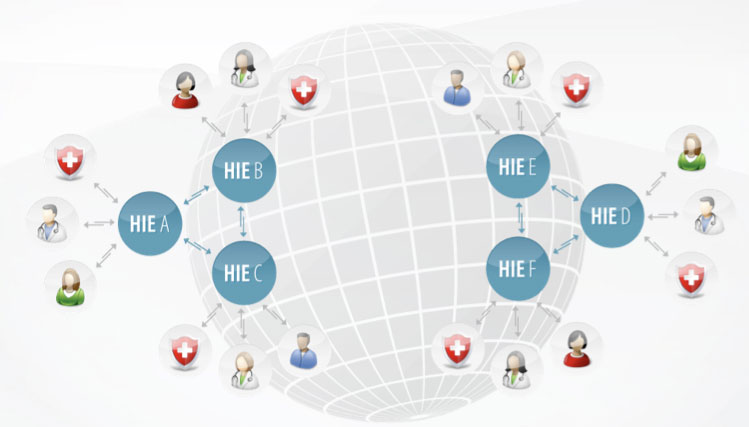
# The Role of the HIE in E-Health Reform



*A Health Information Exchange is defined as the mobilization of healthcare information electronically across organizations within a region or community.*

*HIE provides the capability to electronically move clinical information between disparate health care information systems while maintaining the meaning of the information being exchanged. The goal of an HIE is to facilitate access to and retrieval of clinical data to provide safer, more timely, efficient, effective, equitable, patient-centered care*

**

*Overview* 3

Health Information Exchange 3

Health Information Exchange Technology Stack 5

Foundation 5

Platform Administration Console 5

Anatomy of a Service 7

Anatomy of an Application 8

## *Overview*

E-Health is upon us. The vision entices; patients, doctors and hospitals around the world sharing medical records securely through the Internet. However, the vision is currently just that… a vision. You hear about hospitals adopting EHRs. The reality is that about 1.5% of hospitals are embracing EHRs and the systems they are using work on a “closed loop”. They aren’t interoperable. This is important information to know before sinking $19.2B[[1]](#footnote-0) into building more e-health solutions that do the same.

Creating a completely interoperable system is a very complex task. There are a lot of pieces that need to come together for it all to work. In this paper, I want to cover what we consider to be a “high probability of success” solution. We believe in a centralized solution where there can be federated security and regulations. A solution that works equally well on local levels as it does for the global populace. A solution that can grow in the decades to come and that can give every man, woman and child a better, healthier, future.

## Health Information Exchange

There are two schools of thought when it comes to dealing with modernizing healthcare. The first is the more laissez-faire approach where companies will create software to solve their client’s problems. For example, the client, in this case a hospital, wants a bed management system and they want to pay as little as possible. The technology vendor builds the system to specifications and the client is happy. This approach works and is a natural stage of software development. The evolution of this type of development is that the vendor finds himself writing the same program over and over again for different clients and soon realizes he should be creating a product with all the existing code he has written. The product gets new features every now and then and clients pay for license fees and support contracts.

