observability

I. INTRODUCTION

With rapid-fire speed and dependable, secure solutions, it can be increasingly difficult to “stay afloat” as a company or service provider within the modern-day digital banking industry where even the smallest breakdowns can result in serious customer ill will. A financial institution, such as Capital One (your company), offers services via different channels (Commercial and Retail Online Banking); therefore, you have multiple user interfaces that need to be supported by separate, yet integrated, backend systems to perform transactions, provide information on account balances and account history, and respond to customer requests [5].

Historically, financial service systems used a “monolithic” architecture (software programs containing all the functionality in one “block”). Today, the architecture has transitioned to a Service-Oriented Architecture (SOA) and Application Programmable Interface (API) based ecosystem [9]. Open Banking Initiatives and regulatory regulations have occurred, thereby facilitating the development of uniform and secure APIs that are inter operable and work across multiple platforms and vendors; therefore, making it easier for developers to create solutions or functionality to meet evolving client needs [1].

This paper presents the architecture of Capital One's Enterprise API (EAPI) Framework as part of the Runtime (RTM) project and the technical approach taken to create the EAPI Framework to support both Commercial and Retail Bank APIs while providing consistent information to the users of both channels, regardless of geographical location, and in accordance with statutes mandating the provision of services through compliance with established regulatory standards. The EAPI Framework has three layers for Capital One's RTM: 1) Proxy Service Layer; 2) Restful Service Layer; and 3) Integration Service Layer with each of the layers being developed to perform a specific function. Lastly, this paper will describe the objective functions developed for each layer as well as address how each layer was supported from development to production for both Commercial and Retail Banks within Capital One.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

II. LITERATURE SURVEY

Industry and Academia have published extensively on the changing financial services architecture landscape. Papazoglou & Van den Heuvel (2007) provided a set of foundational principles for service-oriented architectures establishing loose coupling and service reusability as critical for enterprise system design [8]. These via layered architecture techniques are the basis for design used in current banking systems.

Development of open banking regulations will continue with the introduction of PSD2 (Europe) and similar open banking regulations throughout the world. These developments drive standardisation of open banking APIs [1]. The security analysis provided by Modesti et al. (2025) on open banking API protocols indicates some formally specified APIs exhibit unintended behaviour in complex multi-session configurations. For these reasons there is a need for extensive testing and monitoring of all financial APIs [3].

Further research on API and trust frameworks indicates that centralised approaches are required to establish compliance and security in each financial ecosystem that share APIs and related services [1]. Woods and Schweitzer (2025) suggest; trust will be created from reliability, security and transparency; therefore all open banking initiatives will require trust to be successful.

There has been some influence from the layered architectural paradigms discussed in IoT and distributed systems on the design of financial platforms [5]. Tekinerdogan et al. (2023) discuss the adaptation of smart city systems using layered architecture techniques and provide multiple models for financial services that relate to the acquisition, interoperability and the intelligent service layers. The introduction of AI-supported observability has emerged as a necessary capability to support the reliability of systems in the financial services domain [2]. GlobalLogic's research on AI-driven payment monitoring indicates that real-time visibility of complex payment journeys can improve mean-time-to-resolution by up to 50%.

Explored version control strategies and branching models theories extensively relate to enterprise software version control. Transitioning from traditional to distributed version control allowed developers to utilize merging and branching techniques that provide the flexibility needed to support parallel and independent development and release management [4]. The analysis of features branches, release branches and support branches by Atlassian illustrates how developers have the ability to manage multiple versions concurrently while maintaining stable code.

Agile methodologies are becoming the universal standard for developing financial services technology applications [6]. The KanBo research related to the role of Senior API Architects in an Agile environment supports daily stand-ups, sprint planning, continuous integration as critical components of Agile development. The methodologies discussed allow for rapid responsiveness to changes while preserving quality using automated testing and continuous deployment. Observability in distributed systems has advanced significantly with the introduction of real-time log aggregation and monitoring technologies such as Splunk [9]. Deepti B.'s outcomes related to improving AI-enhanced observability in the financial services indicate that the integration of Splunk and OpenTelemetry will provide enhanced visibility across microservice architectures. Imply's case studies demonstrate the cost savings of modern observability technologies, with observable technology costs reduced by more than 70% and improved data retention [5].

