Benefits of ISL-Based Application Integrations

By David Agahchen Architect and Technical Lead

JADE Team

Chief Technology Office/Information Systems Branch | Ministry of Attorney General and Public Safety Sector

March 20, 2024

# Introduction

In today's rapidly evolving digital landscape, the need for efficient and reliable application integration has never been more critical. The agile-integration based Integration Services Layer (ISL) provides a robust, agile, and scalable platform for integrating various applications, leveraging leading-edge architectural paradigms such as Ports and Adapters, Microservices, Event-First (SOA 2.0), and Domain-Driven Design.

This document aims to articulate the practical reasons for selecting ISL as the integration platform within the business sphere. We will highlight the strengths and benefits that ISL offers, ensuring the seamless interoperability of enterprise applications, enabling the reusability of such integrations, and consequently facilitating more effective and efficient business operations.

# Section 1: Example Mock User Story

*As a BC prosecutions officer, I use two separate applications: one for recording charge assessments (JUSTIN) and another for managing case evidence files (DEMS). My workflow is hindered by the fact that these two applications do not communicate with each other, requiring me to manually input data from App A into App B, leading to inefficiencies and potential inaccuracies.*

*With ISL as an integration platform, I need these two applications connected. When an agency-issued charge assessment is recorded in JUSTIN, an event is triggered. This event is then picked up by DEMS, which should automatically create a new case file based on the incident data. This seamless integration would streamline my workflow, minimize the potential for human error, and allow me to focus more on my core duty of ensuring that information surrounding a charge assessment and potentially the court cases are correctly reflected in both systems.*

*Should there be any alterations to the business rules, I need to have the flexibility to modify the conditions under which DEMS responds to the event, all without having to make changes to either JUSTIN or DEMS. Moreover, I need to be able to leverage the same infrastructure and integration in order to keep the accused individuals and their defense counsel informed with the latest developments in their case when the time is right.*

# Section 2: Summary of ISL Key Benefits

Here is a table summarizing the key benefits associated with each architecture/design used in ISL.

|  |  |
| --- | --- |
| Architecture / Design | Summary of Key Benefits |
| Ports and Adapters | 1. Decouples business logic from technology, promoting longevity and adaptability. 2. Enhances testability. 3. Allows for isolation of changes. 4. Encourages reusability and interchangeability of components. 5. Facilitates parallel development. |
| Microservices | 1. Supports independent development & deployment. 2. Allows for technology diversity. 3. Provides independent scalability for different services. 4. Enhances resilience through isolated failure points. 5. Facilitates iterative design. 6. Encourages data isolation. |
| Event-First (SOA 2.0) | 1. Promotes decoupling and independence between services. 2. Supports real-time communication. 3. Enables easy scalability by processing events independently. 4. Enhances resilience through event queuing and replay capabilities. 5. Improves performance and resource usage through asynchronous processing. 6. Enhances a system’s adaptability to evolving business needs. 7. Facilitates traceability and monitoring |
| Domain-Driven Design | 1. Encourages a shared language and understanding. 2. Reduces complexity by breaking down the system. 3. Supports integrated design and strategy. 4. Isolates domain models to minimize impact of changes. 5. Provides context mapping to facilitate interactions. 6. Places focus on business needs. |

It's important to note that the implementation of each architecture/design and their associated benefits are highly dependent on the specific business needs and technological context. A well-integrated system, like ISL, combines these architectures to create a system that is scalable, resilient, and adaptable to changing business needs.

# Section 3: Deep Dive into ISL Architectural Paradigms and Designs

Having provided an overview of the benefits associated with each architectural paradigm and design principle used in ISL, we now turn our focus to a more detailed exploration of these components. In this section, we will dissect each architecture and design, expanding on their mechanisms, nuances, and the specific advantages they offer.

We will start with the Ports and Adapters architecture, focusing on its ability to effectively decouple business logic from specific technologies. We will then transition to discussing the Microservices architecture and its independent and resilient nature. Following that, we will unpack the details of the Event-First (SOA 2.0) approach, with its emphasis on real-time communication and scalability. Lastly, we will look at the Domain-Driven Design (DDD), focusing on its merits in promoting a shared understanding and simplifying complex systems.

Through this section, we aim to provide you with a comprehensive understanding of how these architectures and designs underpin ISL, and how they work in synergy to facilitate seamless application integration. Armed with this knowledge, you will be better equipped to appreciate why ISL is the platform of choice for integrating your enterprise applications.

## **Ports and Adapters**

The Ports and Adapters Architecture, also known as Hexagonal Architecture, is an architectural pattern proposed by Alistair Cockburn. It allows an application to equally be driven by users, programs, automated tests, or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases. The Onion and Clean architectures are built on the Ports and Adapters architecture.

