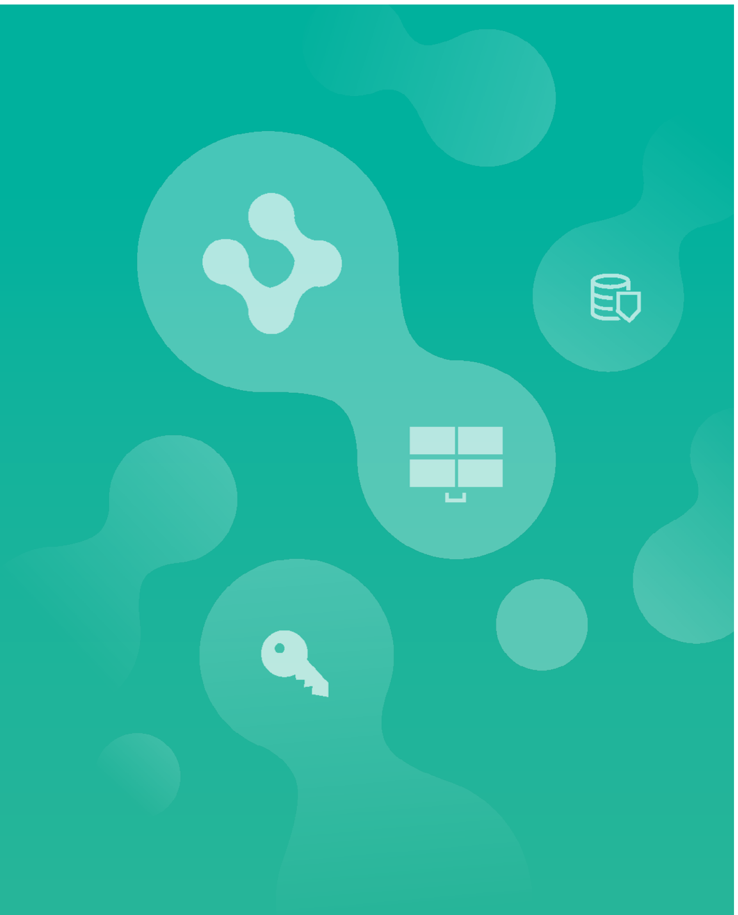


A Guide to Connectivity for OpenFin Architects and Developers

solace

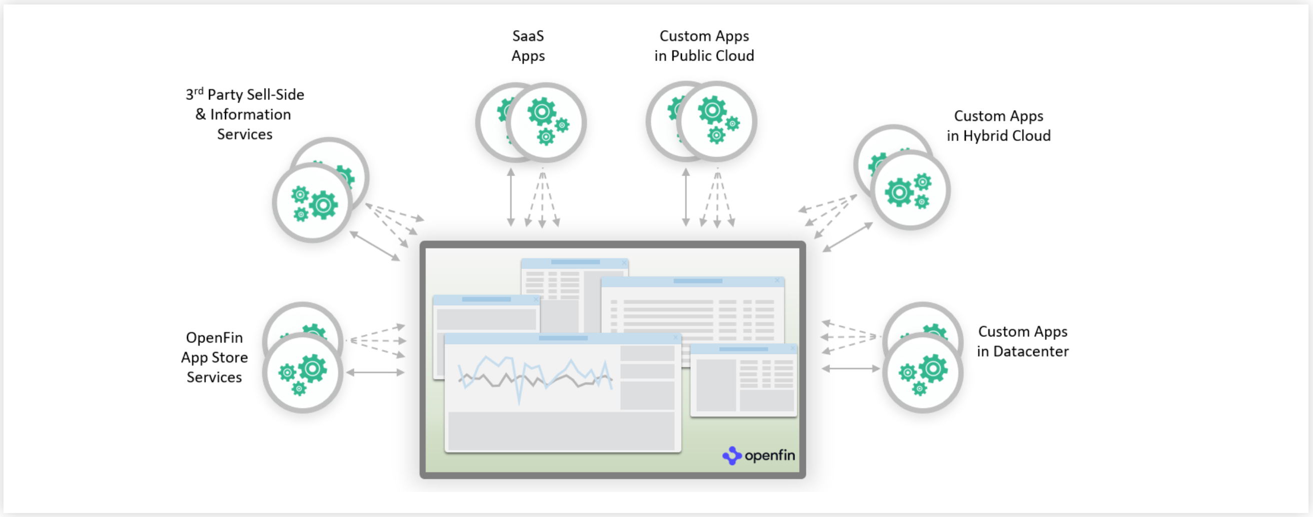
“Web-based technology adoption within financial firms has lagged primarily because of the industry’s special needs around security and regulatory compliance. OpenFin identified this gap and created a new desktop OS that has clearly struck a chord—more than 1,500 firms now rely on it.

THOMAS KUNNUMPURATH

TABLE OF CONTENTS

EXECUTIVE SUMMARY

OpenFin is advancing how data securely interoperates between applications on financial desktops. Of equal importance is how OpenFin applications receive their data in the first place. While the majority of core data services still reside in the datacenter, workloads are increasingly shifting to the public cloud, and financial firms are increasing their adoption of SaaS services and third-party sell-side applications. Different sources have different access patterns (request/reply, publish/subscribe, streaming). Some are built with modern web technology, while many more are legacy.


SaaS Apps
Custom Apps in Public Cloud
3rd Party Sell-Side & Information Services
Custom Apps in Hybrid Cloud
OpenFin App Store Services
Custom Apps in Datacenter
openfin

This diversity makes it challenging for firms to develop a cohesive strategy for firm-wide connectivity that gels with agile development. OpenFin developers need a way to seamlessly and securely interact with these data sources while preserving the agile OpenFin ethos.

This paper outlines three different architectures you may choose to connect OpenFin applications to different kinds of data sources:

1. **Manage connectivity in the OpenFin app**. OpenFin allows you to connect directly to remote Sources. This can work if the number of endpoints is manageable and interactions are simple.
2. **Construct bridges or gateways in the middle**. Adding a middle layer allows you to uncouple your applications and substantially simplify OpenFin development.
3. **Adopt a multi-protocol broker in the middle**. If you want loose coupling and easier OpenFin development—and your data access is mission-critical—an enterprise-grade broker can harden the middle layer.

THOMAS KUNNUMPURATH, Solace's Vice President of Systems Engineering, Americas, will provide an in-depth look at the three different architectures used to connect OpenFin applications to different kinds of data sources. In doing so, he will highlight the pros and cons of each architecture while providing practical examples along the way.

“

By leveraging OpenFin, we’re able to seamlessly join multiple technologies together without impacting the end user.

KIM PRADO

GLOBAL HEAD OF CLIENT INSIGHT, BANKING AND DIGITAL CHANNELS TECHNOLOGY AT RBC

Source: American Banker

solace

INTRODUCTION: MODERNIZING FINANCIAL DESKTOPS WITH OPENFIN

Over the past few years, web-based technology has fundamentally changed how software is developed for consumer and business applications, resulting in faster, more user-friendly user interfaces and measurable productivity improvements. But adoption within financial firms has lagged, primarily because of that industry's special needs around security and regulatory compliance.

Software provider OpenFin identified this gap and created a new desktop OS (also called OpenFin) that containerizes web-based applications with an essential layer of security to protect against unauthorized external access. It also provides a host of other user- and developer-friendly features. OpenFin has clearly struck a chord with its audience, as more than 1500 firms now rely on it to power their user desktops.

DID YOU KNOW?

The OpenFin Mission:

to provide a unifying foundation for the next generation of financial industry applications.

