

Modernize Your Way to a Next Generation Technology Capability

Overview

It is obvious to many that the technology elastic has finally snapped at a lot of firms, and the innovation and efficiency gains of modern tech vs last-generation builds is just too great to ignore. The tech agility required to navigate a post-pandemic data-driven world means firms can no longer ignore the risks of continuing to put off the inevitable.

So, if now is the time to invest in your next-generation system, how should you go about it? And what are you trying to achieve?

Historically, the path forward would be a bold “rip and replace” style operation. Today, modernization offers a different approach with many benefits.

The modernization approach described in this paper, implies incremental improvement that maximizes the benefit of existing (legacy) systems rather than a big bang. Importantly, it can deliver very rapid outcomes, especially in terms of ability to innovate. It also allows for a more flexible cadence where other priorities can interrupt as needed. The cadence and order in which things get done is now at your discretion and can be tailored to fit in with changing business imperatives.

With a **vCore** based architecture that wraps-and-extends your existing systems, it not only empowers your business with a succession of quick wins but allows you to tackle a massive effort in chunks without requiring everything else on the to-do list to grind to a halt.

vCore Concepts

vCore is an application development platform that enables Java development teams to accelerate the build of high-performance, user-facing systems. It combines a proprietary streaming database and transactional engine, a custom web stack that can handle high performance loads with specialized “low/no code” accelerators that enable developers to maximize the time they spend building functionality.

Data Centric

vCore by default is a data centric architecture which is achieved by maintaining a highly available, highly resilient single data cache. Any system built on vCore will natively support collaboration and cross-system workflows as all data is read and writable in one shared location.

Schema Driven

vCore keeps the data model (schema) separate from the application so it can change as needed. The core API is code generated at build time. It's these material gains in improving your ability to change or shorten the innovation cycle that deliver sustained advantage.

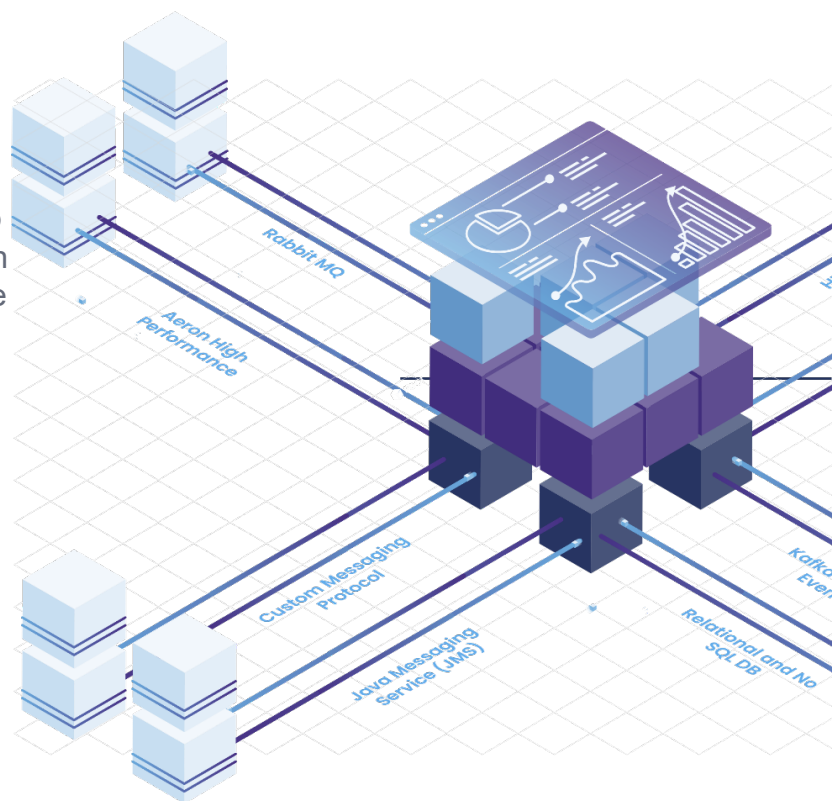
Event-based Programming

As a developer, being able to efficiently deal with the propagation of change events through a high-performance real-time platform is vital for both an efficient development experience and achieving a responsive user experience. vCore achieves this by building on many of the concepts found in Reactive Programming.

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change, enabling developers to write code that can react to state changes quickly. Asynchronous processing pipelines send data to a consumer as it becomes available. Events are captured asynchronously, by observer functions that execute when a value is emitted. The stream is the subject (or “observable”) being observed.

Benefits of the vCore Reactive Approach

- **Improves user experience** : its asynchronous nature and ability to control and handle backpressure mean that you will create a more responsive product for your users to interact with.
- **Faster development** : functions (blocks of code) can be added or removed from individual data streams, which means you can easily amend and reuse.
- **Easier threading** : provides a more intuitive and easier to use model for all types of threading issues.
- **Full-stack** : streams propagate change all the way to the UI.
- **Supportable** : streams are transparent and inspected/edited at runtime.