When integrating two existing enterprise applications, it can be a suitable approach for several reasons:

1. **Decoupling from Technology**: Ports and Adapters architecture can decouple the core business logic of an application from any specific technology, library, or database. This decoupling allows the core business logic to remain unaffected by the changes in peripheral technologies, such as the technology used in another enterprise application.
2. **Testability**: Given that this architecture decouples the application's core logic from its inputs and outputs, it's easier to create unit tests or integration tests. This helps ensure the stability and reliability of the integrated systems.
3. **Isolation of Changes**: If a change needs to be made to one system (for example, changing the database or moving to a different infrastructure), the other system is not directly affected, because the change is isolated by the adapters. This simplifies maintenance and makes it safer to make changes.
4. **Reusability and Interchangeability**: Since the core logic is decoupled from the inputs and outputs, you can use the same core logic with different adapters. For example, if two enterprise applications need to be integrated, they can use the same core logic but have their own custom adapters.
5. **Parallel Development**: As the interfaces (ports) define the contract for the data exchange, teams can work in parallel on the development of different parts of the system. One team can work on the adapters for one system while another team works on the adapters for the other system.

In essence, the Ports and Adapters architecture provides a high degree of modularity and loose coupling, facilitating easier integration of different enterprise applications.

### **Microservices**

Microservices architecture is another architectural pattern that structures an application as a collection of loosely coupled, highly maintainable, and independently deployable services. These services correspond to different business capabilities and can be developed, deployed, and scaled independently.

When integrating two existing enterprise applications, a microservices architecture can be beneficial for several reasons:

1. **Independent Development & Deployment**: Microservices can be developed and deployed independently of one another. This means that integrating new or existing services into the system can be done with minimal impact on the rest of the application.
2. **Technology Diversity**: Each microservice can use a technology stack that is most suitable for its requirements. This means one application could be written in Java, another in Python, and they could still communicate seamlessly using well-defined interfaces (typically HTTP/REST or messaging queues).
3. **Scalability**: Microservices can be scaled independently according to their individual needs. If the demand for one service increases, that service can be scaled without needing to scale the entire application. This is advantageous when integrating large enterprise applications that may have different scaling requirements.
4. **Resilience**: If one service fails, the others can continue to function. This isolation reduces the risk of total system failure. When integrating two large enterprise applications, this resilience can be critical.
5. **Iterative Design**: Microservices allow for an iterative design and development, where the system can easily adapt and evolve with changing business requirements. If one part of the application needs to be updated or replaced, it can be done with minimal impact on the other parts of the system.
6. **Data Isolation**: Each microservice can have its own isolated database. This enables them to avoid being affected by direct database schema changes from other services.

### **Event-First (SOA 2.0)**

Event-First or Event-Driven Architecture (EDA), often referred to as Service Oriented Architecture 2.0 (SOA 2.0), is a pattern that focuses on producing, detecting, consuming, and reacting to events. An event is a significant change in state in a source application that other services need to be aware of. In this architecture, the producer application generates an event, and the consumer application reacts to it.

This approach can be beneficial when integrating two existing enterprise applications due to the following reasons:

1. **Decoupling**: The producer and consumer applications are decoupled, which makes them independent. Changes in one application do not directly impact the other, which simplifies development and reduces the risk of introducing bugs.
2. **Real-Time Communication**: EDA allows for real-time or near-real-time communication between applications, making it a good choice when rapid response to changes is required.
3. **Scalability**: As each event can be processed independently, it's easier to scale the application to meet demand. You can increase the number of instances of the consumer application to process more events in parallel.
4. **Resilience**: If one service fails, it won't stop the entire system from working. Events can be queued and processed when the service is available again.
5. **Asynchronous Processing**: EDA naturally supports asynchronous processing, which can lead to improved performance and resource usage by not requiring immediate response and allowing operations to be spread over distributed systems.
6. **Adaptability**: EDA allows systems to easily adapt to new and changing business requirements. It's easy to add new functionality by adding new event types and handlers. If a new business requirement arises that necessitates processing events in a new way, you can add a new processing logic to consume and process the events without needing to modify the producer or other consumer services.
7. **Traceability and Monitoring**: Events can be logged and monitored, providing an audit trail that can be used to trace the flow of events through the system, helping with debugging and understanding system behavior.

### **Domain-Driven Design**

Domain-Driven Design (DDD) is a development approach that prioritizes understanding the business domain and using a rich, ubiquitous language that aligns closely with that domain. This allows developers and domain experts to communicate effectively, reduces complexity, and improves model precision.

When integrating two existing enterprise applications, DDD can offer several advantages:

1. **Shared Language and Understanding**: DDD encourages developing a ubiquitous language for each bounded context (a boundary within which a particular model is defined and applicable). This shared language helps in integrating the two applications, as it provides a common understanding of the domain model.
2. **Reduced Complexity**: DDD breaks down the system into smaller, manageable parts (bounded contexts). This reduction in complexity helps when integrating the two applications, as each application can focus on its specific domain model, reducing misunderstandings and miscommunications.
3. **Integrated Design and Strategy**: In DDD, strategic design plays a crucial role in defining the relationships between different bounded contexts (through context mapping). This can facilitate the integration of two applications by clearly outlining how they should interact and how their models should align.
4. **Isolation of Domain Models**: Each bounded context in DDD has its own domain model, which is independent of the domain models in other bounded contexts. This isolation can be beneficial when integrating two applications, as it allows each application to evolve independently, with minimal impact on the other.
5. **Context Mapping**: DDD includes techniques for mapping and translating between different bounded contexts. This can simplify the integration of two applications by providing a clear framework for managing the interactions between their different domain models.
6. **Focus on Business Needs**: DDD puts business needs at the center of the design process. This can be beneficial when integrating two applications, as it helps ensure that the integration meets the actual needs of the business, rather than just technical requirements.