CONNECTING OPENFIN APPS WITH DATA SOURCES

As OpenFin is transforming the client side of applications, financial firms are going through an equally powerful transformation in their service-tier applications. In the past, legitimate security concerns and regulatory requirements led financial firms to confine most applications within their datacenters, with all the cost and connectivity challenges that entails. The maturation of cloud security, cost pressures from CIOs, and regulators' increased comfort with the cloud have chipped away at the "never-cloud" crowd to the degree that even the most conservative firms have significant cloud adoption plans.

In addition, third-party information and financial service providers are adopting a cloud-first approach to reduce their own cost of providing services over leased-lines and other bespoke connections.

This paper focuses on the common architectures for custom buy-side applications that need to reach a variety of data sources, both inside and outside the firm.

FOUR QUESTIONS TO INFORM YOUR DATA ACCESS ARCHITECTURE

To successfully choose an OpenFin connection strategy, you need to first answer four key questions.

QUESTION 1. WHERE DOES THE DATA RESIDE?

OpenFin data can originate from many sources:

- Your datacenters (bare metal or private cloud)
- One or more of the popular public clouds (Amazon, Azure, Google)
- Private cloud or container frameworks (OpenShift, Cloud Foundry)
- SaaS applications (Salesforce, Fiserv, SAP)
- Hosted third-party applications (Bloomberg, Thomson Reuters, Morningstar)
- A hybrid cloud (datacenter + public cloud, or multiple public clouds)

Datacenter: Most server-side legacy applications still reside in the datacenter, and core trading applications like order management systems, market data feeds and risk engines often remain on bare metal for performance reasons. Some financial firms have begun deploying private PaaS and IaaS platforms like OpenStack, OpenShift or Cloud Foundry for less performance-hungry applications. In either case, since both the OpenFin user applications and the data sources are inside the enterprise, messaging middleware is the cleanest and easiest way to connect producers and consumers.

Public Cloud: The introduction of enhanced security services like AWS PrivateLink and Azure ExpressRoute have helped financial firms embrace the public cloud as a viable choice for deploying production workloads. For front-office app developers, cheap, secure and elastic compute and storage is appealing, and many firms are also adopting advanced data processing engines like Amazon Kinesis and Google BigQuery. It is much more likely that these services will expose RESTful APIs for inbound requests, although each service will declare its interface depending on its functionality.

SaaS and Third-Party Applications: SaaS and third-party applications publish APIs so your applications can interact with their data and processes. For example, the primary interface with Salesforce is REST, and the Bloomberg Open API can be accessed through a language like Python, .Net or C++. Each SaaS or third-party application will define its APIs and will support different means of connecting.

SUMMARY: *Answering question 1 identifies where the data is. For example, which datacenter, which cloud, which server-side application, which SaaS, etc. The answer to this first question will have implications for what API is best to reach the data, how you may need to deal with security, etc.*

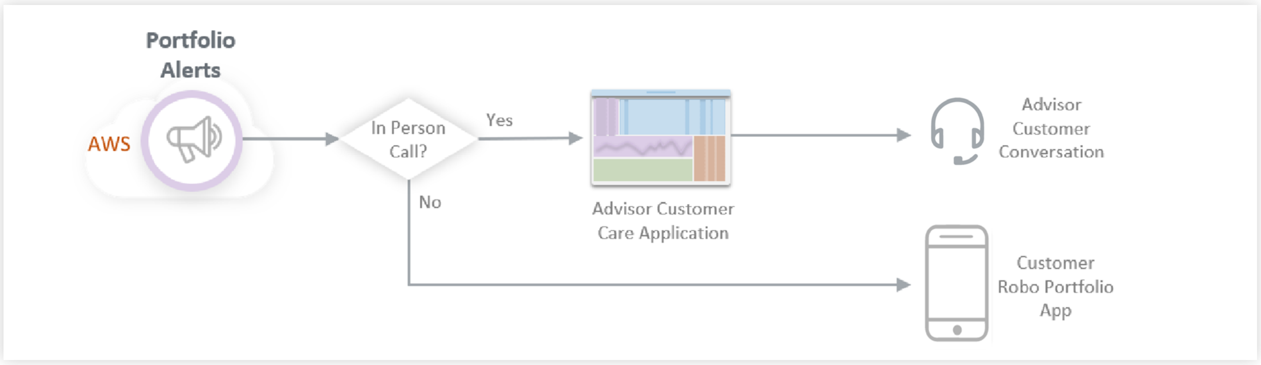
EXAMPLE SCENARIO.

CUSTOMER CARE DRIVEN BY PORTFOLIO EVENTS
(WE'LL USE THIS SCENARIO FOR ALL FUTURE EXAMPLES)

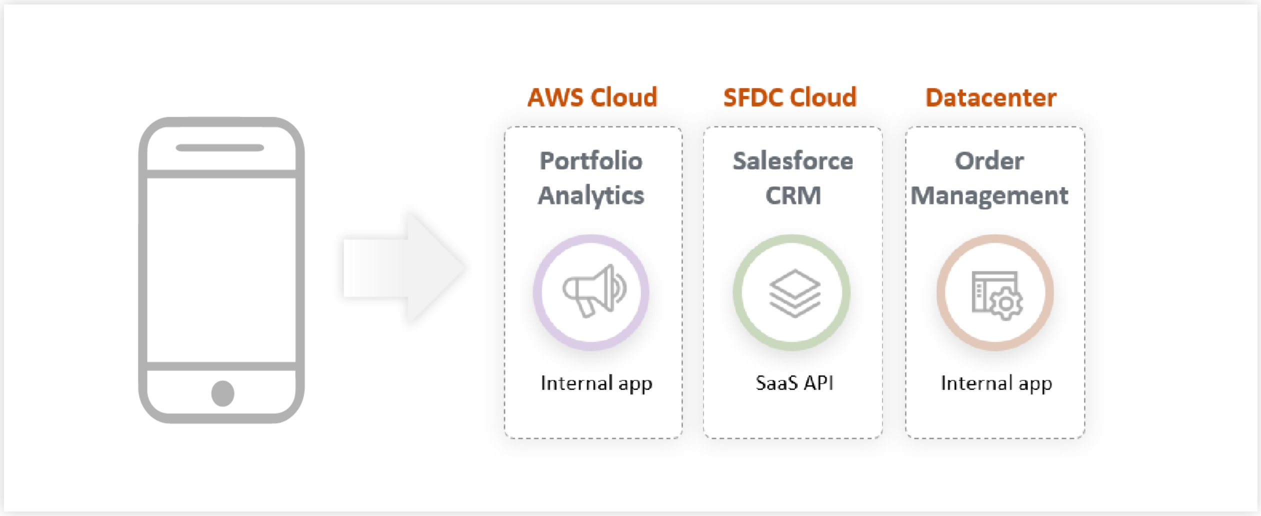
QUESTION 1: WHERE DOES THE DATA RESIDE?

A wealth management firm has determined that customer satisfaction scores are higher when they are proactive with outreach. The firm offers two choices to their client base: a call from their financial advisor, which is more popular among baby boomers, and a robo-advice mobile app, which is popular among their younger clients.

