elasticWarehouse

The Elastic, Content-Managed Warehouse System

Robots, goods-to-person, new previously unheard-of automation – that is the story of warehouse systems in 2020. So, what are vendors and, correspondingly, customers to do?

On the one hand, the warehouse systems vendors want customers to stay *current* on their latest and greatest so they can, understandably, minimize the variability in their support organizations and optimize their SaaS revenue. We get it. It is very difficult to maintain knowledge and staffing to support many past releases while meeting contractual service level agreements with customers. But, how on earth do keep customers current while the warehouse landscape shifts under your feet. Some customers, for example, want to work with Robot Vendor X, others with Goods-to-Person Vendor Y, and still others want to work with Virtual-Reality Vendor Z. How do you make it all work – all under the umbrella of one multi-tenant, multi-site Warehouse System?

We have a solution; one that we've been thinking about for the past 2-3 years: ***The Elastic, Content-Managed Warehouse.***

Warehouse Systems as a Content Management Problem

We've had a lot of time to think about this and our next software release***. Interestingly, we are NOT aiming to build a system to compete with the traditional WMS and WES vendors, but one to complement them.*** But, to achieve this, we'd have to build the discipline into our architecture to 1) know that we are co-existing with multiple WMS and WES systems, and 2) support a very flexible, repeatable deployment model. So, our solution was to build a multi-tenant, multi-site content-managed system.

It all sounds a bit like techno-mumbo-jumbo, but what it boils down to is a simple idea: separate the capabilities for integrating human resources, WMS and WES functions, robots, AGVs, etc. from the recipes needed to make them work. In doing so, we could create a flexible way for customers to use our services, to extend WMS and WES functionality, while not interfering with any vendor's core systems.

Obviously, in any warehouse system, there are a variety of artifacts like paperwork, label formats, reports that may differ from site to site. These represent things you can traditionally think of as content (in a true Content Management System). But beyond that, from a business and warehouse Ops perspective, there is so much more. While low-level componentry (APIs) may be the same, every site might want to use them slightly differently by combining them in different ways. In other words, they may want to tailor the flows to meet the exact requirements of their operation vs living with, say, a 75% fit. Subtle changes in screen flow, screen content, the sequencing of logic within business processes, can yield a dramatic impact on operational performance.

Thus, we have spent a lot of time trying to build an architecture that attempts to put all the controls in the hands of implementers, business analysts, and our operational savvy IT colleagues. Here are some examples:

User Interface (mobile App) Definition as Content

Everyone likes a good App. Now, with the new generation of Android devices making their way to the warehouse floor, we can put great Apps onto this new hardware. But one of the big problems with this is the fact that the Apps might require regular updates to get bug fixes or enhancements to the floor.

Managing Apps on 250 industrial mobile devices, which are used constantly 2-3 shifts per day, is a much bigger problem than worrying about downloading the newest version of *Uber Eats* to your phone every few months. It can be a difficult and time-consuming coordination problem. Our solution? Separate the App on the mobile device from content yielded by the App. In other words, the forms and flow the users see running within their App (the look of the forms and the flows) are content. In our case, we define them in XML, store them as resources, and fetch them at run-time to render the App. It can then be managed and distributed like content - and not EXE or APK files which require installation on every device. Here is the breakdown of how we did this.

Basic Screen Definition:

Basic screen configuration is stored in XML and rendered on-the-fly by the ElasticUI Form Engine. Note the tags for I18N (Internationalization) configuration. At the time of writing this document, the ElasticUI supports 13 of the most commonly used languages including multi-byte languages like Hindi. An example of a screen definition is below.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="3.0">
    <xsl:template name="tabForm">
        <form title="Tab Form" form-name="tabForm" featureGroup="elasticwms" feature="elasticwmsservice">
            <!-- Begin Tab Component, shortcut keys for corresponding components given in property with type "tab" -->
            <property name="testTab" type="tab" shortcutKeys="CTRL+a,CTRL+b,CTRL+c,CTRL+d">
                <property name="personal" type="object" i18nId="i18nTabHeader1" title="Personal Information" icon="information-circle">
                    <property name="firstName" i18nId="i18nText1" type="string" title="First Name" />
                    <property name="lastName" type="string" i18nId="i18nText2" title="Last Name"/>
                    <property name="age" type="number" i18nId="i18num1" title="Age"/>
                </property>
                <property name="education" type="object" i18nId="i18nTabHeader2" title="Education" icon="home">
                    <property name="degree" type="string" i18nId="i18nText3" title="Degree"/>
                    <property name="diploma" type="string" i18nId="i18nText4" title="Diploma"/>
                    <property name="years" type="number" i18nId="i18num2"  title="Years"/>
                </property>
                <property name="address" type="object" i18nId="i18nTabHeader3" title="Personal Address" icon="home">
                    <property name="city" type="string" i18nId="i18nText5" title="City"/>
                    <property name="country" type="enum" i18nId="i18nenum" vpType="Combo" lookUpCode="elasticwmsservice.elasticwms.i18nenum" title=
                    "City" enum="usa, in" enumTitles="United States, India" submit="false"/>
                    <xsl:if test="(global/country = 'usa')">
                        <property name="state" type="string" i18nId="i18nText6" title="State"/>
                    </xsl:if>
                </property>
                <property name="Course" type="object" i18nId="i18nTabHeader4" title="Course" icon="contacts">
                    <property name="Course" type="string" i18nId="i18nText7" title="Course"/>
                    <action name="submit" i18nId="i18nButton" title="Submit"/>
                </property>
            </property>
            <!-- End Tab Component -->
        </form>
    </xsl:template>
</xsl:stylesheet>
```

Example of Adding some Advanced Features:

To demonstrate some of the more advanced features of the ElasticUI, we teamed the UI framework with the smallest (easy to holster), least expensive, industrialized mobile device we could find (Zebra TC-20 – about $500/USD) for this example.

See the very short (one minute) video of our ElasticUI below, showing voice cues and a tightly-coupled integration with a unique wearable product called the ProGlove Display.

The behavior of both the ProGlove Display and the voice cues were added to the cycle count flow by a pair of simple tags in the form definition XML.

In the case of the ProGlove tags, they look like this* on the first from:

```
<property name="progloveAreaVerify" type="proglove" template="PG1"
enumTitles="Area Code" enum="SD99" i18nId="progloveAreaText" />
```

And this on the second form:

```
<property name="progloveLocationVerify" type="proglove"
template="PG2I" data="1||Location|2||B380050301" destroy="true"
i18nId="progloveLocationText" />
```

* For example only, in a projection application the DATA would be coming from the application dynamically

Similarly, the voice cues are configured in the XML and are Internationalized.

```
<property focus="true" name="locationConfirm" shortcutKey="F1"
type="string" audioLocale="en-US" title="(F1) Verify Count Location"
audioText="Enter or scan the location code shown on the sign at the
end of the aisle" matchValue="locationDisplay" required="true" />
```

The above example uses some of the new features released in the most current version of the AttunedLabs Leap FORM UI (Euphonium Release).

API/Services Definition as Content

The major focus of any modern system is to direct applications to the correct back-end components and/or services. In most systems, even with the newest technology stacks, these back-end services are pretty much hard-wired. This is the scenario with most WMS and WES systems - they think they *own* all the execution. Unfortunately, the notion of this level of *control* is an artifact of old-style thinking. As the overwhelming trend in today's warehouse systems shows a strong need for more collaboration between

automation technology vendors, we found ourselves in a situation where we needed to create an entirely new architectural approach.

The *Always Integrating* Warehouse Architecture

We looked at a lot of the work we'd done in the past 15-20 years – the common threads and difficulties, We asked some hard questions about why we spend so much labor doing integrations. Imagine what happens when you take WMS architects who wrote one of the original context processing engines (MOCA) and have them add some younger blood to re-imagine a new framework for this century.

You find yourself with a philosophy that we call the **Always Integrating Service**. In other words, we built integration into our services architecture as a core capability (and not a bolt-on or external software). We began with the approach by embedding Apache Camel, the light-weight routing and mediation engine, inside our architecture. In addition to wanting the ability to control flow as a content problem, which Camel does nicely, we wanted to build services that were integration aware or ***Always Integrating***. Camel comes with hundreds of components and connectors, allowing the inclusion of even the newest technologies (Kafka, crypto components, and dozens of Amazon AWS abilities). In the end, this new architecture removes the requirement for external ESB systems like Mulesoft to perform integrations (see the discussion of Big-Block integrations later in this paper).

Vendor Solutions as First-Class Services

One of the most important concepts of our approach to APIs is the acknowledgment that the landscape inside the warehouse is ever-changing. New solutions, new vendors, and new ideas are now the norm and not the exception. Thus, one of the major concepts within our framework is the idea of vendor-based implementations. In other words, we assume we are in a multi-vendor site and any particular service might be performed by one or more vendor solutions. Vendor solutions are not cobbled into fragile external integrations. Instead, they become first-class, fully enabled service providers. This is something we call ***Vendor-Selectable Services***.

Vendor-Selectable Service Example for Parcel

Integration with Parcel Systems is one of the best examples of needing vendor-selectable services. A customer may have a contract with UPS or FedEx, yet need to ship other parcels - maybe with a specialty carrier for white-glove deliveries. Historically, this would mean involving a Parcel Middleware System (PMS) at, not insignificant, extra expense. In our framework, it is possible to, within a single service definition specify the scenarios to use any number of different carrier APIs within one service definition. The application simply calls a single service. The system decides, dynamically, which vendor service to use, it mediates the difference in methods and formats and performs the correct service request. PMS vendors have great offerings, but using them to solve simple problems like the one cited above creates a lot of extra expense and overhead. Vendor-selected services provide customers with another choice.

Once again, ***Vendor-Selectable Services*** are all just content to us. Here is a snippet from a Good-To-Person robot picking service definition. Our architectural standard within service flows includes a *"Vendor Selection"* route. This route is responsible for looking at the necessary data to determine the vendor to perform, in this case, a pick. In this example, we are deciding to send data to either Locus Robotics or 6 River Systems robots.

