ScaleOut Digital Twin Builder™
User Guide

Version 0.5-Beta for ScaleOut StreamServer™ V5.8

Introduction

The ScaleOut Digital Twin Builder™ is a software toolkit for creating stateful stream-processing applications using the digital twin model. The digital twin model lets the application define state information and an event-processing algorithm for each *type* of data source (for example, a wind turbine), and it co-locates both dynamic state information and event messages for each *instance* of a data source (for example, windturbine 17). It leverages well understood object-oriented techniques to simplify application design. This model is well suited for building stream-processing applications in many fields, including industrial IoT, manufacturing, logistics, financial services, medical monitoring, ecommerce, security, and more.

This revolutionary approach to organizing application code and data offers several key advantages over traditional, pipelined software platforms for stream-processing:

- automatic tracking of dynamic state information associated with data sources that generate event messages, enabling deeper introspection
- automatic correlation of incoming event messages for each data source
- a convenient, object-oriented basis for encapsulating event processing algorithms for each type of data source
- a predefined domain for creating data-parallel analyses to detect aggregate trends across groups of data sources in real time
- performance advantages, including reduced network overhead to access state information and automatic throughput scaling by the underlying compute engine

The ScaleOut Digital Twin Builder defines APIs in Java and C# (with JavaScript support to be released soon) that let application developers easily:

- define digital twin models for several types of data sources
- deploy those models for event processing within ScaleOut StreamServer's in-memory data grid
- connect the execution environment to three types of messaging frameworks that connect to data sources: Kafka message brokers, Microsoft Azure IoT Hubs, and ScaleOut's REST web server
- process event messages using digital twin instances dynamically created by ScaleOut StreamServer for each data source
- send event messages to digital twins from other twins, enabling the construction of a logical hierarchy among the digital twin models
- send command messages to data sources from digital twins

This User Guide first describes the digital twin model in more detail. It then describes the software architecture for building and running digital twins within ScaleOut StreamServer and gives a tour of the Java and C# APIs in the toolkit along with several examples.

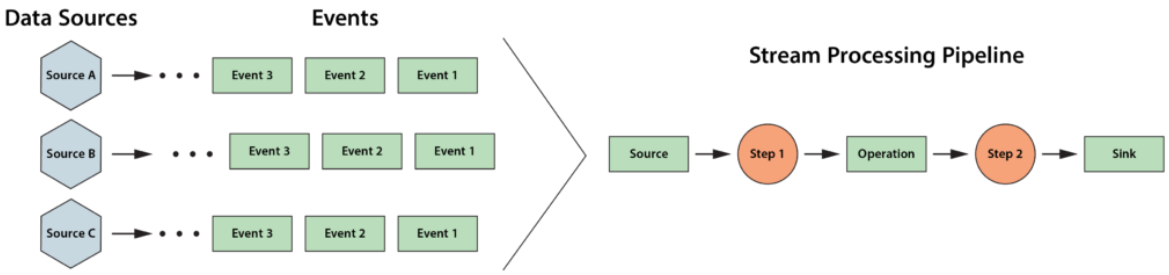
The Digital Twin Model for Stateful Stream-Processing

This section provides an overview of the digital twin model for stateful stream-processing and explains how it uses object-oriented design principles to simplify the design of large applications.

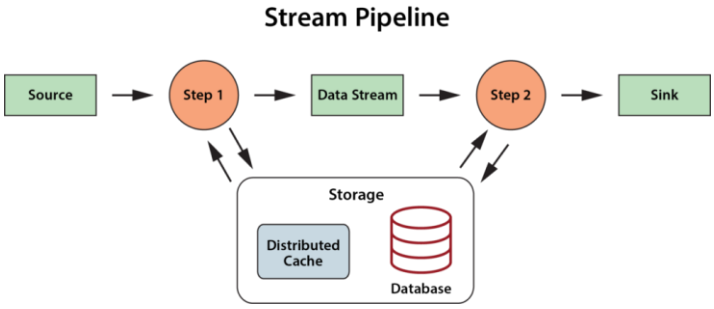
What is the Digital Twin Model?