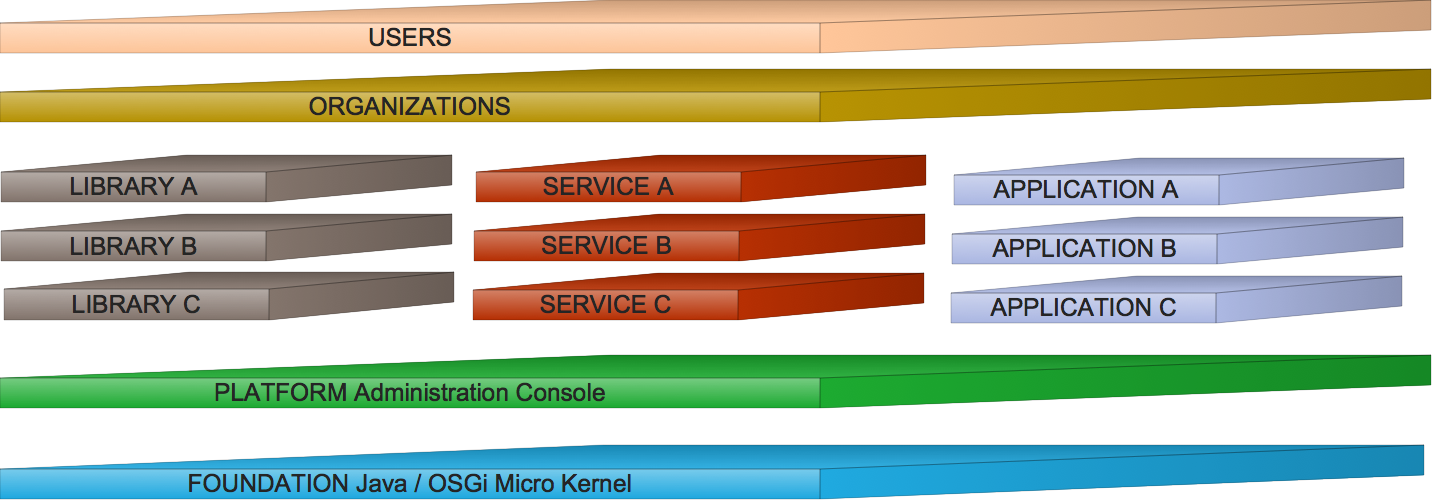
The problem arises when these software applications start to multiply and it becomes costly for the hospital to manage multiple disparate applications that do not communicate with each other and require separate installations. At this point, a hospital wants to simplify and they will look for a vendor that can offer them everything they need in one big package. This comes at a cost and hospitals will pay a premium for these software solutions. The end result is that the hospital spends a lot of money on software infrastructure that gets passed on to the patient but the patient does not derive real benefit because the systems themselves are not interoperable with other external systems the patient is connected to, such as another hospital system across the street.

We could take the argument of the laissez-faire approach one step further to exaggerate the eventual outcome. Hypothetically, if we assume that technology vendors adopt the right standards and protocols and are able to interoperate, the result will be 100000+ installs of interoperable systems with no federated security or user base and no centralized intelligence that can delegate the flow of information. In other words, it is not a feasible solution.

The second approach is highly contested for being too risky a venture because of its likelihood of failure. It assumes a centralized system where data flows through to the consumers who are the hospitals, the patients, and the doctors. It moves healthcare IT out of hospitals and onto the “cloud” (massively parallel computing grid) that can cater to all hospitals instead of one. Software has to be written for the distributed enterprise where services are always available, to everyone, with a federated security system guarding all data points. This is a Health Information Exchange, a medical data aggregator.

I would venture to say that the latter is indeed in the realm of possibility and, with a little bit of planning, can be a strong candidate for managing our country’s health in the decades to come. We have built a software platform we believe can serve as the infrastructure for future e-health applications to build upon. The platform was built for that very purpose. To support our own e-health services. Below, we’ll go into detail about how our platform was created.

## Health Information Exchange Technology Stack



### Foundation

Our foundation is written in Java and runs inside an OSGi micro-kernel that supports versioning and hot-swapping of libraries and services at runtime. This gives maximum flexibility when needing guaranteed uptimes of 99.99% and supporting services that require different versions of libraries within the same enterprise environment. The OSGi micro-kernel lives on top of an infinitely expandable, distributed cloud computing grid that has enough computing power to service the entire planet.

*Example*

*We have an existing VoIP service running on the platform and we’d like to install a newer version that both fixes some bugs and adds some new features. In a non-OSGi environment, we would have to stop the Java container and re-deploy the new version and any new third-party dependencies that it relied on. In an OSGi environment, we can deploy new libraries at runtime and upgrade our services without stopping the container.*

### Platform Administration Console

The admin console is a web application that gets deployed on top of the OSGi micro-kernel. The admin console has many responsibilities:

#### Deploying artifacts

There are 3 different types of artifacts that get deployed through the admin console.

##### Library.

These are OSGi compatible Java libraries (jars) that are required by the services. Libraries need to be installed before the services that depend on them are installed.