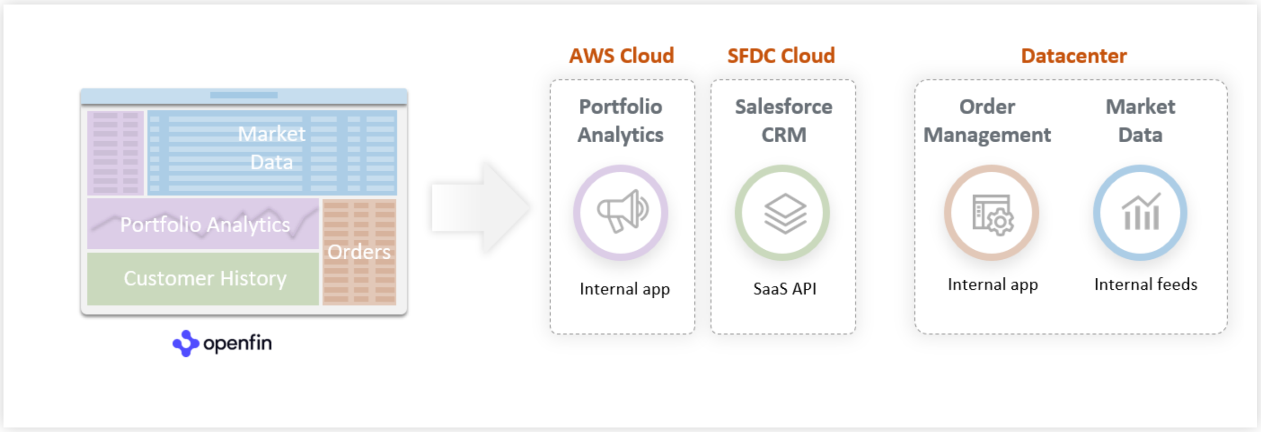
The workflow is as follows: an AWS analytics application has customer portfolio data and performs real-time portfolio analysis using Amazon Kinesis. Each customer's portfolio is continuously measured against a set of customer-specific metrics. When the portfolio falls outside the metric boundaries one of two outcomes occurs: the financial advisor for that client is notified with customer context pre-loaded and a set of recommended changes to discuss, or a robo-alert is generated to a customer-facing mobile app.


Portfolio Alerts
AWS
In Person Call?
Yes
No
Advisor Customer Care Application
Advisor Customer Conversation
Customer Robo Portfolio App

The mobile robo application receives the portfolio alert and interacts with the portfolio application in AWS to present the user with the best choices based on the user's criteria. It also interacts with the firm-specific order management system in the datacenter to initiate any trades, and the firm's Salesforce instance to capture all recommendations and order activity.



For the advisor-initiated call, the advisor's customer care application interacts with several systems: customer history is fetched from Salesforce, live market data is streamed from internal consolidated feeds in the datacenter, and orders are also executed against the firm's order management system.



Since the advisor application is a superset of the mobile robo-advice application, this paper will focus on the advisor apps' connectivity requirements as representative of both. Summarizing the answer to question 1 for this wealth management scenario:

	Data Needed	Location
1	**Portfolio data and alerts**	**Amazon AWS**
2	**Customer Data**	**Salesforce**
3	**Market Data**	**Datacenter**
4	**Orders and Confirmations**	**Datacenter**

DID YOU KNOW?

Financial firms will spend 47% of IT budgets on cloud by 2019.

Source: Thomson Reuters survey, July 2018

QUESTION 2.

WHAT DATA EXCHANGE PATTERNS ARE REQUIRED IN YOUR APP?

The next question is what kinds of data operations is your application looking to perform? Are you looking up a customer record? Kicking off a business process like a trade or order? Archiving or logging interactions for compliance? Receiving market data? Receiving risk or trade threshold alerts? These different types of interactions can be one-way or two-way, one-to-one or one-to-many, client-initiated or server-initiated, and require different levels of persistence to deal with failure conditions.

Of course, a single application can require multiple or even "all of the above," since a single application can have so many different workflows, each with different exchange patterns.

SUMMARY: *The answers to question 2 give you a clear picture of what kinds of exchange patterns are needed between the application and each of the data sources. If any of your exchange patterns are one-to-many, or asynchronous, you will likely be best served by a broker and a more capable protocol than REST.*

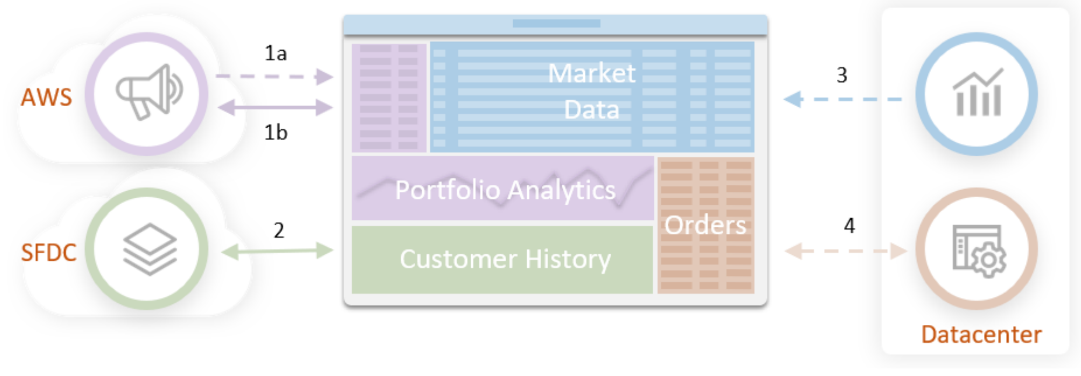
EXAMPLE SCENARIO.

QUESTION 2: WHAT DATA EXCHANGE PATTERNS ARE REQUIRED IN YOUR APP?

Returning to our example scenario, let's look at the data sources and the app's workflows with each:

- Portfolio alerts are the crux of this application, as they initiate customer service interactions, which is the purpose of the app. These events can occur at any time based on threshold calculations in the Amazon Kinesis cloud analytics platform. As a result, the OpenFin app needs to be listening for them at all times.
- Once a portfolio alert is received the application will set up its context with a series of request/reply interactions:
    - it will synchronously look up the customer's entire portfolio and trade recommendations from the portfolio application.
    - Customer history is synchronously fetched from the Salesforce API.
    - Real-time market data is received as streaming data, filtered to just the portfolio and recommendation assets.

- Initiating new orders is an asynchronous request/reply interaction with the order management system (since orders can take some time to complete due to limits, liquidity and other reasons).



	Data Needed	Location	Exchange Patterns
1a	Portfolio alerts	Amazon AWS	server-initated alerts
1b	Portfolio data	Amazon AWS	request/reply sync
2	Customer data	Salesforce	request/reply sync
3	Market data	Datacenter	server-init streams
4	Orders and confirmations	Datacenter	request/reply async

QUESTION 3.

WHAT ARE THE BEST PROTOCOLS TO REACH EACH DATA SOURCE?

As an OpenFin developer, you likely prefer using web-based interfaces such as REST and WebSocket because they are native to HTML5/JavaScript. However, many legacy sources won't have a web interface, which means you'll need to find another way. Fortunately, OpenFin provides means for connecting to nearly anything via native-language APIs in managed containers.

NATIVE WEB INTERACTIONS

When it's available, and you are performing a post or request/reply interaction, REST-based interactions are the most natural choice. You'll find that REST will be a good choice for many public cloud services, SaaS API interactions, and when your firm has embraced technologies like API gateways.

If your app calls for initiating one-to-many interactions, you will need either some server-side code to iterate to many endpoints, or to feed the REST call into a publish/subscribe network to handle the one-to-many fan-out.

When the script is flipped and you need to handle server-initiated events, you'll need something more powerful than REST. WebSocket is convenient because it can ride through the same proxies and gateways as your REST calls, but few of your data sources will speak WebSocket directly. This means you'll need to set up a middle layer to receive streaming data (usually from a messaging broker) and forward those updates through a WebSocket connection. The alternative is to use an OpenFin-managed container and bind the messaging API directly into the application. It's one or the other: make life a little easier for the OpenFin developer and add a new layer to the infrastructure, or push additional complexity on the OpenFin developer.