Traditional stream-processing and complex event processing systems have focused on extracting interesting patterns from incoming data with *stateless* applications. While these applications maintain state information about the data stream itself, they don't generally make use of information about the data sources or their context. For example, if an IoT application is attempting to detect whether data from a temperature sensor is predicting the failure of the medical freezer to which it is attached, it looks at patterns in the temperature changes, such as sudden spikes or a continuously upward trend, without regard to the freezer's usage or service history.

The following diagram depicts a typical stream processing pipeline processing events from many data sources:

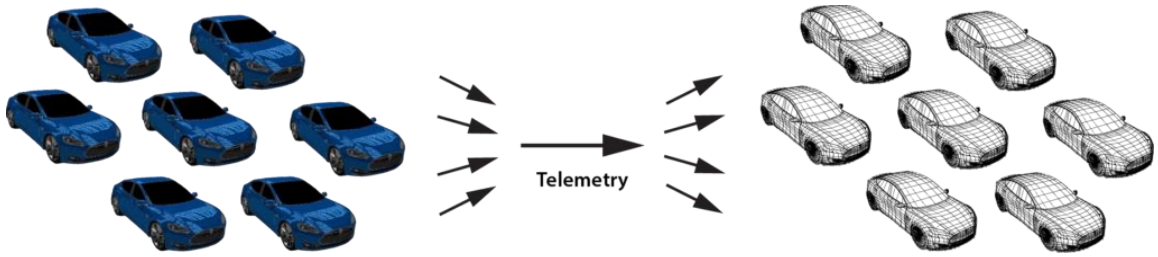


More recent stream-processing platforms, such as Apache Flink, have incorporated stateful stream-processing into their architectures in the form of key-value stores or databases that the application can make use of to enhance its analysis. But they do not offer a specific semantic model which applications can leverage to organize and track useful state information and thereby deepen their ability to analyze data streams.



The digital twin model offers an answer to this challenge. While this term was coined by Dr. Michael Grieves (U. Michigan) in 2002 for use in product life cycle management, it was recently popularized for IoT by Gartner in a 2017 report. This model offers key insights into how state data can be organized within stream-processing applications for maximum effectiveness. In particular, it suggests that

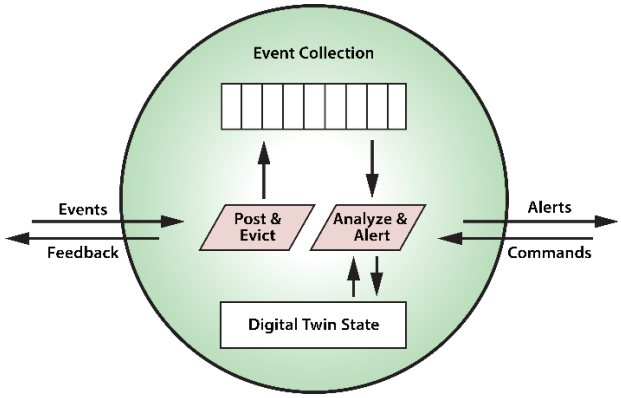
applications implement a stateful model of the physical data sources that generate event streams, and that the application maintain separate state information for each data source. For example, using the digital twin model, a rental car company can track and analyze telemetry from each car in its fleet with digital twins:



The digital twin model thereby provides an intuitive approach to organizing state data, and, by shifting the focus of analysis from the event stream to the data sources, it potentially enables much deeper introspection than previously possible. With the digital twin model, an application can conveniently track all relevant information about the evolving state of physical data sources. It can then analyze incoming events in this rich context to provide high quality insights, alerting, and feedback. For example, digital twins of medical freezers could track detailed facts about the specific model, its service history, environmental conditions, and usage patterns for each physical unit to help analyze telemetry from a temperature sensor and make more informed predictions about possible impending failures.