Cases

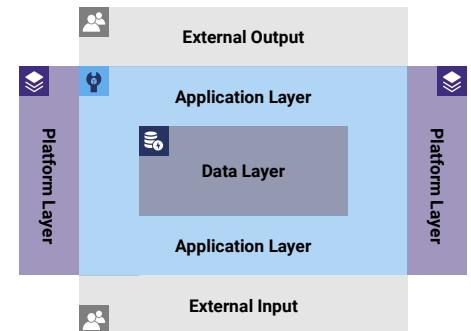
In this section we discuss 3 different business cases where a **vCore** modernization approach can be used.

1. System Consolidation {"Strangler Pattern"}
2. Consolidated Web Enabled User Interface
3. "Building on Top" with Data Centralization

vCore Flow Diagrams are a convenient way to explain conceptually how the underlying technology actually works. Flow Diagrams, break up the processing of datastreams into 3 logical layers.

Data Layer

A proprietary streaming database and transactional engine. For clarity we re-use SQL idioms for convenience when defining the data schema itself. The runtime API is code-generated and our "data-in-motion" paradigm, gives the developer up-to-date transactional and streaming data, when and where needed.



Application Layer

Specialized full-stack adaptations of "low/no code" accelerators. Low/No-Code tools do not work for high performance mission critical software. We have adapted some of the underlying techniques, such as model-driven development (MDD) and code generation to work in the high-performance space.

Platform Layer

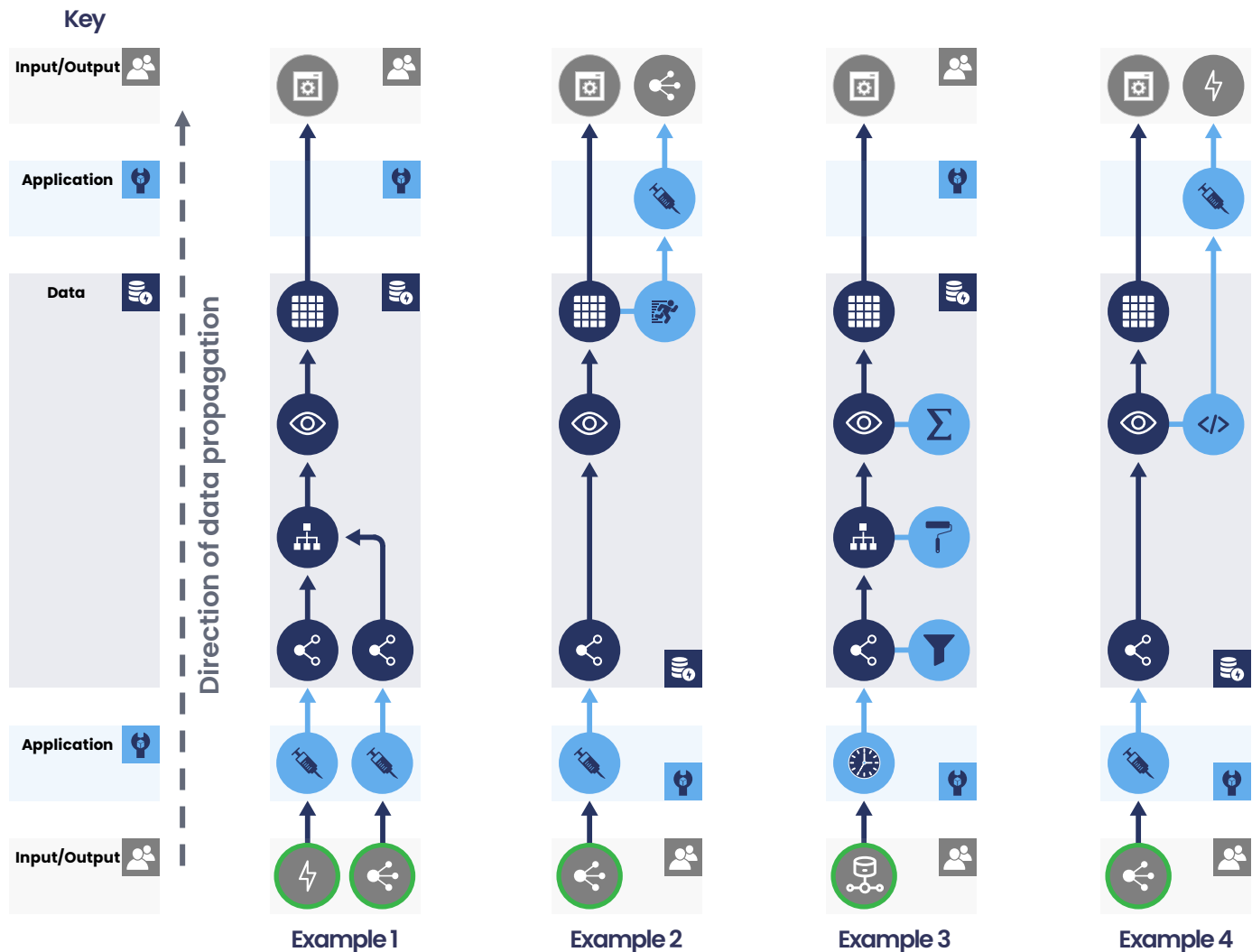
Contains many of the underlying non-functional components required to stand up a mission-critical system. In addition to a custom web stack that can handle high performance loads while reducing the amount of hand-crafted code required to stand-up enterprise UI's, includes support for fine grain entitlements and permissions, user configuration management and production support helpers.