##### Service.

A service is also an OSGi compatible Java library but that’s where the similarity ends. Thanks to the Spring framework, these jars each contain their own Spring context that gets initialized at time of deployment. Once initialized, you can make some of Spring’s components available as OSGi services by publishing them in the OSGi service registry. For example, the first service you might wish to publish could be the data source bean to a common database other services are going to rely on. You could also publish a complete set web services using the same technology. Your choice of data transport really depends on who the consumer is and where he is. If the consumer is internal to the micro-kernel there is no need for slow processes such as web services because the consumer has direct access to the service reference of the underlying business process. Building a service that can support all types of consumers is covered below.

##### Application.

An application is created using Adobe Flex or Adobe AIR technologies. While the services are the intelligence of the system, the applications are the ones utilizing the services on behalf of the end users. When the right services are in place, such as a PHR service, the end user, or patient, is able to access her PHR through the Adobe Flex application that knows how to talk to the service. So every application needs one or more services in order to function. Both need to be deployed using the admin console.

By creating services and applications using a component-based architecture means that we have greater leverage over where these artifacts are deployed. One service could be deployed with HIE A while another could be deployed with HIE B. Upgrading applications for all users is done quickly as it is a centralized install. Reverting back to previous versions can easily be done thanks to versioning.

#### Manage users and their roles

Users in an HIE need to be portable because there will be more than one HIE in the world. If a user was created externally to this HIE, the system needs to query the creating entity for that information. We are using openID for this purpose. OpenID can verify the authenticity of the user as well as any roles the user might have both for externally and internally created users.

#### Manage organizations

Our concept of organization is a broad one with many responsibilities. They live on our platform as producers and consumers of services and applications. Our company, Health XCEL Inc, is one of many organizations that are registered with the platform. The organization is made up of users who are our members. We have also created and deployed services to be consumed by us and other organizations. Many of our services we offer up at no charge while others require a license. Along side our services we have published the necessary applications that end-users can interact with.

When an organization licenses an application for consumption from another organization, the system validates the license and grants users from within that organization access to its services. Users within an organization can have an extra set of roles in addition to the global ones. That way, a user could be an administrator of one of the applications owned by her organization.

#### Manage regulatory functions

Every country has regulations that software systems need to be cognizant of when operating in those countries. The admin console provides a way to create a super-set of meta-rules that can be overridden on a per-country basis. Applications and services can leverage those functions at runtime to give the users the right data for the country they are in.

### Anatomy of a Service

A service needs to be able to do two things right. Control who can access it and distribute information in a legible manner.

#### Access Control

An entity wishing to access a service needs to be registered with the HIE. The entity can be internal or external to the system but it needs to be registered as an authenticated entity before it is allowed access to data. An entity is an organizational entity, as described above. Before Hospital X can access real-time data from Source Y, Hospital X needs to be a registered consumer of Source Y.

#### Data distribution

In any system there is data and there are ways to transport the data from one place to another. In a single system, the path is easy as it usually goes from the database to the user and back to the database. The data is known as the schema and the UI are proprietary. When a system is comprised of many interrelated systems, data and data transport need to be standardized.

A service, such as a PHR service, needs to serve up data according to a PHR specification. It also needs to be able to send the data across common channels such as HTTPS. As long as a standard exists, the service knows how to persist data. In the world of HIEs, without an adequate standard, there won’t be a service as it will be useless. No other party would know how to consume your data.

The last step as part of data distribution is the ability to mesh data together. It’s a way to bring related data together from different channels and export it as part of a larger data set. This is also what people refer to as the Semantic Web[[2]](#footnote-1) and what we believe will be the future of e-health.

The topic of data distribution would not be complete without mentioning the Enterprise Service Bus.

##### Enterprise Service Bus – ESB

For every service, there is data coming in and data flowing out of the system. There are also many different types of callers, internal and external to the OSGi micro-kernel. Callers might be new or legacy systems that use the same standard but different types of data channels. How can one easily support a wide variety of data channels in a cost-effective way? The Enterprise Service Bus!