NON-WEB BASED INTEGRATION

To connect directly to non-REST APIs in the public cloud, third-party cloud services with real-time APIs, or messaging-based datacenter applications, OpenFin provides a couple of choices:

- Embed legacy code directly into OpenFin apps through managed containers. From OpenFin's perspective, anything that does not present a web-native API is "legacy" and access requires a binding to another language (like .Net, Java, or C++). Containers are OpenFin's way of isolating security exposure of external systems from data used elsewhere within OpenFin.
- Embed legacy code through Node.js addons or other kinds of connectivity bridges. This is the same principle as the first choice, but with the work moved to the server side.

SOMETIMES YOU HAVE TO CHOOSE BOTH

Remember, you may need to connect to a given data source differently for multiple data exchange patterns. For example, choosing REST for client-initiated requests and WebSocket for web-based, server-initiated alerts and a messaging protocol for streaming data, depending on what APIs are available for that data source.

These choices are discussed in much more detail in the next section: Three Architectural Choices for Connecting Data Sources.

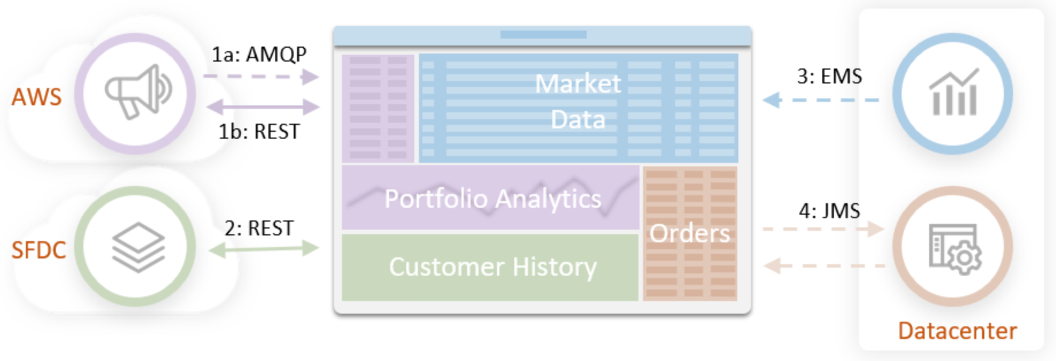
SUMMARY: *Question 3 identifies which APIs and protocols are supported for each exchange pattern and data source. If the APIs are all web-native, you may be able to architect entirely in REST and WebSocket. If your data sources include JMS, AMQP, or other messaging protocols, a broker to bridge to those environments will make operations smoother.*

EXAMPLE SCENARIO.

QUESTION 3: WHAT ARE THE BEST PROTOCOLS TO REACH EACH DATA SOURCE?

Let's layer in what we know about protocols in our example scenario:

- The portfolio services running in AWS use persistent AMQP queues to send alerts to the OpenFin application, which in turn uses REST for portfolio lookups. AMQP is accessible using the native Apache Qpid API.
- The customer care app calls the Salesforce RESTful API to look up customer records.
- A firm-specific consolidated market data feed is published through a legacy Enterprise Messaging Service (EMS) connection, using the Java EMS API.
- The order management system receives and sends messages via the Qpid JMS API.



	Data Needed	Location	Exchange Patterns	Exchange Protocol
1a	Portfolio alerts	Amazon AWS	server-initiated alerts	AMQP
1b	Portfolio data	Amazon AWS	Request/reply sync	REST
2	Customer data	Salesforce	Request/reply sync	REST
3	Market data	Datacenter	Server-initiated streams	EMS
4	Orders and confirmations	Datacenter	Request/reply async	JMS

QUESTION 4.

WHAT SECURITY IS REQUIRED BETWEEN THE OPENFIN APP AND EACH DATA SOURCE?

While OpenFin provides you with Application Identity and Group Policies to secure data within OpenFin's inter-application bus, you'll want to further assure that user security limits are respected in remote systems. For example, a user logged into OpenFin should be able to map their individual or role-based credentials into a remote system to allow appropriate access and, importantly, to restrict unauthorized access.

How this is accomplished will vary, depending on your firm's security architecture and the technologies you've deployed. Security for remote systems may include:

- **Web-based systems.** the most common means of connecting with web-based systems is to use OAuth to pass token credentials downstream to remote systems without exposing passwords. For example, user john.doe authenticated in OpenFin can initiate a REST call to a web-based system and see only the data that john.doe is authorized to see in the remote system.
- **LDAP or Active Directory integration.** Many enterprises rely on a central directory to authenticate users for database or message broker access. This technique works very much like OAuth, but is not limited to web-based technologies.
- **Access Control Lists (ACLs).** Where the remote system is accessible through a messaging platform, often users will be configured for user credentials or roles that allow access to select topics or queues to restrict access from others.
- **Other or proprietary approach**. Where none of these security approaches is supported, or where a unique legacy data source is required, you may have to fashion another security scheme to keep data flows safe.

SUMMARY: *Answering question 4 helps you plan for how security should best be established end-to-end for a given user or role. In general, OAuth is best where user-specific security matters (for example, a trader's unique portfolio holdings), and role-based authentication is best where groups of users share the same privileges (for example, market data access for any bond trader in the US).*

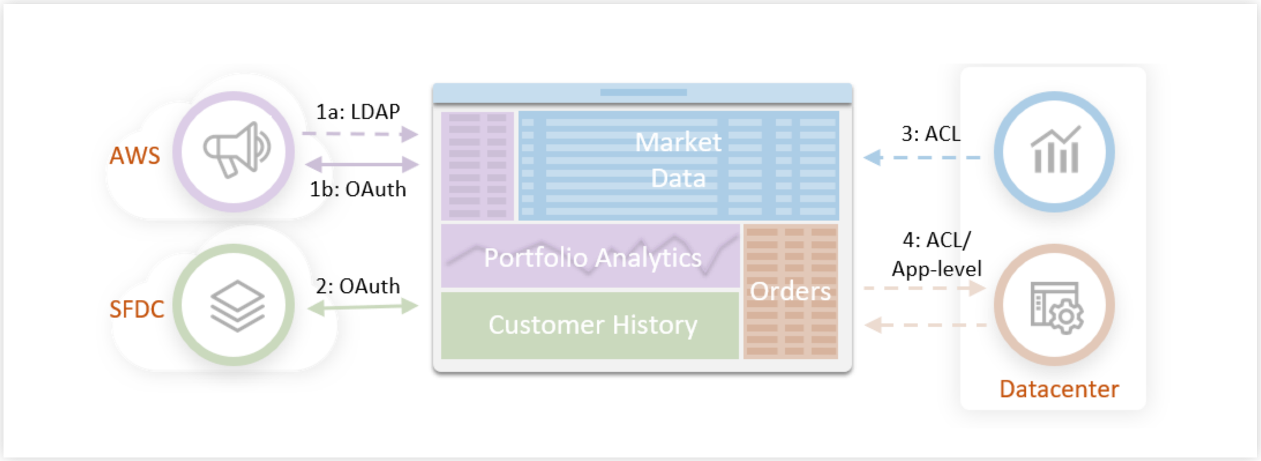
EXAMPLE SCENARIO.

QUESTION 4: WHAT SECURITY IS REQUIRED BETWEEN THE OPENFIN APP AND EACH DATA SOURCE?

In our example scenario, the security requirements of each system are a little different.

- Login to Salesforce CRM will use OAuth so all data lookups and all changes are attributed to the OpenFin credentialed user.
- The portfolio application authenticates using LDAP from the AWS cloud to initiate outbound requests through AMQP. For portfolio lookups, it uses OAuth to assure only valid portfolio managers can see their client's account details.
- The market data system maps users into roles configured as ACLs at the EMS server, since all portfolio managers have the same privileges in that system. It submits a unique token at login for auditing to track which users are using the feeds (required by their market data licensing agreements).
- The order management system also uses ACL mappings and requires each message to include the authenticated OpenFin user token along with the customer ID to provide an application layer check that the user is permitted to trade on behalf of that client. In this case, transaction-level security is handled in the application.