Flow Diagram Examples

To further illustrate how Velox Flow Diagrams work, below are 4 examples describing common activities within a **vCore** implementation:

1. **Combining and viewing data streams** : Data from a request/reply system API and a real-time data feed are injected into **vCore**, combined and rendered into a web UI. *[READ ONLY]*
2. **User acting on a vCore screen** : A UI action is bound to some application code that when invoked writes to an external system API. *[READ & WRITE]*
3. **Transforming RDBMS tables**: The input data is filtered, additional columns computed on the fly and the data is then aggregated. *[READ ONLY]*
4. **Custom datastream processing**: As the input data changes, custom code is executed that writes to an external system API. *[READ & WRITE]*

Miscellaneous "Flow Diagram" Examples : Rendering External Datastreams in a vCore UI



Key:

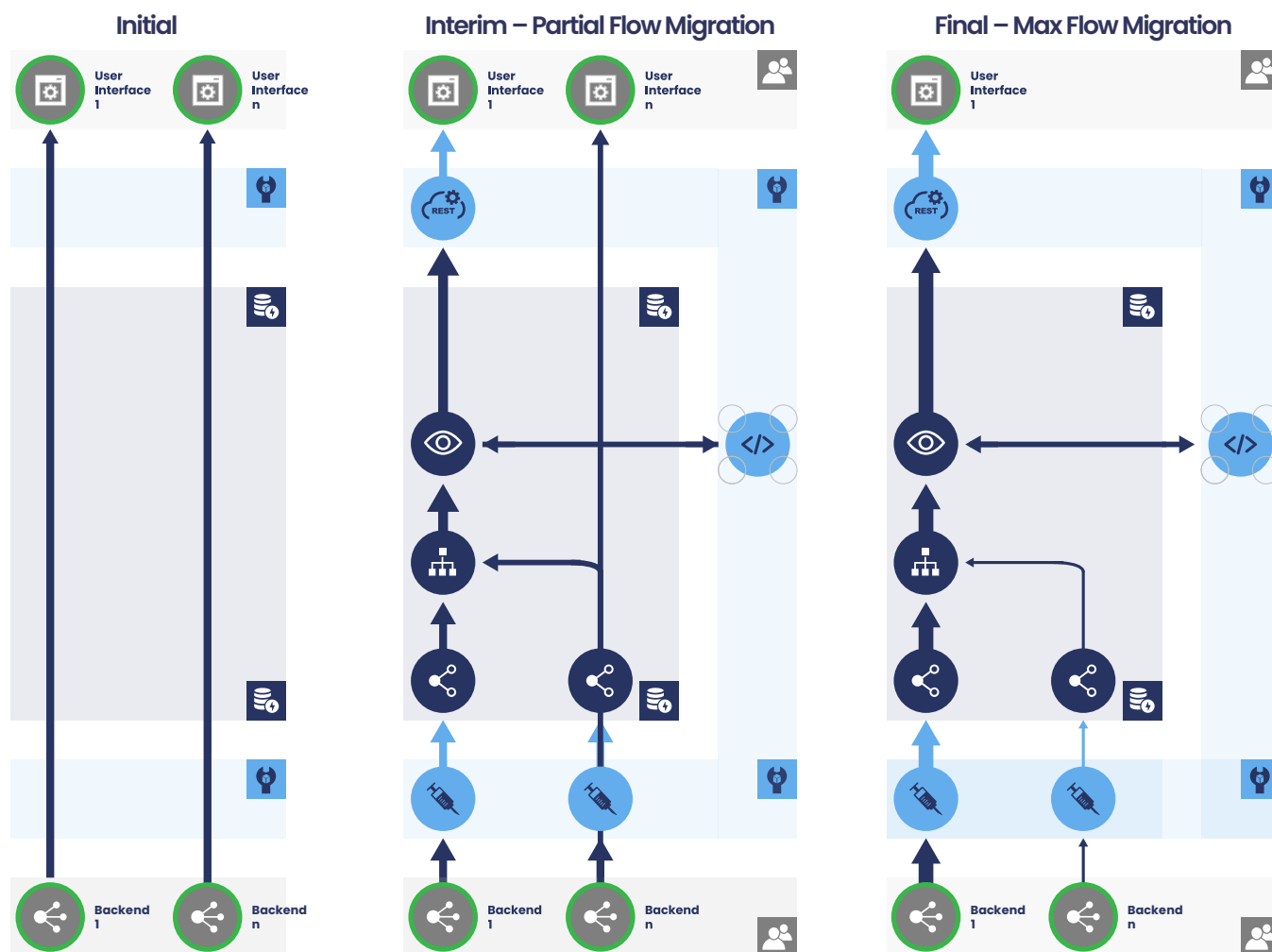
- User Interface**: Desktop web browser or local install application.
- Real-time read/write API**: Usually, an external system.
- Database**: Typical RDBS or other well known database type.
- Data Stream**: Usually, a high velocity market data feed
- Screen**: A combination of tables, values and actions rendered within a browser SPA.
- View**: A rationalized representation that stands up information efficiently for readers.
- Joins**: Streamlined "JOIN" logic, abstracts away complexity of synchronizing sources.
- Tables**: Logic can be written without explicit synchronization with the subscribers.
- Aggregation**: Column specific methods to be used to calc rollups after a pivot action.
- Decorator**: On-the-fly calc of virtual tables and columns for analytics and meta data
- Filter**: A rationalized representation that stands up information efficiently for readers
- Polling**: Timer based pull of a request / reply interface to a database.
- Injector**: Split the computational load of a join or aggregation across many cpu's/jvm's
- Code**: Application code (including Main) that executes business logic
- Action**: Act on a resulting data set using bound source system API
- External component**: Either a non-vCore front-end or non vCore backend