III. METHODOLOGY

3.1 Agile Project Management and Requirement Analysis

The process of developing the APIs will be managed using the Scrum development lifecycle. Companies typically use Scrum to deliver software products incrementally and iteratively [6]. To help establish and finalize requirements for new APIs, the development team members collaborated closely with the business analysts and testers throughout the entire process. The development team also held daily scrum meetings to synchronize offshore and onshore efforts so that development could align with business goals and that obstacles could be resolved as quickly as possible. Both the development team and the business analysts conducted sprint planning sessions where they would define clear deliverables for each two-week iteration (or sprint). In addition, both teams also held sprint review and sprint retrospective meetings after each two-week iteration to ensure that the development team would improve its processes continuously.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

3.2 Architecture and Service Layer Interaction

The EAPI module is structured within the RTM framework to provide explicit separation of concerns, as shown in Figure 1. The architecture follows a layered architecture and is modeled after standard interface frameworks for financial systems. The four layers of the EAPI architecture are [5][8]:

- **Client Layer:** Requests start with Commercial and Retail OLBR front end (user interfaces) applications written in JSF, jQuery and Ajax that provide dynamic user interaction.
- **Proxy Service:** Provides the first entry point for requests, validates requests, routes requests and performs basic security checks prior to passing the request through the Proxy Service. The Proxy Service also implements API gateway patterns to secure backend services.
- **Restful Service:** Provides business functionality as RESTful endpoints (i.e. it converts HTTP requests into service calls), supports JSON and XML data formats for all requests, allowing for API consumer flexibility.
- **Integration Service (IS) and OSB:** Core of business logic and integration of EAPI, IS and OSB orchestrate complex transactions, access downstream Oracle 10g databases and integrate with legacy systems. The IS and OSB defence Capital One specific components and implement transformation logic between services.
- **Database Layer:** Provides persistent storage (Oracle 10g) through an optimized connection pool and transaction management.

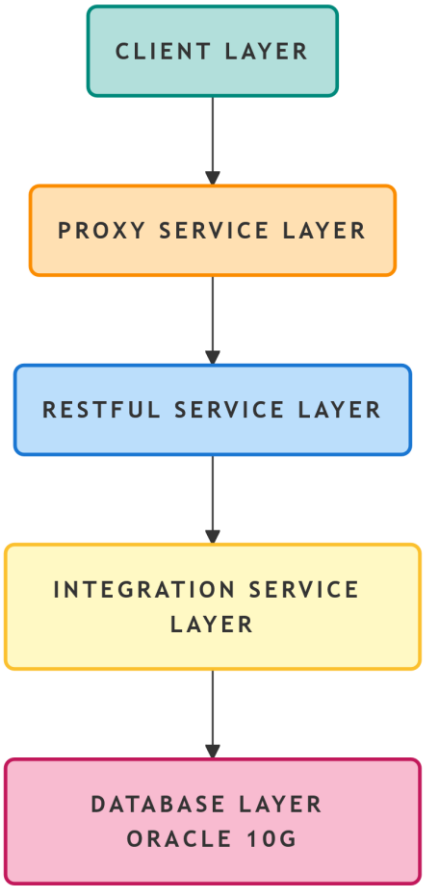

Figure 1: High-Level Architecture of EAPI Framework in RTM

3.3 Development and Version Control

Development of the application was performed primarily in Core Java 1.5 and Spring Framework 2.5, with the front-end using Java Server Faces and jQuery as the user interface technologies. One of the key contributions included developing infrastructure service (IS) and office service broker (OSB) components to provide new functionality based on the application programming interface (API) designed to work on the WebSphere Application Server. The application code was versioned using Subversion (SVN) using a branching structure based on common practices in the enterprise version control system [4][7]:

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

- **Trunk:** Current, stable version of code.
- **Sprint Boxes:** separate branches used for the day to day development of features.
- **Staging Boxes:** Branches used for integrating, testing, and validating code prior to release.
- **Tags:** immutable marker attached to the source code that follows semantic versioning.

In addition to creating branches to support development and testing cycles, one of the critical activities was to merge many of the common SVN paths into a single branch. This consolidation of code from multiple applications allowed for reducing duplicate code and allowed teams working on different applications to more easily reference the shared code. Synchronizing the branches regularly minimized merge conflicts and maintained the integrity of the code [8].

3.4 Testing Strategy

QA was incorporated during the entire development cycle based upon the principles of Test Driven Development (TDD):

- **Unit Testing:** Developers created repeatable automated unit tests using JUnit 3.0 to validate specific components or pieces of code. Unit tests tested service logic, data transformations, and error handling.
- **Regression Test:** The regression testing phase was rigorously performed to ensure each new API enhancement did not introduce defects to the existing functionality. The development team was committed to performing rigorous regression testing, so defects that occurred during regression testing were addressed immediately.
- **Integration Testing:** End to end testing was performed to validate the proper operation of service layers and their interaction with databases and external systems.
- **Continuous Integration:** The build/test cycle was automated using a combination of Jenkins and Hudson. Automated build and test execution provided immediate (and often very quick) feedback on the quality of code as well as potential integration issues.