As you can see from this example, these specifics will vary with the security profile of each end system.



THREE ARCHITECTURAL CHOICES FOR CONNECTING DATA SOURCES

If you've answered these four questions, you've got a solid picture of your app's connectivity needs. But before you get to work hooking up endpoints, it's worth stepping back and looking at your three architectural paths forward.

ARCHITECTURE 1.

MANAGE CONNECTIVITY IN THE APP

The most direct option is to directly connect your OpenFin apps to each data source. Let's weigh the tradeoffs of this approach:

PROS:

- **Independence.** The primary reason to consider this approach is if you don't have the option of deploying a middle layer, or if the idea of working with your infrastructure team to do so gives you the cold sweats. Embedding all the connectivity directly into the app gives you just one throat to choke—your own!

CONS:

- **OpenFin development complexity.** Depending on the protocols you're connecting, your application may need multiple managed containers. Most OpenFin developers are web developers first, and will have to deal with calling out to other languages like C++ or .Net. Each connection binding will also have its own set of error handling for failure scenarios (back pressure, flow control), unique to each protocol. Furthermore, your application code will have to implement end-to-end security to each of the systems that application needs to access.
- **Tight coupling.** By forgoing a middle layer, your applications will be embedding the details of each connection directly within the apps. If the data source moves or its data interfaces change, you'll need to modify all the apps accessing that data source. The benefits of loose coupling are many, and this is a big reason why messaging architectures are so pervasive in financial services.
- **Substantial additional testing.** Agile development may allow for quickly hacking these many connections together, but making the code bulletproof will require a lot of testing. Each connection comes with its own corners, which means there are more corner cases to handle.
- **App bloat**. Embedding all those APIs and the libraries they pull into your app will increase the memory and resource consumption and could impact application or overall system performance—particularly if you build many desktop apps with this strategy, each of which handles connectivity on its own.

If all of your endpoints have REST interfaces, or you're connecting just one remote data source, this might be the best approach for you. Applications that need to access mostly non-web data sources will be better served by one of the next two options.

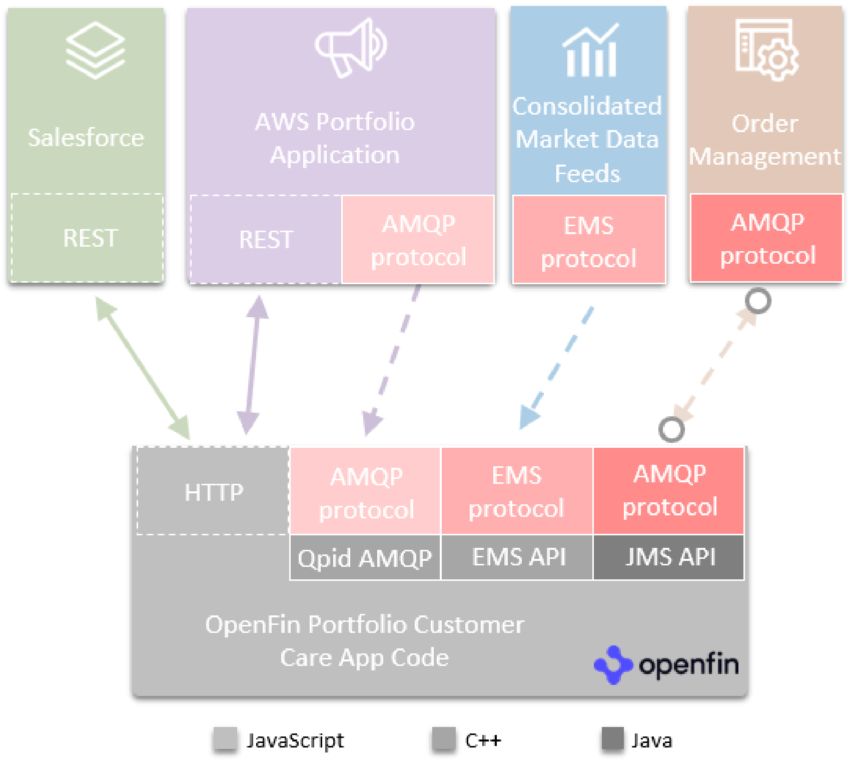
EXAMPLE SCENARIO.

ARCHITECTURE 1: MANAGE CONNECTIVITY IN THE APP

In our scenario, for the app to take responsibility for all connectivity, it has to be prepared to speak each of the protocols directly within the client app. The application uses native HTTP URI requests to initiate the REST calls, and embeds the AMQP, EMS and JMS APIs and protocols within OpenFin-managed containers.

Note that for simplicity, the following diagram does not show the messaging brokers or daemons that would exist between the clients and servers. The connection is a logical connection across the underlying infrastructure. Note also the APIs on the server side are not shown because this paper focuses on OpenFin development, which is independent of server-side

API choices.



In contrast to the next two architectures, you can see that the client-side code is much more complex.

ARCHITECTURE 2.

GATEWAYS, BRIDGES, ADAPTERS AND WRAPPERS

The flipside of having “nothing in the middle” is having something, and Architectures 2 and 3 will look at two ways to proceed with a middle layer. The largest system implication of choosing to go with a middle layer is that each endpoint can be simplified to support fewer connectivity options with the differences being resolved centrally. This means that an OpenFin app that would most naturally speak REST or WebSocket, and needs to reach an AMQP data service, can rely on an intermediary to do the translation. The job of the intermediary is to terminate the first protocol and initiate the second while preserving the payload and security properties.

This approach goes by many names depending on where the translation takes place:

- **Gateways or bridges.** These are standalone processes that connect two or more communications methods and handle messaging protocol or security schema differences. For the most popular pairings, you may be able to find some open source or commercial options for these bridges, or sometimes you may need to build them yourself.
- **Adapters.** Adapters are essentially plugins to one environment allowing connection to another. For example, a JMS message broker may have adapters that are aligned with the architecture of that broker and connect to other protocols, such as MQTT. It should be noted that if you’re using open source or vendor brokers for adapters, the adapters may or may not exist, or the products may or may not allow you to create your own adapters.
- **Wrappers.** Wrappers are API adaptations for a specific language. While you could wrap APIs on the client side using the managed container approach in Architecture 1, it is much more common for wrapping to occur on the server side. For example, you may want to allow a data source to be accessible via REST in Python to abstract the specifics of a legacy interface like MQ or a SQL database API.

The right option for you will depend on your firm’s philosophical approach to IT, the details of the specific pairings you need, and the infrastructure already in place that might be leveraged. For the sake of this document and its examples, let’s assume it is possible/ practical to pair up all combination of different endpoints needed for your OpenFin data access. Let’s take a look at the good and bad of this approach.

To avoid saying “gateways, bridges, adapters and wrappers” each time, let’s group these conceptually together as “intermediaries”:

PROS:

- **Simplifies OpenFin app development.** With protocol differences resolved through intermediaries, your OpenFin apps can be simplified to make all their calls directly from JavaScript. Those might be REST calls, receiving streams via WebSocket, or (in the case of some JMS products) handling JMS connections directly from JavaScript. All other interactions will use an intermediary so your client can use JavaScript and still reach legacy data sources.
- **Leaner applications.** By isolating all OpenFin interactions to only native web protocols we eliminate the app bloat from the prior architecture.
- **Loose coupling.** The coupling happens in the configuration (sometimes static, sometimes dynamic) of the intermediaries, and each application is only bound to its intermediaries. If a data source changes locations or data formats, one change in the middle eliminates the need for all apps to be updated.