System Consolidation {"Strangler Pattern"}

It is not unusual to be running multiple systems that have a considerable amount of functional overlap. At some point it's likely that these systems will need to be collapsed, either to simplify the user workflow, reduce operational costs or to simply reduce overall complexity.

A significant amount of benefit can be gained from being able to tackle the user facing component independently from the backends. This has the potential benefit of realizing the business gains quickly without making them dependent on complex backend migrations.

This examples shows **vCore** being used as a general-purpose abstraction layer that connects to **n** backends. Application code is written to augment any differences on backend operations to comply with the requirements of the single UI. Over time activity in the legacy backends is reduced.



Key:

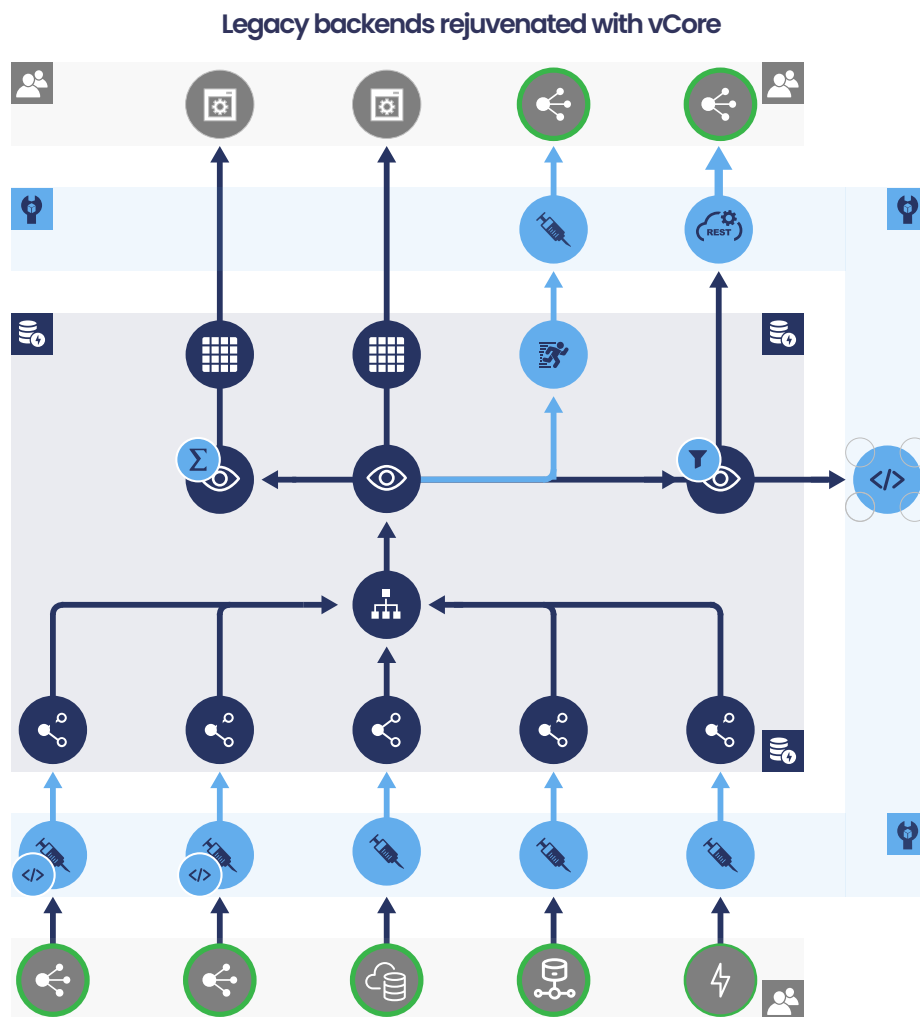
- User Interface:** Desktop web browser or local install application.
- Real-time read/write API:** Usually, an external system.
- Database:** Typical RDBS or other well known database type.
- Data Stream:** Usually, a high velocity market data feed
- Screen:** A combination of tables, values and actions rendered within a browser SPA.
- View:** A rationalized representation that stands up information efficiently for readers.
- Joins:** Streamlined "JOIN" logic, abstracts away complexity of synchronizing sources.
- Tables:** Logic can be written without explicit synchronization with the subscribers.
- Aggregation:** Column specific methods to be used to calc rollups after a pivot action.
- Decorator:** On-the-fly calc of virtual tables and columns for analytics and meta data
- Filter:** A rationalized representation that stands up information efficiently for readers
- Polling:** Timer based pull of a request / reply interface to a database.
- Injector:** Split the computational load of a join or aggregation across many cpu's/jvm's
- Code:** Application code (including Main) that executes business logic
- Action:** Act on a resulting data set using bound source system API
- External component:** Either a non-vCore front-end or non vCore backend