```
<route id="roboticsservice-data-ImplementationSelectionRoute">

      <from uri="direct:roboticsservice-data-ImplementationSelectionRoute" />
      <toD uri="direct:$V{in.header.implroute}" />

</route>
```

Vendor Auditions

We discovered something interesting after we created this multi-vendor fabric inside our architecture. It enables multi-vendor *Auditions* or *Try-outs*. Take the idea of Work-Force Management (sometimes called Labor Management) software as an example. Our **Always Integrating** approach allows us to multi-cast transactions to these systems simultaneously. This idea allows customers to get to a whole new level of detail when considering new vendors by using very precise *Apples-to-Apples* comparisons. See the discussion of Entities, below, and how, using our new Entities framework, we quickly created mirrored integrations for Locus Robotics and 6 River Systems.

Entity Definitions as Content

Data acquisition, and some of the associated data actions, is core to any system. Many systems permit minor flow changes with table settings, configurations, and exit points. However, changing settings, policies, and configuration only gets you so far. Sometimes, you might just need to get access to some additional bit of data to avoid a lot of extra keystrokes on the floor. So, being able to create abstract entities, even entities exposed from other systems, and augment or reshape them can produce more desirable results in a system.

A new feature to the Leap Framework, called *Leap Entities*, provides a whole new set of capabilities in this regard. Leap Entities permit you to define an abstract, vendor aware, entity (data item or action) which has some defined attributes:

- ***Taxonomy***. This is the *lingua franca* the system with which you are communicating – a way to describe data (what data is called and how to reference it). It supports the definition of a base Taxonomy, say BY2019, and any number of foreign (vendor or system) taxonomies. BY2019 calls the pick identifier a wrkref while another system might call it PickID. Our XML-based taxonomy definitions provide a means of understanding the basic ways disparate systems reference similar data.
- ***Projection***. Where the taxonomy tells us how to describe data, the projection tells us how to form the data – the format of data payloads. The days of flat structures ended with the introduction of XML and have been further deprecated by the popularity of formats like JSON. Both payload formats permit convenient and ubiquitous ways of defining and communicating complex objects. The XML-Based projection definition provides a roadmap for the Leap Architecture to create complex object payloads on the fly.
- ***Transformation***. From an integration perspective, a big part of mediating the interchange of data between systems is handling the "*getting from here-to-there*" problem. In other words, how do you take on the actual transformation? With the taxonomy and the format of the payload defined (the projection) – the last step is the actual transformation. In most systems, these 3 elements (especially transformation) are external to the architecture. They would require an external system like Mulesoft or WebMethods to take care of this step. This is NOT the case in the Leap, ***Always Integrating,*** architecture. In this architecture, the transformation

step is just another configuration, performed in-line during service execution. It permits the definition of custom transformation adaptors or the use of XSLT. Thus, the necessity of an external integration tool or framework is largely eliminated.

Below is an example of the Entity configuration for a data service for interfacing with robots. The base taxonomy is BY2019 WMS, and the destinations are Locus Robotics and 6 River Systems. Note the highlighted differences in the configuration: ***TransformationConfiguration***, ***ProjectionFileName*** (both extracted from a Swagger file), and a ***TaxonomyFileName.***

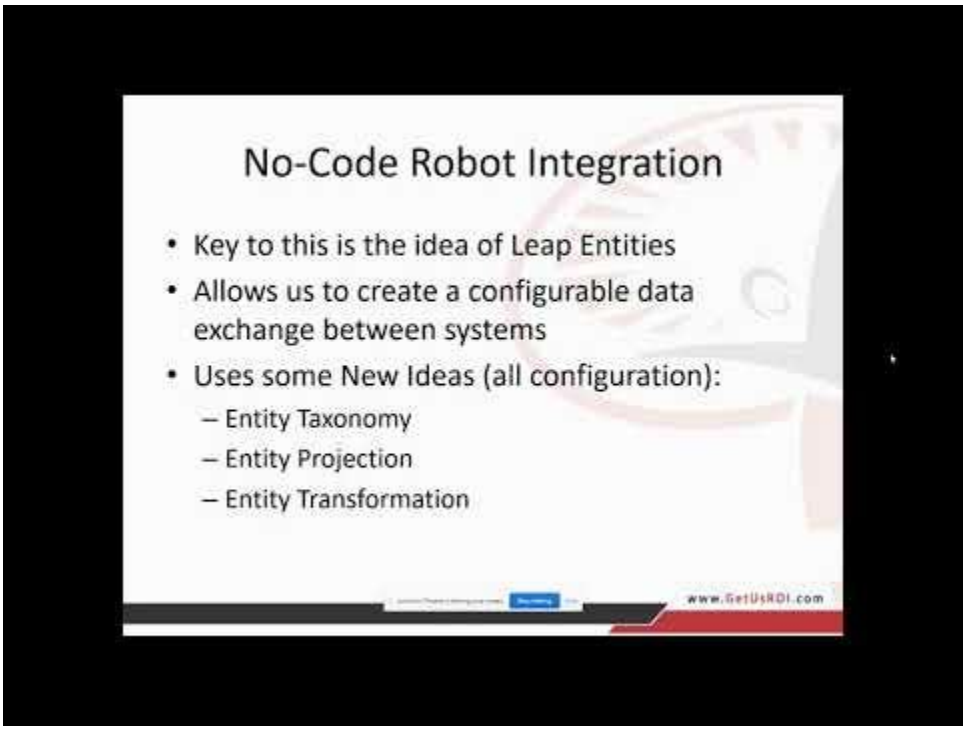
Locus Robotics Entity Configuration:

```
<EntityRestRequestBody source="LDC">

   <TransformationConfig required="true" fileName="Locus.xsl">
   </TransformationConfig>

      <LDCRequestConfigs>

          <ApplyLDCConfig>

              <LDCSchema required="false"
              schemaFileName="schemaFileName" /><LDCProjection required="true"
              projectionFileName="robotics_Locus_Swagger.json"
              projectionSource="swagger" />

              <LDCTaxonomy required="true" taxonomyFileName="Locus" />

          </ApplyLDCConfig>

       </LDCRequestConfigs>

   </EntityRestRequestBody>

</EntityRestRequest>
```

6 River Systems Robotics Entity Configuration:

```
<EntityRestRequestBody source="LDC">

   <TransformationConfig required="true" fileName="6RS_Pick.xsl">
   </TransformationConfig>

      <LDCRequestConfigs>

          <ApplyLDCConfig>

              <LDCSchema required="false"
              schemaFileName="schemaFileName" /><LDCProjection required="true"
              projectionFileName="robotics _riverSystems_Swagger.json"
              projectionSource="swagger" />

              <LDCTaxonomy required="true" taxonomyFileName="6RiverSystems" />

          </ApplyLDCConfig>

       </LDCRequestConfigs>

   </EntityRestRequestBody>
```

See a short view of doing a codelsss integration to these two vendors in the video below.



Super Entities

As we were developing our entity framework, we found that not all vendors had their APIs completely fleshed out, had implemented important variations with custom filters or ODATA queries, or were just missing important elements. To handle this, we create the idea of a Super Entity. This is a concept like the component wrapper ability in MOCA or the Superclass in object-oriented programming. We can abstractly redefine the Entity via configuration. This permits us to retrieve data from multiple services, or even multiple disparate systems and return the data in a single entity request. Once again, this uses the core capabilities of the Leap architecture to define, in XML content, a set of flows that are bound under one *Super* entity.

What of Big-Block Integration?

When you have an ***Always Integrating*** framework such as ours, much of what people normally call *integration work* is baked into services. Virtually, all peer-to-peer small-block integrations are covered by our ***Always Integrating* Services** framework.

Still, there are occasions when you still might need a big-block oriented integration (think order downloads or receipts downloads). For these use-cases, we use Apache Nifi. Nifi is an open-source, graphical integration configuration and monitoring tool. Configure it to read from files, to execute services, perform transformations. It provides the ability to monitor the flows at runtime to view errors and backpressure. These integrations are stored in XML and treated, again, like a Kind of content.