CONS:

- **New layer to manage.** The intermediaries will need to be managed and maintained, just like the data sources they're accessing. The degree of management pain will be proportional to the number and complexity of intermediaries deployed.
- **Development complexity in the middle.** The work you don't do in the OpenFin applications you are going to have to do in the middle. The good news is you only have to set up or build each intermediary once, after which new interactions are at most configuration changes. Depending on how many different intermediaries you need to stand up or build, it can be complicated to develop a cohesive security strategy to enforce which users can see what data since it is unlikely that there will be just one way to reach all sources.
- **Long deploy/test cycles.** The number of moving parts is much higher, so the amount of testing is also going to be higher and it will take more time to debug issues.
- **Operationally fragile**. More moving parts means more things to go wrong. More monitoring and tooling will be required to keep the network of intermediaries up and running.
- **Independent scale and resilience.** These intermediaries become part of your critical path, which means you'll need a way to assure high availability and scale beyond a single instance as your OpenFin application use increases.

Once a reliable set of intermediaries is in place, the benefits of this approach will accelerate OpenFin app development significantly, with some new work for the DevOps team keeping the middle layer running.

EXAMPLE SCENARIO.

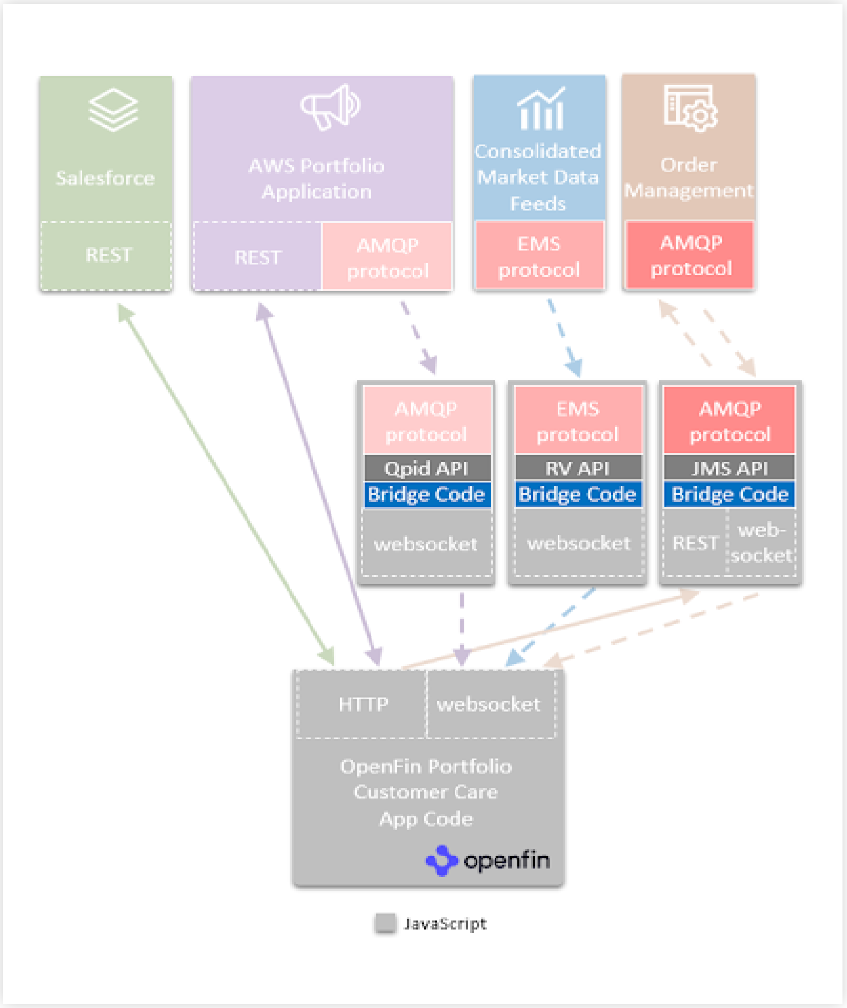
ARCHITECTURE 2: GATEWAYS, BRIDGES, ADAPTERS AND WRAPPERS IN THE MIDDLE

Applying Architecture 2 to our example scenario, the client side of the application is much more simplified, with the OpenFin developer using only native protocols like REST and WebSocket to send and receive synchronous and asynchronous messages.

Instead, the environment differences are resolved in the intermediaries that speak web-technologies when interacting with OpenFin apps, and the data source's native protocols to access data.

In our scenario, the middleware team supporting the OpenFin initiative chose to build the bridges themselves. The code to terminate one message receipt and initiate the next is not overly complex; most of the work in coding this kind of bridge is in operationalizing it since it will need:

- A means of mapping REST parameters to messaging semantics (as shown in the JMS request/reply interaction). This may be as simple as a configuration file if the mappings are fixed and static, or it could need to be dynamic if the client is subscribing and unsubscribing from topics.
- A means to scale past one instance.
- A means to provide resilience if the bridge fails.
- A way to preserve access control and possibly encryption tokens across protocols.
- To be instrumented with logging or monitoring hooks to track its operational details.



ARCHITECTURE 3.

INTEGRATED MULTI-PROTOCOL BROKER

The third approach is an evolution of the second, with the OpenFin application still using its native web protocols REST and WebSocket, and the cloud and datacenter services offering their native interfaces via protocols such as REST, AMQP, JMS, MQTT and WebSocket. But instead of a set of pairwise, handcrafted intermediaries, Architecture 3 uses an integrated multi-protocol broker—Solace PubSub+.

Solace is uniquely suited to connect OpenFin applications to the many data sources they need to reach because:

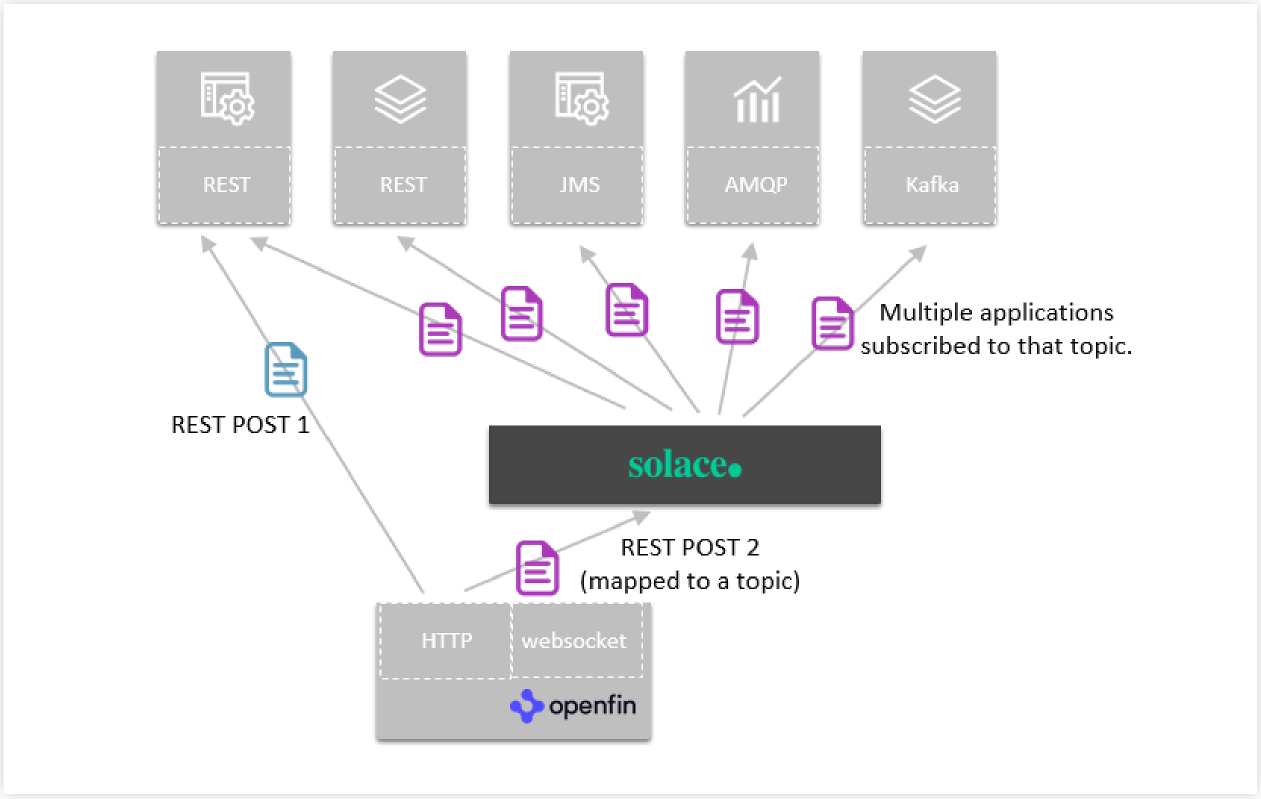
- **It's multi-protocol by design**. Solace natively speaks all the open standard web and messaging protocols, including REST, WebSocket, AMQP, MQTT, and JMS. It can further bridge to proprietary protocols like MQ, RV, EMS, Kafka and others directly from the broker.
- **It's multi-platform and multi-cloud**. Solace can be deployed on bare metal, in private cloud PaaS infrastructures like Cloud Foundry and OpenShift, and to public clouds like AWS, Azure and Google Cloud Platform. There's even an as-a-service offering for hybrid cloud that seamlessly connects with your datacenter.

When it's time to scale, you can elastically do so through multiple Solace message brokers which automatically coordinate to form an event mesh. Scaling, resiliency, security and connectivity across any cloud boundaries are handled automatically.

PROS:

- Simplifies app development. With protocol differences resolved in the event mesh, your OpenFin apps can be simplified to making all their calls directly from JavaScript. Likewise, a data service or legacy resource can use its native access protocol, for example AMQP or JMS, without regard to which applications need access. Security is also handled end-to-end through ACL mapping to topics, reducing the variety of security approaches required in the second architecture.
- Leaner applications. As with the second architecture, by isolating all client-side calls to just native protocols we eliminate the app bloat from the first architecture.
- Loose coupling. All endpoints only need one binding—to the Solace broker. If any data sources move or data structures change, the changes are isolated away from the endpoints and can be handled in the middle. That means server-side changes do not require applications to change, and vice-versa.

- Fast deploy/test cycles. You can get started developing using 100% Javascript in minutes on your local laptop, in the public cloud, or by spinning up an instance with Solace PubSub+ Cloud. Testing corner cases and failure conditions is far simpler because the number of messaging networks has been reduced to one, which is already unit-tested before it ships.
- Operationally stable. High-availability, fault-tolerance, elastic scaling, and monitoring also come out-of-the-box so your event mesh can be counted on through service outages and traffic spikes.
- Publish/subscribe through REST. There are times when it would be convenient for a single REST POST to behave more like a publish event, with multiple subscribed recipients. By directing REST to the Solace broker, it can result in a variable number of endpoints receiving the message. Security is handled through ACLs configured in the Solace broker.


REST
REST
JMS
AMQP
Kafka
Multiple applications subscribed to that topic.
REST POST 1
solace
REST POST 2 (mapped to a topic)
HTTP
websocket
openfin

CONS:

- New layer to manage. There's no getting around the fact that "something in the middle" means something to manage, which was not required in the first architecture. However, when compared to the second architecture which required creating and managing many intermediaries, with Solace you have just one thing to deploy, operate and manage to handle most endpoint pairings. If you're deploying in one or more public clouds, you also have the option of consuming Solace "as a service" with the PubSub+ Cloud offering. So while Solace does have to be managed, it's a fraction of the effort of the second architecture.

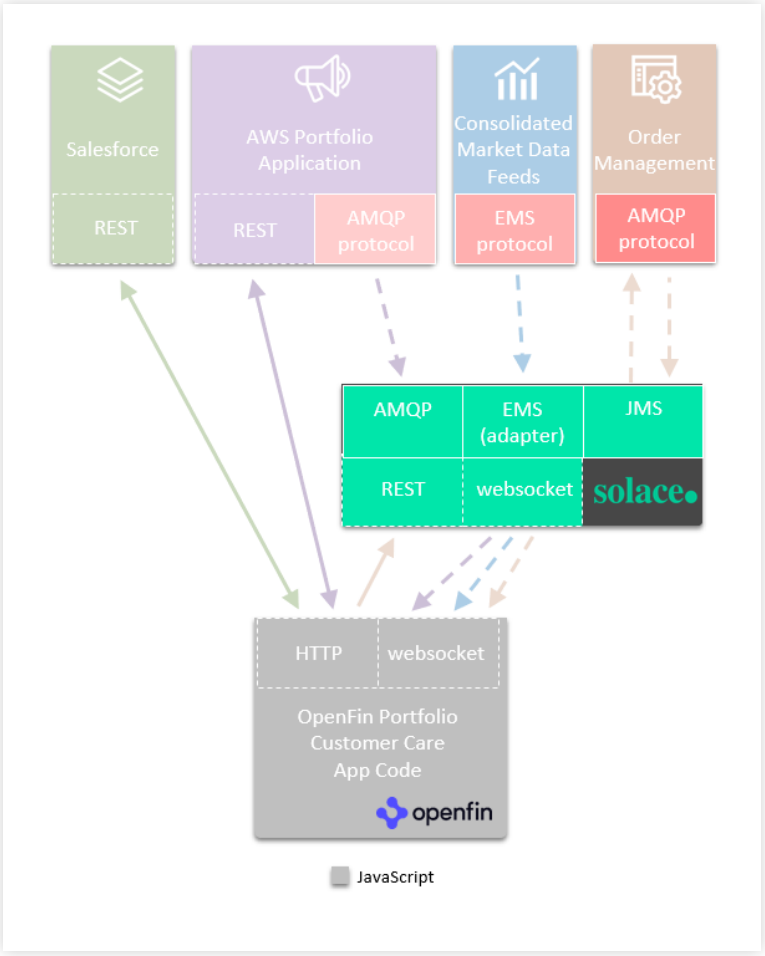
Architecture 3 gives OpenFin app developers a loosely coupled, reliable and secure infrastructure that can reach across any topology—from datacenter to public cloud to managed service. It frees OpenFin developers to stay agile and keeps their apps lean and efficient.

EXAMPLE SCENARIO.

ARCHITECTURE 3: GATEWAYS, BRIDGES, ADAPTERS AND WRAPPERS IN THE MIDDLE

In our example scenario, Architecture 3 simplifies the middle layer by eliminating the need to custom-build any bridges or gateways. All of the endpoints are native open protocols, with the exception of EMS, which is reached through a Solace-to-EMS adapter.

The major difference is that unlike the pairwise intermediaries in Architecture 2, the Solace PubSub+ broker used in Architecture 3 is just one thing to install, one thing to manage, and one thing to scale. It is worth taking a closer look at that difference.