Consolidated Web Enabled User Interface

Whether it be to leverage the deployment advantages of a web browser UI or to replace an old UI with one that is more functionally aligned to current business goals, technology teams can often find themselves needing to replace the front-end but not the back.

In this example **vCore** connects into *n* backends, in some cases. special processing is required in the injector to manipulate the data from the external system into a single common data model.

In addition to a new web-based UI built in **vCore**, this example shows **vCore** writing back to the backend sources either programmatically or invoked by user action on the UI.



Key:

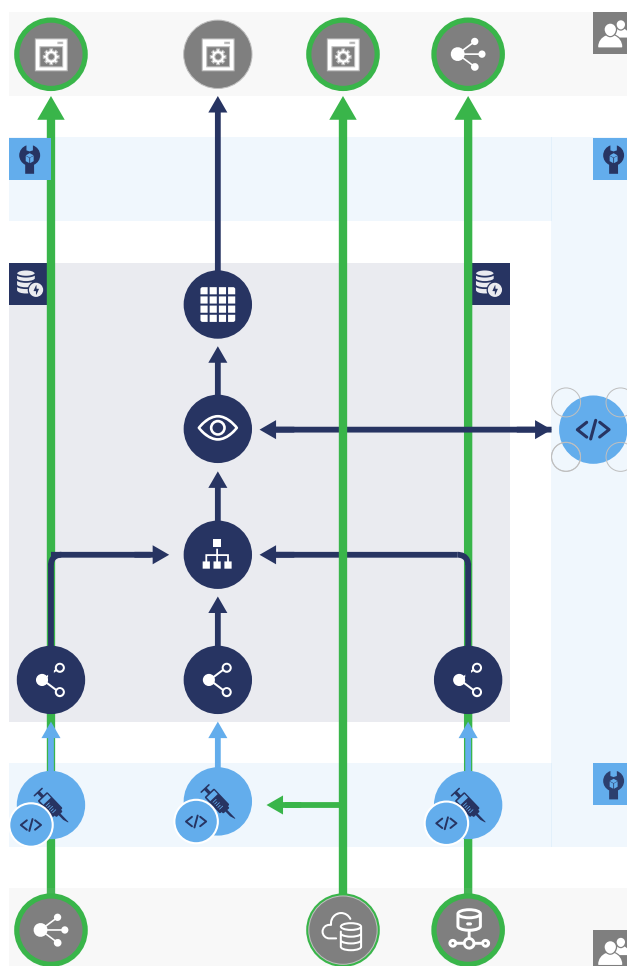
- User Interface**: Desktop web browser or local install application.
- Real-time read/write API**: Usually, an external system.
- Database**: Typical RDBS or other well known database type.
- Data Stream**: Usually, a high velocity market data feed
- Screen**: A combination of tables, values and actions rendered within a browser SPA.
- View**: A rationalized representation that stands up information efficiently for readers.
- Joins**: Streamlined "JOIN" logic, abstracts away complexity of synchronizing sources.
- Tables**: Logic can be written without explicit synchronization with the subscribers.
- Aggregation**: Column specific methods to be used to calc rollups after a pivot action.
- Decorator**: On-the-fly calc of virtual tables and columns for analytics and meta data
- Filter**: A rationalized representation that stands up information efficiently for readers
- Polling**: Timer based pull of a request / reply interface to a database.
- Injector**: Split the computational load of a join or aggregation across many cpu's/jvm's
- Code**: Application code (including Main) that executes business logic
- Action**: Act on a resulting data set using bound source system API
- External component**: Either a non-vCore front-end or non vCore backend

“Building on Top” with Data Centralization

















This example takes aspects of the previous two examples to create a completely new user-facing system that sits along side the existing application estate. This is particularly useful when the legacy systems are very rich in functionality but only occasionally used.