In the DEMS Integration, the pilot project for ISL, the ISL Court Case Management (ISL-CCM) application encapsulates three distinct domain models: the EDT DEMS enterprise application, the JUSTIN enterprise application, and a specific business model.

Hard-coding a direct mapping between the DEMS system model and the JUSTIN system model would be restrictive and could prohibit reuse. Hard-coding refers to explicitly setting the connections or dependencies between two systems in the program's code. This approach is often inflexible, making the code harder to modify, maintain, or reuse for other integrations because any changes would require code modifications.

To avoid this, the transformation between DEMS and JUSTIN occurs via the intermediary of the CCM business model. This approach allows for more flexibility and promotes reuse, as the transformations are not directly tied to the specific systems.

By doing this, ISL-CCM successfully navigates complex data mapping scenarios, such as associating JUSTIN Agency Files and JUSTIN Court Files to DEMS Cases, without losing the business context. The system maps each item to their corresponding business objects, specifically Charge Assessments and (Approved) Court Cases. This technique ensures a clearer, more context-sensitive correlation between different systems, promoting code reuse and overall system adaptability.

# **Section 4: Comparing ISL-Based Integrations to Point-to-point Integrations**

As we further explore the capabilities and benefits of ISL, it's crucial to understand its performance relative to traditional integration methodologies. In this section, we will compare ISL-based integrations with point-to-point integrations, which have been widely used in enterprise application integration. Both have their uses and both can be supported in the sector.

Point-to-point integration, while functional in certain contexts, often faces limitations in terms of scalability, flexibility, resilience, maintenance, complexity management, and reuse potential. The comparison will focus on these aspects, providing a comprehensive perspective on how ISL's strategic use of Ports and Adapters, Microservices, Event-First (SOA 2.0), and Domain-Driven Design architectures can address these limitations and deliver superior performance.

This comparison table serves as a snapshot of the strengths of ISL-based integrations over traditional point-to-point integrations, particularly in contexts that demand high adaptability, scalability, and resilience.

|  |  |  |
| --- | --- | --- |
| Aspect | ISL-Based Integration | Point-to-Point Integration |
| Scalability | Designed for scalability. Each component can be independently scaled according to demand, due to its event-driven and microservices architectures. | Scalability is dependent on the specific implementation but can become challenging as connections increase, due to the tightly coupled nature of point-to-point integrations. |
| Flexibility | Highly flexible. Changes can be made to one service without impacting others. Events can be added and handled independently. | Flexibility can be limited. Any change often requires changes in both the sending and receiving applications due to the direct nature of the connections. |
| Resilience | Enhances resilience through event queuing and replay ability. Failure in one service does not cause system-wide failure due to the decoupled nature of services. | Resilience depends on the specific implementation. However, failures can propagate through the system due to the direct dependencies between services. |
| Maintenance | Easier to maintain. Changes to one service don't necessitate changes to others. The system can evolve over time without major overhauls. | Maintenance can be more challenging. Changes often require code modifications in multiple systems due to the tightly coupled nature of the integrations. |
| Complexity | Managed complexity through clear domain boundaries, decoupling services, and promoting modular design. | Complexity can increase with the number of services, leading to a tangled web of direct connections that can be difficult to manage. |
| Reuse | Promotes reusability. A service can easily be reused by other services or applications by subscribing to the relevant events. | Reusability can be limited. Each integration is specific to the two systems being connected and often cannot be easily reused elsewhere. |

# **Section 5: Summary**

In summary, the agile-integrated Integration Services Layer (ISL) provides a robust and flexible solution for integrating enterprise applications. Its use of Ports and Adapters, Microservices, Event-First (SOA 2.0), and Domain-Driven Design architectures offers significant advantages, including enhanced adaptability, scalability, resilience, maintainability, and reusability.

Moreover, alongside these design benefits, there is also a significant economic advantage to adopting ISL-based integrations. Leveraging the ISB’s JADE team for development and operational support can result in noticeable economies of scale.

While a business area could certainly choose to develop and maintain their ISL-based integration solution independently, they stand to benefit from using the corporate service provided by the IDS team. This approach eliminates the need for the business area to staff a dedicated development and support team, thereby saving resources and potentially increasing the speed and efficiency of the integration process.

Overall, ISL offers an integrated platform that not only improves the architectural design and functionality of business operations but also presents economical benefits that enhance operational efficiency and resource management.