Beyond providing a powerful semantic model for stateful stream processing, digital twins also offer advantages for software engineering because they can take advantage of well understood object-oriented programming techniques. A digital twin can be implemented as a data class which encapsulates both state data (including a time-ordered event collection) and methods for updating and analyzing that data. Analytics methods can range from simple sequential code to machine learning algorithms or rules engines. These methods also can reach out to databases to access and update historical data sets.

As illustrated in the following diagram, a digital twin can receive event messages from data sources (or other digital twins). It also can receive command messages from other digital twins or applications. In turn, it can generate alert messages to applications and feedback messages (including commands) to data sources:



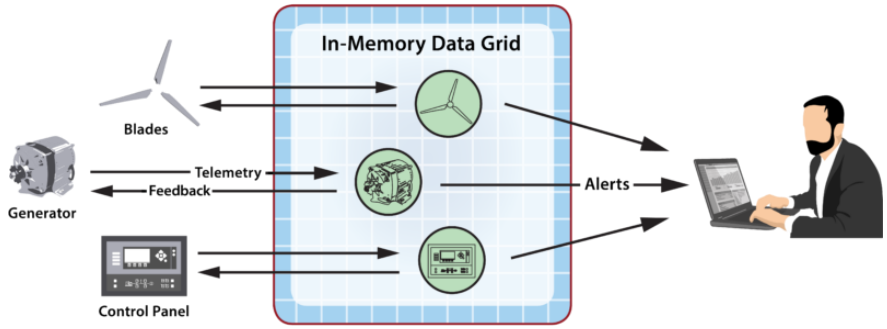
For each physical data source, an instance of a digital twin model is created by the stream-processing system to receive and analyze events. It is the responsibility of the system to correlate data from a given data source for delivery to each instance of a physical twin. In many applications, a stream-processing system may host thousands (or more) digital twins to handle the workload from its data sources.

The granularity of a digital twin can encompass a model of a single sensor or that of a subsystem comprising multiple sensors. The application developer makes choices about which data (and event streams) are logically related and need to be encapsulated in a single entity for analysis to meet the application's goals.

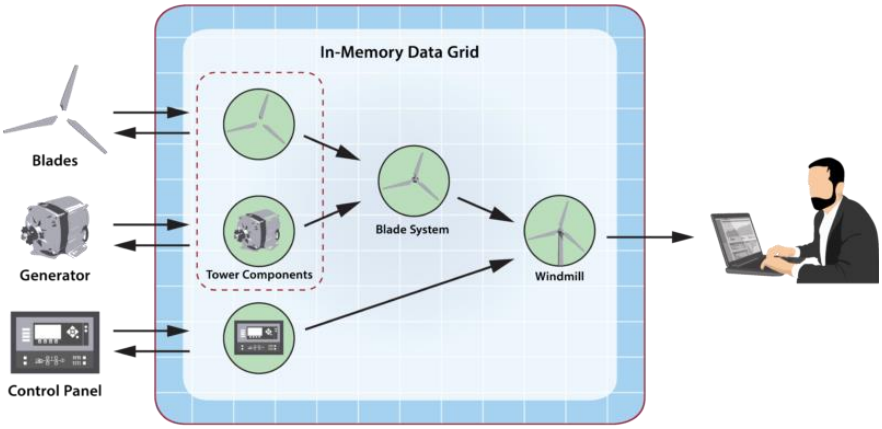
In summary, the digital twin model provides a powerful organizational tool that focuses on the state of data sources instead of just the data within event streams. With this additional context, it magnifies the developer's ability to implement deep introspection and represents a new way of thinking about stateful stream-processing and real-time streaming analytics.