The new **vCore** system covers and modernizes the 80% of functionality that is constantly in use and takes advantage of all the data and backend services being in one place to make it possible to implement more powerful workflows and functionality.

vCore Enriching and extending the life of legacy systems



Key:

-  **User Interface**: Desktop web browser or local install application.
-  **Real-time read/write API**: Usually, an external system.
-  **Database**: Typical RDBS or other well known database type.
-  **Data Stream**: Usually, a high velocity market data feed
-  **Screen**: A combination of tables, values and actions rendered within a browser SPA.
-  **View**: A rationalized representation that stands up information efficiently for readers.
-  **Joins**: Streamlined "JOIN" logic, abstracts away complexity of synchronizing sources.
-  **Tables**: Logic can be written without explicit synchronization with the subscribers.
-  **Aggregation**: Column specific methods to be used to calc rollups after a pivot action.
-  **Decorator**: On-the-fly calc of virtual tables and columns for analytics and meta data
-  **Filter**: A rationalized representation that stands up information efficiently for readers
-  **Polling**: Timer based pull of a request / reply interface to a database.
-  **Injector**: Split the computational load of a join or aggregation across many cpu's/jvm's
-  **Code**: Application code (including Main) that executes business logic
-  **Action**: Act on a resulting data set using bound source system API
-  **External component**: Either a non-vCore front-end or non vCore backend

Performance Characteristics

vCore achieves high-performance user-facing systems by taking a user-centric approach (top-down design rather than bottom-up).

This is because in a system that needs to operate over massive fast-moving datasets, with humans-in-the-loop, the physical limit of how much information a human can consume vastly restricts how much data needs to be calculated and transported from server to client. The architecture makes use of this limitation through various techniques:

- Lazy calculations of derived columns that are not visible and are not part of the dependency tree of a data element that is visible.
- Data virtualization so only data elements in the viewport need to go on-the-wire.
- Conflation and throttling to an update rate that is humanly digestible.
- Statically defined high-load streaming table operations like aggregations and filters.
- Rapid app dev framework to massively shorten the innovation cycle when new aggregations/filters are identified and need to be statically defined.

On this basis, vCore can be used in a wide variety of scenarios. For example, we have run a vCore system with ~100MM rows, ~400 columns and sustained update rate of ~500k-1MM rows per second. These numbers are not upper limits, and the absolute limits depend on many factors.

100 MM

Number of
rows in a Velox
table

10 k

Number of
updates per
seconds

100 k

Number of
concurrent
browser sessions

5 ms.

Time to pivot
a grid containing
1M rows

1

Full-stack front-
office
development
platform

5 yrs.

The amount of
time the core
code has been in
production.

> 250

Data sources and
API's
simultaneously
connected.

> 100

Grids within a
single user
session

Building Applications

Here are a few fundamental examples to give a flavor of what the **vCore** developer experience is like. Starting with the Data Dictionary, which is probably the most important single concept in the platform. The Data Dictionary is how the data domain is separated from the application itself. It is effectively the “model” that drives the API code generation process. When using **vCore** to sit on top of multiple legacy datasources/API's, developers use this opportunity to migrate to one single clean data model.

Table

Most objects are modeled as either a Table or a Join. A Join is just a nested table with some platform added "joining" capability

```
<table name="TableName">
  <field name="fieldName" primaryKey="true"/>
</table>
```

Join

Equivalent of a relational database left outer join. Unlimited number of tables can be added to the Join and this can be streaming / non-streaming etc. The Join operator ensures sources remain synchronized.

```
<join name="JoinName" primary="JoinTableA">
  <field type="joinTableB" root="joinFieldName"/>
</join>
```

Decorated Join

Derived real-time fields can be added via a mechanism which enriches streaming joins with calculated fields. These can be defined in the dictionary and refer to pieces of Java code, the Join mechanism ensures that these get updated in sync

```
<join name="JoinName" primary="TableName">
  <field name="decoratedField" type="Analytics"
    prePublishDecorator="Analytics::calcAnalytics"/>
</join>
```

View

A projection of the fields from an underlying table which allows: rename of fields; expose a subset of the fields; create a new derived column; set the caption and format; specify aggregation function and the way a field should be rendered.

```
<view name="ViewName" table="TableName">
    <column name="fieldName" aggregation="SUM" caption="Field Name"/>
</view>
```

Aggregated View

Aka. pivots or group-by or roll-ups. If you want to pivot a Table either programmatically, or if the user does it from the UI, then each column in the table has to know how to aggregate itself. In the dictionary you can specify whether to use a canned aggregation type (mathematical operator) or refer to a custom type that defined in Java code. If the out-of-the-box aggregations (min/max/sum/avg etc) are not enough, you can define your own and then make them available to all in the Dictionary. Aggregations tell the cache what to do with a column when it is rolled-up as part of a user-defined pivot.

```
<view name="OrderJoin" table="ETOrderJoin"
    resultTable="ETOrderJoinAgg">
    <column name="side" source="order.side" aggregation="customSet"
        <aggregationParameter converter="Aggregate::aggregatedSide"/>
    </column>
</view>

public static Aggregate aggregatedSide (Set<Side> sides){}
```