The ESB supports many different types of incoming and outgoing data channels. These channels could be HTTP, FTP, SSH and others. The data going through those channels could be XML, JSON, binary, HL7v3 and others. In essence, we can now focus on the business problem while the ESB takes care of the plumbing.

### Anatomy of an Application

An application is created using Adobe Flex technologies and is the main consumer of services. There are two types of applications, the “platform” and the “application”. The platform is a smart shell that knows how to launch other applications within itself and then give them certain “street-smarts” so they know how to do their job. Both types of Flex application have several responsibilities.

#### Access Control

Every application will have access to the user’s credentials and roles. Based on that, the application is smart enough to give access to only those parts of the application the user can work with. That might mean a button is disabled or removed or whole chunks of the application are either not available or just accessible in read-only mode.

#### Locale-based behavior

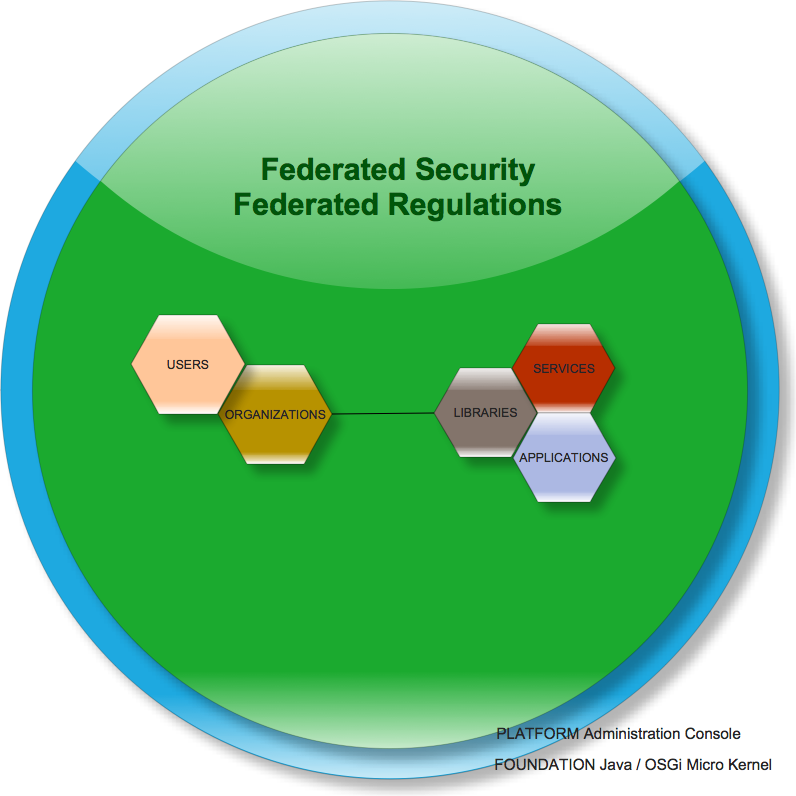
As we mentioned before, the administration console has the ability to manage regulations on a per-country level. These regulations are implemented in the application and should the end-user happen to be in a country where regulations stipulate that certain processes are different, the application needs to cater to those regulations accordingly.

#### Injected intelligence & Customizations Per Owner and/or Licensee

When an application starts, it will be injected with a set of initialization parameters that have been specified by the creating organization and that can be overridden by its licensees. Initialization parameters affect the behavior and the look-and-feel of the user interface. Organizations who take great care in exposing the right elements of the application through parameter loading will see far greater success when licensing their application to third parties.

*Example:*

*Organization Y wants to license Application A from Organization X. The licensee (Org. Y) wants to use its own color scheme to match its existing applications. The licensee also wants to modify the how many entries the end user is allowed to make within the application. The licensor (Org X) already supports changing all the colors of its application, and the number of records the end user is allowed to make, via initialization parameters.*



1. President Obama’s health care stimulus package [↑](#footnote-ref-0)
2. http://en.wikipedia.org/wiki/Semantic\_Web [↑](#footnote-ref-1)