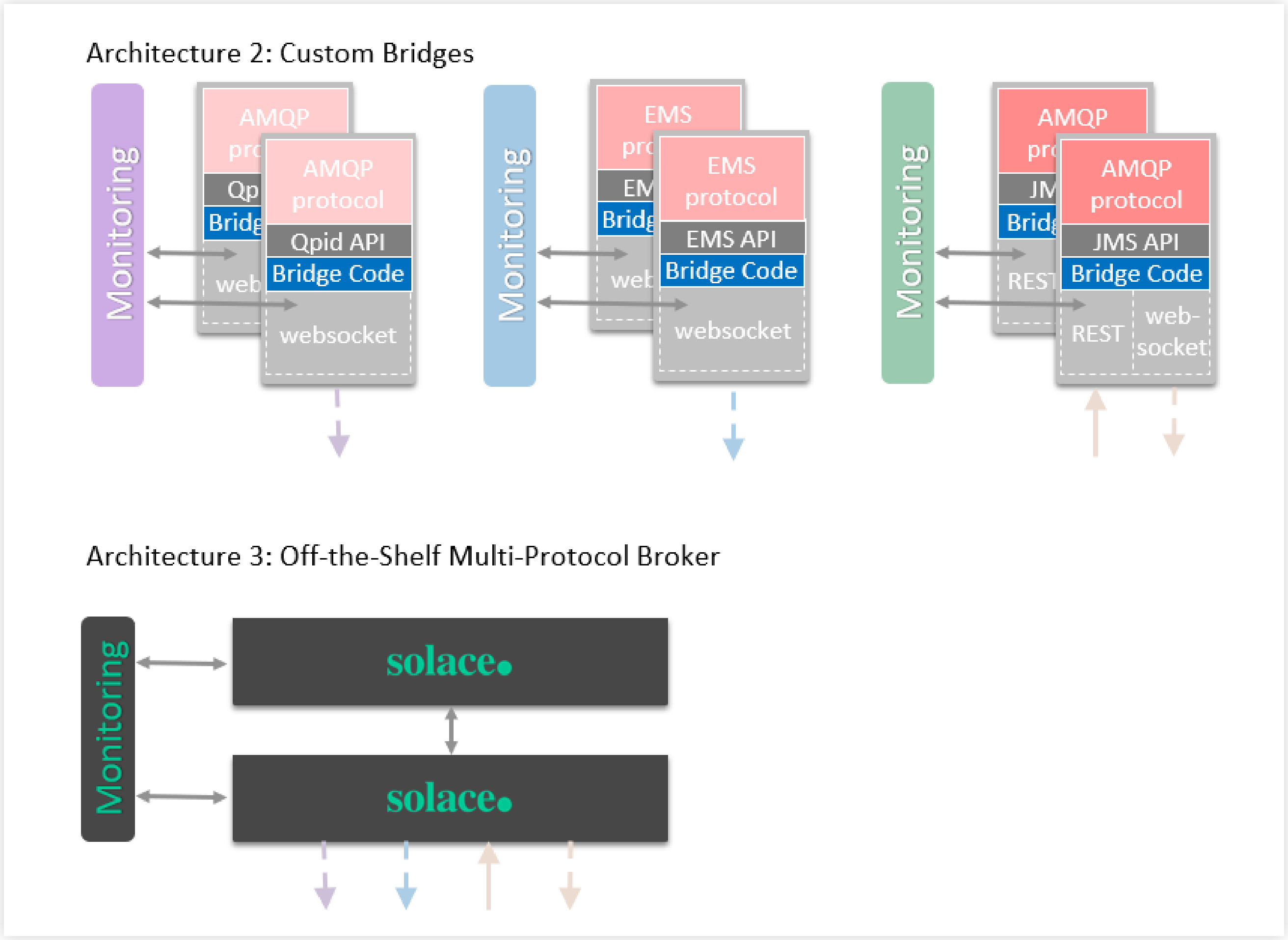


UNDERSTANDING THE DIFFERENCES BETWEEN ARCHITECTURE 2 AND 3

You can see from our example scenario that Architectures 2 and 3 are identical on the client side, and only differ in the middle. On the whole they look very similar, with the difference appearing to boil down to build vs. buy. In real-world scenarios, there's much more to the story.

Here are three key differences:

1. **Scaling and resilience**. If you choose a middle-layer approach, the connectivity bridges need to be at least as scalable as the aggregate data volumes to downstream systems. Let's look at Architecture 2 and 3 with just redundant pairs added, before even considering scaling. You can see that Architecture 2 gets complex quickly. Management and monitoring, in particular, become concerns as the number of moving pieces grows.



2. **The number of bridges can get overwhelming**. Our example scenario uses just three messaging networks, but most firms have ten or more. They can be a mix of open, legacy or proprietary protocols such as REST, WebSocket, AMQP, JMS, MQTT, MQ, RV, Ultra Messaging, EMS, Kafka, Amazon SQS, Google Pub/Sub, etc. With Architecture 2, you need a bridging strategy for each pair of protocols. In our OpenFin-centric example scenario, the bottom pair is always REST and WebSocket, which keeps things simple.

   In the real world, anything may need to bridge to anything. For example, an AMQP data source may want to share with a source accessible only by MQ or Kafka. That's two more bridges. With Solace, all combinations can be handled using the PubSub+ broker. Open protocols and APIs like AMQP, JMS and MQTT are native to the broker, and other protocols are integrated using a bridge (pre-built bridges exist for many legacy environments).

3. **A consolidated security strategy**. Architecture 2 will likely lead to a pairwise authorization strategy, depending on how each endpoint can be accessed. In Architecture 3, Solace uses the same authorizations approach for all endpoints. This is accomplished by the use of topic-based Access Control Lists (ACLs) where only authorized users are allowed to publish or subscribe to specific topics or queues.

Architecture 3 greatly simplifies operations and scaling, and keeps you out of the business of building complex, production-grade infrastructure. If your environment is mission-critical or needs to reach many endpoints, Architecture 3 will result in easier development and management.

CONCLUSION

OpenFin is changing how financial desktops are delivered, making it easy to embrace web-based technology and agile development methodologies. Architecting for reliable data access requires carefully considering your objective and needs. OpenFin developers will need to weigh the tradeoffs of directly connecting to more than a very small number of endpoints, as complexity and app maintenance can grow substantially.

When choosing to offload that work to a middle layer, the decision point changes to consider the number of endpoints, types of interactions, security, and resilience requirements. The degree to which this information is mission-critical to your OpenFin users will inform whether you choose to build one or more data bridges, or if it makes more sense to invest in a multi-protocol broker that takes care of these needs for you.

Thomas Kunnumpurath **is Solace's Vice President of Systems Engineering for the Americas. His past experience includes doing full-stack development for distributed trading systems for an investment bank. As such, he is acutely aware of the challenges with the traditional way of developing trading desktop solutions and is excited about using Solace, a world-class advanced event broker, with OpenFin to help bring trading desktops into the 21st century.**

ABOUT SOLACE

We are the creators of PubSub+, an advanced message broker that can be used to create an event distribution mesh. Solace provides the only unified advanced event broker technology that supports publish/subscribe, queueing, request/reply, message replay and streaming using open APIs and protocols across hybrid cloud and IoT environments. Our smart data movement technologies rapidly and reliably route information between applications, devices and people, as well as across public and private clouds. Established enterprises such as SAP, Barclays and the Royal Bank of Canada, high-growth companies such as VoiceBase, and industry disruptors such as Jio use Solace to modernize legacy applications and successfully pursue analytics, hybrid cloud and IoT strategies.

A FEW OF OUR CUSTOMERS

airtel BARCLAYS bet365 NASA SAP

RBC Standard Chartered FxPro Liquidnet TMX

OUR FEATURED PARTNERS

DELL Boomi Pivotal SAP f5 openfin

solace.

solace.com