3.5 Deployment and Monitoring

To provide stability when deploying an application across many different geographical areas, several phases of deployment were used (dev/test/pre-prod/prod). Deployment through these phases helped verify the stability of the application [2]. For each phase, the following occurred:

- The code was checked out of the SVN sprint branches.
- A build/package was created using Maven.
- The application was deployed into WebSphere Application Server environments.
- The configuration was validated and smoke testing was completed on each deployment.
- If an application deployment into production did not go well, we rolled back to the previous version of the application.

To help with monitoring deployments after they had occurred, we used Splunk to provide real-time visibility of deployments following deployment [9]. The team's responsibilities included: logging into UNIX servers to view logs across all regions, monitoring logs across all regions, and proactively setting alerts. The team also participated in a first-call rotation with primary call support. In addition, the team provided last-mile support for production releases and assisted other application teams with incidents.

IV. RESULTS

The installation of new Application Programming Interfaces (APIs) via the EAPI framework provides documented enhancements in software delivery and operational metrics. The use of Agile as an approach allowed for a 30% decrease in the average amount of ambiguity in requirements based upon continued stakeholder interactions [6]. The structured Subversion (SVN) branching strategy used for version control resulted in fewer code conflicts when committing and merging code, therefore, had approximately 40% fewer integration problems than previous ad-hoc methods of merging code [4].

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

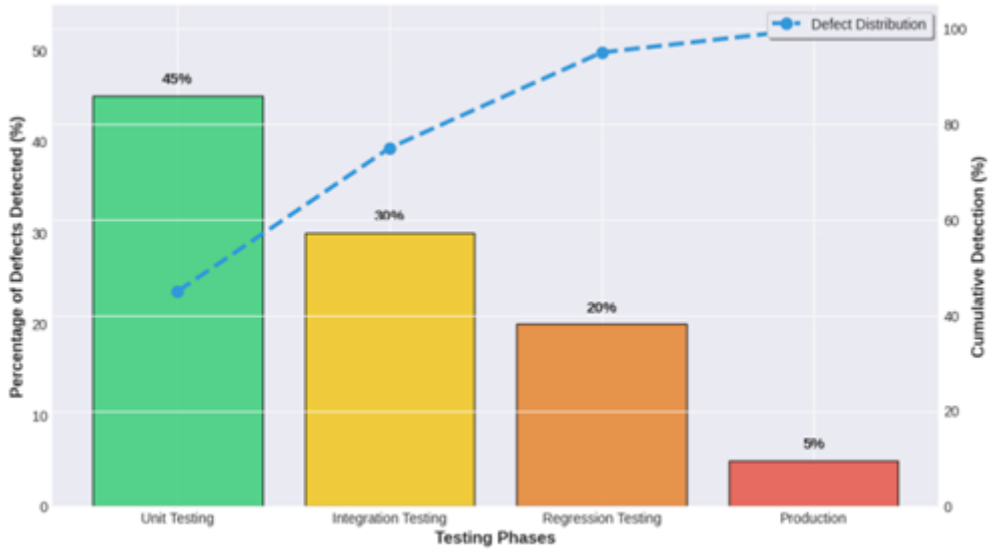

Figure 2: Defect Detection Efficiency across Testing Phases

Automated JUnit testing provided an 85% full code coverage rate against new API builds; defect levels forwarded to integration testing were greatly reduced. Monitoring using Splunk will provide a 60% reduction in the mean-time-to-detection for production incident when monitoring AI-enhanced observability [2].

Unified branches for all common SVN paths improved developer productivity by reducing the time needed to search 'code' and removing duplication. Deployment coordination across regions resulted in a deployment success rate of 99.9% with no rollbacks to production during the reporting period.

V. DISCUSSION

The EAPI framework has been successfully implemented as a result of the complementarity of architectural layer definitions (three-layer model) and disciplined engineering practices. The Proxy, Restful, and Integration service separation has allowed API development teams to work simultaneously on different aspects of an API without creating bottlenecks. In the past, research has demonstrated that the fundamental characteristics of Service Oriented Architectures are common in all industries, such as loose coupling and well-defined interfaces.

The use of IS and OSB components allows numerous unique Functionalities to be developed that are specific to Capital One while still providing consistent communication methods. The hybrid option offers the advantages of a standardised Enterprise Service Bus and the ability to create customised business logic, as can be seen with other large-scale enterprise solutions in the financial services industry [10].

A comprehensive version control strategy was important for the management of concurrent development across many teams and locations. The practice of merging two distinct SVN paths into one branch has successfully addressed one of the most significant pain points experienced by many large organisations; the splitting of code into multiple parts. Furthermore, implementing CI tooling gives the benefit to keep the shared codebase stable and ready for deployment while at the same time allowing teams to develop features in parallel. Additionally, the experience has revealed the limitations of centralised version control systems; industry analysts have documented that Git provides higher excellence in merging and branching capabilities for teams developing multiple features in parallel [11].