Building a Hierarchy of Digital Twins

To implement stream-processing applications for complex systems, digital twin models can be organized in a hierarchy at multiple levels of abstraction, from device handling to strategic analysis and control. Consider an application that analyzes telemetry from the components of a windmill. This application can receive telemetry for each component and combine this with relevant contextual data, such as the component's make, model, and service history, to enhance its ability to predict impending failures. The following diagram illustrates how the digital twin model correlates telemetry from three components of a hypothetical windmill (blades, generator, and control panel) and delivers it to associated digital twin objects within an IMDG, where event handlers analyze the telemetry and generate feedback and alerts:



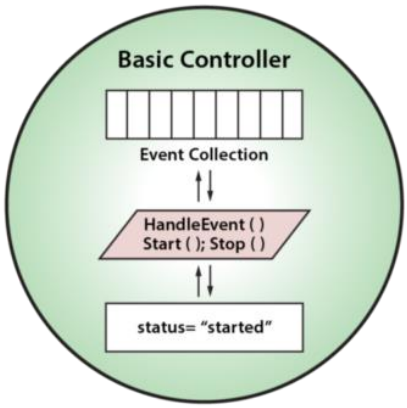
In this example, the blades and generator work together to generate power managed by the control panel. Taking advantage of a hierarchical organization as shown below, the digital twins for the blades and generator could feed telemetry to a higher-level digital twin model called the Blade System that manages the rotating components within the tower and their common concerns, such as avoiding over-speeds, while not dealing with the detailed issues of directly managing these two components. Likewise, the digital twin for the blade system and the control panel feed telemetry to a yet higher-level digital twin model which coordinates the overall windmill's operation and generates alerts as necessary. This hierarchy of components could be interconnected as follows:



By partitioning the application into a hierarchy of digital twins, the code can be modularized and thereby simplified with a clean separation of concerns and well-defined interfaces for testing.

Using Object-Oriented Design Principles

The use of object-oriented techniques can simplify the design of digital twins, reduce overall development time, and enhance maintainability. Because a digital twin encapsulates state information and associated analysis code, it naturally can be represented as a user-defined data type (often called a *class*) within an object-oriented language, such as Java or C#. The use of an object class to represent the controller conveniently encapsulates the data and code as a single unit and allows an application to create many instances of this type to manage different data sources. For example, consider the digital twin for a basic controller with class properties (status and event collection) describing the controller's status and class methods for analyzing events and performing device commands. This class can be depicted graphically as follows:

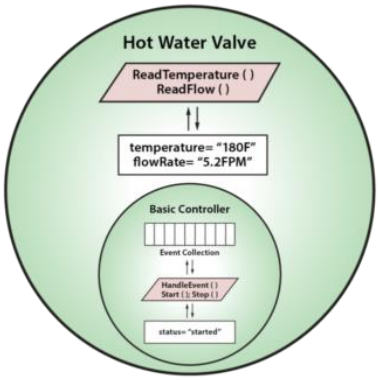


Here's how a basic controller class could be written in Java:

```
public class BasicController {
    private List<Event> eventCollection;
    private DeviceStatus status;

    public void start() {…}
    public void stop() {…}
```

```
    public void handleEvent() {…}
}
```

An application also can make use of the class definition to construct various special purpose digital twins as *sub*-classes, taking advantage of the object-oriented technique called *inheritance*. For example, we can define the digital twin for a hot water valve as a subclass of a basic controller that adds new properties, such as temperature and flow rate, with associated methods for managing them:



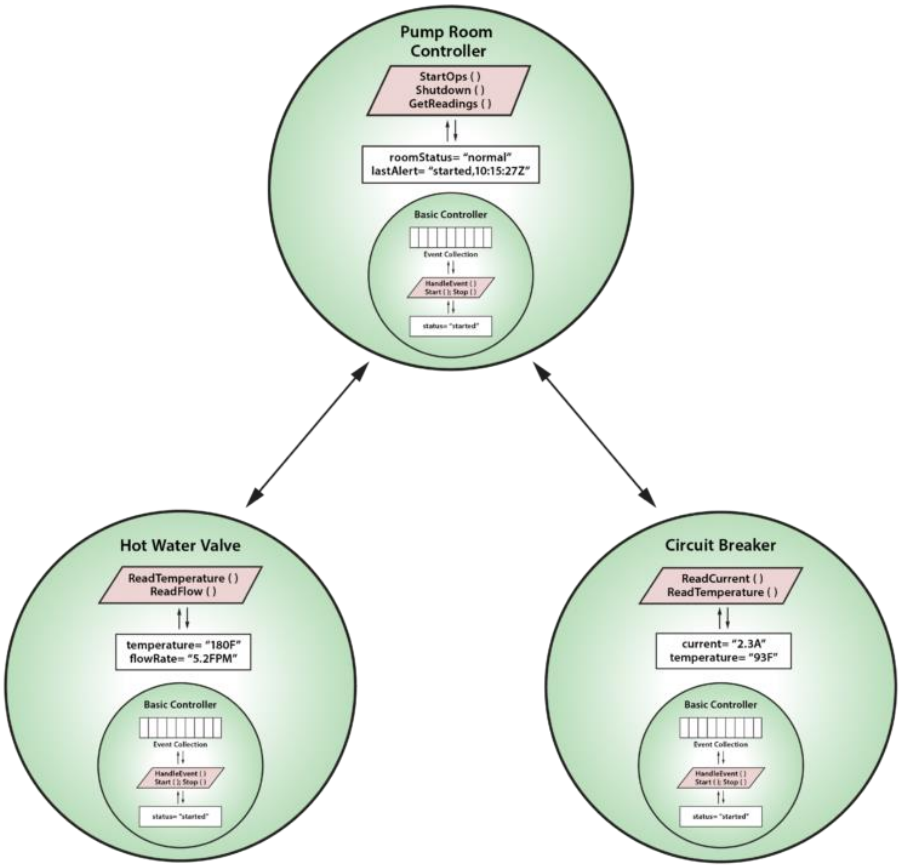
This subclass inherits all of the properties of a basic controller while adding new capabilities to manage specialized controller types. Using this object-oriented approach maximizes code reuse and saves development time.

Here's a Java example that illustrates how inheritance could be used to create the hot water valve class. It also shows how the hot water valve class can override the implementation of the Start and Stop methods defined by the basic controller:

```
public abstract class BasicController {
    protected List<Event> eventCollection;
    protected DeviceStatus status;

    public abstract void start();
    public abstract void stop();
    public abstract void handleEvent();
}

public class HotWaterValve extends BasicController {
    private double temperature;
    private double flowRate;

    public double readTemperature() {
        return temperature;
    }

    public double readFlowRate() {
        return flowRate;
    }

    @Override
    public void start() {…}

    @Override
```

```
    public void stop() {…}

    @Override
    public void handleEvent() {…}
}
```

As discussed in the previous section, applications also can build a hierarchy of digital twins that represent successively higher levels of analysis and management for complex systems, and this hierarchy can further leverage object-oriented techniques. Consider the following set of interconnected digital twin instances used in managing a hypothetical pump room:



In this example, the pump room has two digital twins connected directly to devices, one for a hot water valve and another for a circuit breaker. These twins are both implemented as subclasses of a basic controller and add properties and methods specific to their devices. They feed telemetry to a higher-level digital twin instance which manages overall operations for the pump room. This digital twin also can be implemented as a subclass of a basic controller even though it is not connected directly to a device. Note how both object inheritance and hierarchy play separate roles in defining the digital twin objects which work together to analyze event streams. Inheritance lets us refine the behavior of digital twin models to customize their actions, and hierarchy lets us build systems of interconnected digital twins which process events at successively higher levels of abstraction.

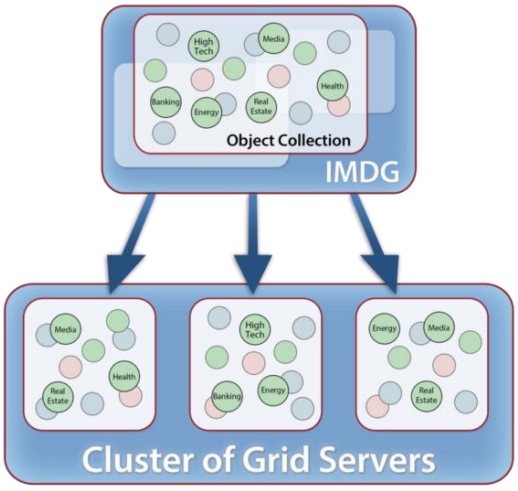
ScaleOut StreamServer® Software Architecture

This section explains how ScaleOut StreamServer's software-based, in-memory data grid (IMDG) and integrated stream-processing engine can host digital twin models and offer fast event handling, scalable performance, and high availability.