Screen Table

Represents the view model of the screen. There are three types of control: singlevalue (i.e text,label), actions (i.e dropdown, checkbox) and grids which are the most powerful and feature rich control type on the platform. Highly complex forms can be constructed with combinations of these primitive types.

```
<screen name="ScreenName">
    <control name="gridCntrolName" type="datagrid" datatype="TableName"
        keytype="Object" viewname="ViewName"/>
    <control name="actionControlName" type="action"/>
</screen>
```

State Table

Holds the state of the business object. The structure of a state looks the same as a table defined in the schema. In fact, a state can be used anywhere where a table can be used. If we consider the functionalities, a state is basically a superset of a table.

```
<state name="StateTableName">
    <field name="fieldName" primaryKey="true"/>
</state >
```

Command

Similar to a stored procedure. Allows you to define actions that can change one or more state in an ACID-compliant manner.

```
<command name="CommandName">
  <field name="fieldName"/>
  <result name="resultName" type="ResultType"/>
</command>
```

Screen Layouts

Along with the dictionary the developer needs to provide HTML5 layout files which provide coarse-grain placement instructions and style sheets (CSS). HTML5 layout files provide coarse-grain placement instructions for visual components. We also use vue.js to interoperate with other web components like react and angular.

```
<div class="ScreenName screen vue">
  <div class="button-group"> // dynamic buttons
    <button class="btn fieldName flex-1" is="vue-action" v-model=modelName">
      <span style="font-size: 14px">Last</span>
      <span class="last">{{fieldName.Value}}</span>
    </button>
  </div>
</div>
```

Screen Config

HTML5 layout files provide coarse-grain placement instructions for visual components. We also use vue.js to interoperate with other web components like react and angular.

```
"visibleColumns": [{"columnName": "column1"}],
"columnSettings": {"column1": {"displayName": "Column 1", "width": 80,
"columnName": "column1"}},
"sorts": [{"columnName": "column1", "direction": "DESCENDING"}],
"filters": [{"filterExpression": "( > field1 'value1' )"}]
```

Data Adapters

Translate and synchronize between external data sources and API's. You can connect to any external source/system that has a Java API. A simple field mapping exercise needs to be performed. Allows you do any pre-processing and have complete control of the data access methodology

```
// connect to the external data source
var consumer = new KafkaConsumer(kafka top); consumer.subscribe(topics);

// get a reference to the Velox table
CachePublisher<marketData, ?> marketDataPub =
    m_dc.getPublisher(marketData.class);

// for each record consumed
while (true) {

    // for each field on the record write into the Velox tabl
    for (JsonNode MarketDataJsonElement: MarketDataJson) {
        Builder builder = Builder.newBuilder();
        Builder.field(getString(JsonElement,"fieldName",null));
    }
}
```

Screen Handlers

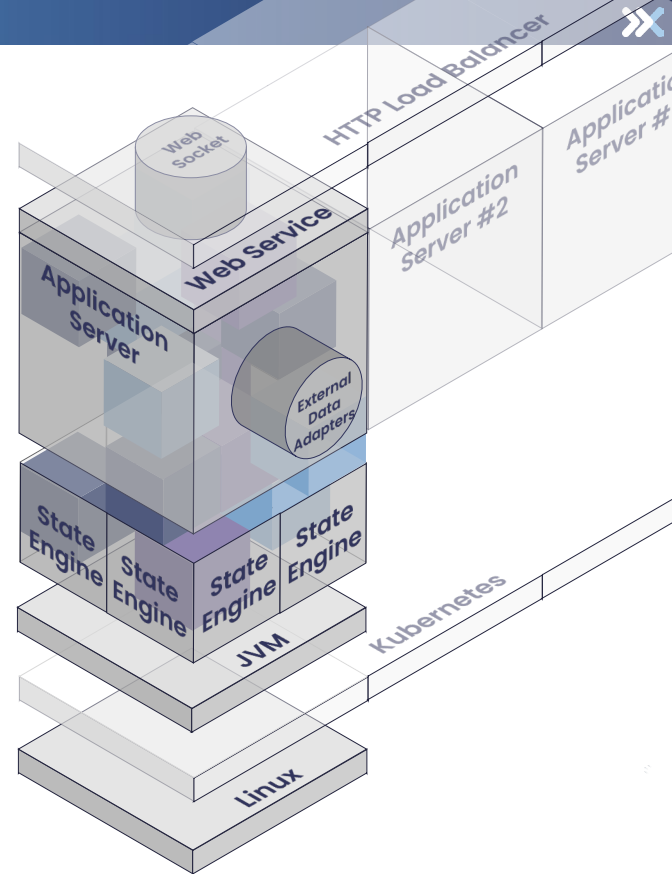
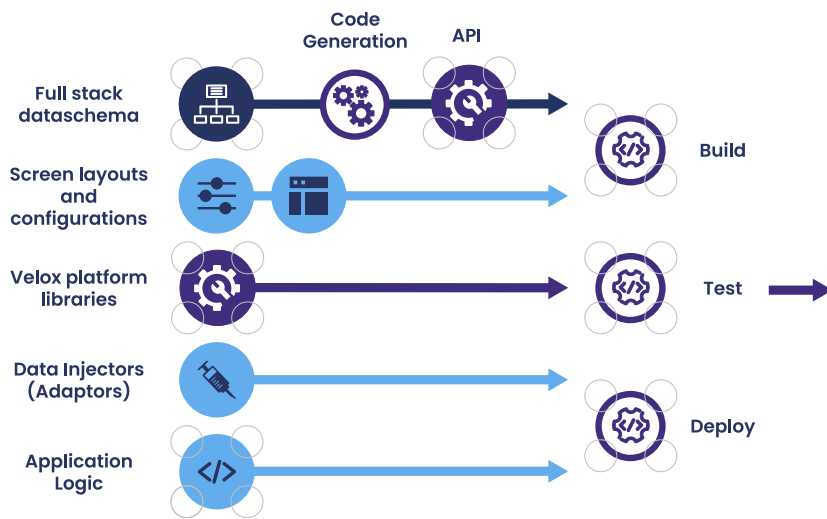
For every screen a screenhandler needs to be provided. Each screen provider typically has a handle on the global application object where all global data services are accessible from so it's convenient to be able to get access to it when writing business logic. If the screen can be accessed directly from the launch page, it should be extended from BaseScreenProvider as the launch page needs to access basic information such as icon, caption and grouping information for this screen.

```
public class ThisScreenProvider extends BaseScreenProvider<> {

    public void create(SessionState state, ClientNotifier clientNotifier) {
        // DataContextAccessor is the interface used by the app access to app data
        // access all global tables defined in the schema using DataContextAccessor
        // sessionState contains the user session information
        DataContextAccessor dc = state.getDataContextAccessor();
        // create the screen. the params are ThreadingContext and grid data sources

        final ThisScreen screen = new ThisScreen(state, tablePublisher.getTable());}
}
```