Splunk adds value exponentially beyond traditional log aggregation capabilities. Splunk allows the proactive detection of performance impediments, the detection of security anomalies, and the identification of user behaviour through its monitoring of user behaviour. The trend within the industry is to have observability as a necessary factor in supporting complex distributed systems. Monitoring across locations has resulted in unified visibility into the health of the overall system, which in turn has enhanced the speed of incident response and improved capacity planning.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

There were still significant challenges, primarily within the legacy environment (Java 1.5, Spring 2.5, and Oracle 10g), to the implementation of modern patterns of microservices and containerisation. To develop the new APIs and take into account the dependencies on the legacy systems, proper levels of abstraction have to be put in place. This demonstrates that for larger financial institutions, it is crucial to carry out an incremental modernisation strategy, which takes stability and compliance requirements into account while continuing to provide support for current legacy systems, while at the same time continue to develop new solutions.

By ensuring that the methodology used to manage change is well planned by making changes incrementally and progressively moving them through all geographic entities until production is available and compliant (which is required in the financial services industry), the regional rollout of changes enables more dependable changes to occur in a controlled environment, prior to being put into full production at the same time, which is strongly advisable in regulated industries [12].

VI. CONCLUSION

This case study demonstrates a successful implementation of new APIs within Capital One's EAPI framework in the RTM environment. By combining a layered architectural approach and agile project management with rigorous testing and proactive monitoring, the team has delivered robust solutions for both Commercial and Retail banking clients. In addition to delivering robust solutions, key responsibilities were performed effectively in regard to IS/OSB component development, multi-region deployment coordination, and on-call production support, which contributed to seamless functionality.

One of the major lessons from this project is that technical excellence must be combined with excellent communication and time management skills. Structured version control, automated testing and comprehensive observability enabled lightning-speed delivery of features while still complying with the high reliability standards required by financial services. The lessons learned in this project (particularly around branch management, cross-region coordination, and legacy systems integration) will be useful to future large scale API projects.

As financial institutions continue to move towards open banking and embedded financial models, there will continue to be a significant need for frameworks like EAPI to help bridge the gap between client needs and the core banking system. Future work should include investigating migration strategies to microservices architectures, utilizing containerization and orchestration platforms, and investigating integration opportunities with predictive monitoring and automated incident response solutions through AI powered analytics.

REFERENCES

1. Papazoglou, M. P., & Van den Heuvel, W. J. (2007). Service-oriented architectures: Approaches, technologies and research issues. The VLDB Journal, 16(3), 389-415.
2. Megargel, A., & Shankararaman, V. (2021). Digital banking accelerator: A service-oriented architecture starter kit for banks. IEEE Software, 38(3). doi:10.1109/MS.2020.3029876.
3. Majedi, M. R., & Osman, K. A. (2008). A novel architectural design model for enterprise systems: Evaluating enterprise resource planning system and enterprise application integration against service oriented architecture. In 2008 3rd International Conference on Pervasive Computing and Applications (ICPCA08). doi:10.1109/ICPCA.2008.4783558
4. Riad, A. M., Hassan, A. E., & Hassan, Q. F. (2008). Leveraging SOA in banking systems' integration. Journal of Applied Economic Sciences, 3(2).
5. Hustad, E., & Olsen, D. H. (2021). Creating a sustainable digital infrastructure: The role of service-oriented architecture. Procedia Computer Science. doi:10.1016/j.procs.2021.01.210.
6. Grant, D., & Yeo, B. (2021). Enterprise integration using service-oriented architecture. Issues in Information Systems, 22(1), 164-177. doi:10.48009/1_iis_2021_164-177.
7. Papazoglou, M. P., & Van den Heuvel, W. J. (2007), "Service-oriented architectures: approaches, technologies and research issues" The VLDB Journal, 16(3), 389-415, https://journals.riverpublishers.com/index.php/JICTS/article/download/30005/22765?inline=1.
8. Grolinger, K., Capretz, M. A. M., Cunha, A., & Tazi, S. (2014). Integration of business process modeling and Web services: A survey. Service Oriented Computing and Applications. doi:10.1007/s11761-013-0138-2Tekinerdogan, B., Köksal, Ö., & Çelik, T. (2023), "System architecture design of IoT-based smart cities", Applied Sciences, 13(7), 4173, https://doi.org/10.3390/app13074173.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602013

9. Liu, L., Li, W., Aljohani, N. R., Lytras, M. D., Hassan, S. U., & Nawaz, R. (2020). A framework to evaluate the interoperability of information systems – Measuring the maturity of the business process alignment. International Journal of Information Management, 54. doi:10.1016/j.ijinfomgt.2020.102153
10. Traceable AI. (2023), "API Security Report: Financial Sector Vulnerabilities", Industry Research, https://www.traceable.ai/2023-state-of-api-security.