Hosting Digital Twins in an IMDG

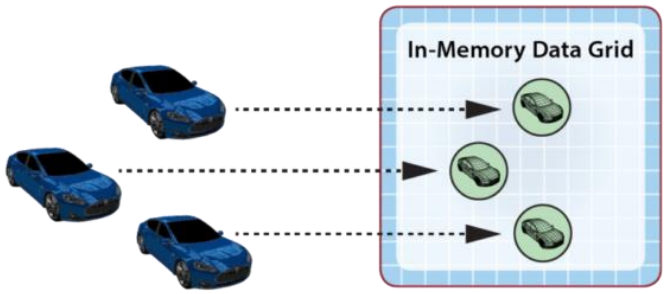
IMDGs have a long history dating back almost two decades. They were originally created as middleware software to host large populations of fast-changing data objects, such as ecommerce session-state and shopping carts, in memory instead of database servers, thereby speeding up access. An IMDG implements a software-based, key-value store of serialized objects that spans a cluster of commodity servers (or cloud instances). Its architecture provides cost-effective scalability and high availability, while hiding the complexity of distributed in-memory storage from the applications which use them. It also can take full advantage of a cluster's computing power to run application code *within* the IMDG — where the data lives — to maximize performance and avoid network bottlenecks.

Using an example from financial services, the following diagram illustrates an application's logical view of an IMDG as a collection of objects (stock sectors in this case) and its physical implementation as in-memory storage spanning a cluster of servers:

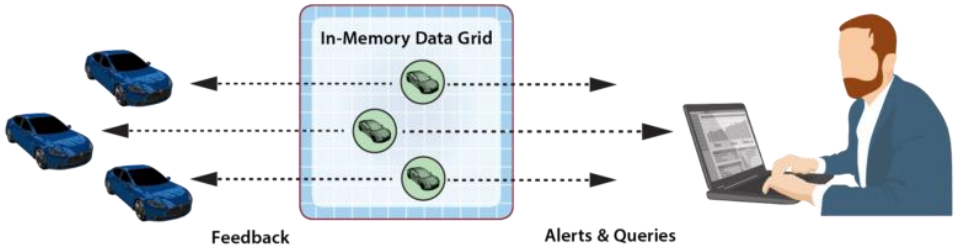


Because digital twin models are constructed using object-oriented techniques, IMDGs with integrated stream-processing are ideal for hosting them in memory and performing fast, scalable stateful-stream processing. An IMDG can scale its storage capacity and event-processing throughput by adding servers to hold many thousands of instances as needed for all the data sources and hierarchical digital twins. The grid's key-value storage model automatically correlates incoming event messages for the corresponding digital twin instance based on the data source's identifier and delivers them directly to the twin. The streaming engine then runs the model's application code for event ingestion, analysis, and alerting in real time and with low latency. Since the event message is delivered to its associated digital twin, no network overhead is incurred to access the twin's state information and event history.

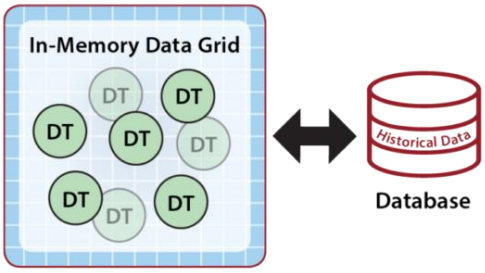
As an example, the following diagram illustrates the use of an IMDG to host digital twin objects for rental cars. The arrows indicate that the IMDG directs events from each car to its corresponding digital twin for event-processing.