Build Process



Type	Artefact type	Artefact description
XML	Full Stack Schema	Define the data entities (tables) and their relationships, from data sources to output screen.
HTML/ XML	Screen Layout & Config	Coarse-grain layout instructions in HTML5 and CSS. Workspaces, grids.
Java	Data Injectors	Maps data elements from an external API to Velox table elements; apply pre-processing.
Java	Application Logic	Business logic, analytics, aggregations, screenhandlers, commandHandlers etc.

Physical Architecture

Two discrete service types are deployed as standard binaries. Most applications will run on a single xl AWS ECS host .

Appserver

Handles user actions (from the web UI) and all other application logic & processing. High-availability and scalability is supported by running sufficient, identical, appservers behind an HTTP load balancer.

The workload on an appserver can vary greatly depending on the usage profile of a particular user. It will be up to the developer to work out the most efficient scheme, but generally a smaller than normal number of high-powered users will be allocated to a single app server.

Can also be horizontally scaled (by instrument groups or some other classification) in which case a client request would be handled based on group inclusion.

State Engine

Used to handle all the transaction processing that needs to be ACID compliant and for persistent storage when data integrity needs to be maintained across system restarts. The state engine is generally used in situations where vCore is the master of a particular data set (as in the case of an OMS). High-availability and scalability are supported by multiple hot-hot nodes running in a resilient fault-tolerant cluster.

Major International Investment Bank – Equities Technology

August 2019

Without a multi-purpose development platform to build solutions, the team are constantly re-inventing the wheel, building and maturing core application scaffolding, like wrangling disconnected data sources, standing up scalable, supportable, testable backends and finessing visual components.

Out-of-the-box, **vCore** provides the common components and non-functional attributes that all mission critical applications need, but without the limitations that can arise from other development accelerators when your problem moves beyond their intended use-cases.

A vCore USP is focusing on the needs of the professional developer. Developers want to work with the business framework as they see it as enhancing their skills and output by increasing the amount of time spent on differentiating business features. While achieving this without being forcing them into an environment that's unfamiliar and does not leverage their years of software engineering experience.

By centralizing to one core application container and bringing consistency to how and where business logic is developed and operated, the benefits can grow over time as more and more code is reused between teams.

Project Overview

With only 2 developers, significant progress has been made in terms of business delivery and legacy system elimination.

Highlights

- Many applications are now managed and operated in one place, reducing tech support and maintenance costs.
- Apps are all accessible by the user from one place.
- Users can build different workspaces to fit their needs.
- With the underlying data and API's centralized, it enables higher-level workflows to be built, spanning multiple silos.
- Functionality is logically decomposed into services making it easier to modify in the future.

About Velox

Velox enables software development teams to build high-performance user-facing systems up to 10x faster.

The Velox full-stack application development platform (vCore) provides professional developers with tools that amplify their expertise in Java and Web programming, allowing them to focus on building differentiating business functionality.

Founded in 2018 by 3 veterans of front-office technology, vCore is the catalyst broker dealers, investment banks, exchanges and data and tech vendors need to accelerate their digital transformation and modernization journey.