As part of event-processing, digital twins can create alerts for human attention and/or feedback directed at the corresponding data source. In addition, the collection of digital twin objects stored in the IMDG can be queried or analyzed using data-parallel techniques (e.g., MapReduce) to extract important aggregate patterns and trends. For example, the rental car application could alert managers when a driver repeatedly exceeds the speed limit according to criteria specific to the driver's age and driving history. It also could allow a manager to query the status of a specific car or compute the maximum excessive speeding for all cars in a specified region. These data flows are illustrated in the following diagram:



Because they are hosted in memory, digital twin models can react very quickly to incoming events, and the IMDG can scale by adding servers to keep event processing times fast even when the number of instances (objects) and/or event rates become very large. Although the in-memory state of a digital twin consists of only the event and state data needed for real-time processing, the application can reference historical data from external database servers to broaden its context, as shown below. For example, the rental car application could access driving history only when incoming telemetry indicates a need for it. It also could store past events in a database for archival purposes.



In summary, what makes an IMDG an excellent fit for stateful stream-processing is its ability to transparently host both the state information and application code within a fast, highly scalable, in-memory computing platform and then automatically direct incoming events to their respective digital twin instances within the grid for processing. These two key capabilities, namely correlating events by data sources and analyzing these events within the context of the digital twin's real-time state information, enable IMDGs to offer a breakthrough approach to stateful stream-processing.

ScaleOut StreamServer's Software Architecture for Digital Twins

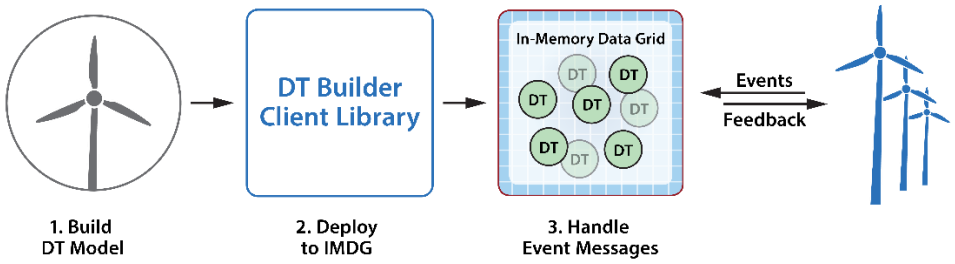
ScaleOut StreamServer provides a software-based execution platform for running digital twin models defined and deployed by a user application using the ScaleOut Digital Twin Builder Toolkit. The platform connects to various messaging endpoints, such as Azure IoT Hub, Kafka, and a REST service. Instances of these models are automatically created as necessary when incoming messages are received. These instances process messages using handler methods defined by the digital twin models. Within these handlers, an application can send messages to other digital twins or back to their data sources.

ScaleOut StreamServer stores instances of digital twin models as memory-based objects within its in-memory data grid (IMDG), which automatically distributes them across a cluster of servers. This provides transparent scaling of both memory capacity and throughput, keeping message-handling latency low even as the number of instances grows. It also ensures that messages and digital twin objects are collocated on the same server, avoiding network overhead which could increase latency and restrict throughput scaling.

Overview of Application Structure and APIs

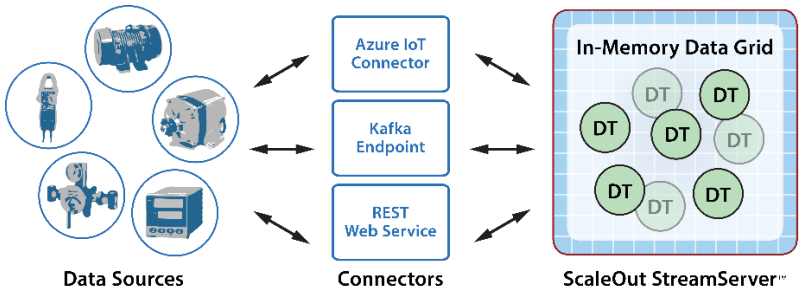
The APIs for deploying and managing digital twins are designed to make application development and deployment as straightforward as possible. They encapsulate the details of managing their implementation within the IMDG (such as object namespaces, invocation grids, and event posting) so that developers can focus on digital twin models and the exchange of messages with data sources.

The software development process is illustrated in the following diagram. After a new digital twin model is defined by an application, it is then deployed to the IMDG using the ScaleOut Digital Twin Builder API Libraries for Java and C#. This action causes the class definition and associated methods to be shipped to all servers within the IMDG in preparation for handling incoming messages.



Using the ScaleOut Digital Twin Builder API Libraries, an application can deploy connections to Azure IoT Hub, Kafka, and REST as illustrated in the following diagram. These connections deliver event messages from data sources to digital twin models and return alerts and commands back to data sources from these models. Azure IoT and Kafka connections are implemented in a scalable manner and run on all

servers in the IMDG. Likewise, multiple REST servers can be deployed behind a load-balancer to scale delivery of REST-based messages.



The IMDG automatically creates a new instance of a digital twin's state object when an event message first arrives from a data source. To maximize simplicity, all data sources are identified with a single string identifier, and digital twin instances are identified by a combination of a string model name (for example, "windmill" or "generator") and the instance identifier.

When available, multiple messages will be received by a connection and delivered to a digital twin instance as a group for handling with a single call to the digital twin instance. This minimizes message handling latency and maximizes overall throughput.

To ensure that messages are not lost, the Azure IoT Hub and Kafka connections do not acknowledge the receipt of messages from data sources until they have been processed within the IMDG. The REST API synchronously awaits the processing of an event message before returning to the client application. Note that message processing implements "at least once" delivery, and applications need to expect possible re-delivery if a server within the IMDG fails during processing prior to acknowledging reception to the connection.

ScaleOut Digital Twin Builder API Libraries

This section describes the API libraries supplied with the ScaleOut Digital Twin Builder for building stateful stream-processing applications using the digital twin model. Their goal is to enable developers to focus on building and deploying digital twins without the need to handle low level details regarding their in-memory storage and message processing within the IMDG.

Overview

The ScaleOut Digital Twin Builder includes API libraries for both Java and C# applications. API libraries for implementing digital twin models include the following components:

- a base class definition for a digital twin model's state object which describes the state information to be maintained by each instance of the model
- a base class definition for a message processor which describes the application code that processes incoming messages using the state object and its associated message list
- a processing context class that supplies context information to a message processor and includes utility methods, for example, for sending a message back to a data source
- API libraries for connecting ScaleOut StreamServer to Microsoft Azure IoT Hub and Kafka to exchange messages with data sources
- API libraries for sending messages to digital twins from Java and C# client applications

Note that messages are assumed to have a user-defined type. Messages can be sent from any data source (or a digital twin) to either Java or C# digital twins as long as they can be correctly interpreted using this type. All messages are assumed to be serialized using JSON, and JSON serialization is used to store state objects within the IMDG.

To minimize storage requirements and processing latency, a record of incoming messages is not maintained by default. Instead, a digital twin state object can incorporate time-ordered message lists as part of its dynamic state, and these lists can be managed and analyzed in using ScaleOut's time windowing libraries. These lists can be used to record all incoming messages and/or significant events and alerts, as dictated by the needs of the application.

Additional API libraries are provided to deploy digital twin models to the IMDG and to send messages to instances of DT models from C# and Java applications. As discussed above in Building a Hierarchy of Digital Twins, instances of digital twins that connect to data sources can exchange messages with higher level twins in a logical hierarchy defined by the application.

The following sections take a closer look at the API components.

Digital Twin State Object

This object holds application-defined, in-memory state information for each an instance of a digital twin model. It is used to provide deep context for analyzing incoming messages. Because it is stored within the IMDG, it is immediately available to application code for access and updating when a message arrives.

For each digital twin model, a state object is defined by the application as a subclass of a toolkit-defined base type. In Java, the base type is as follows:

```
public abstract class DigitalTwinBase {
    protected String Id;
    protected String Model;

    public void init(String id, String model) {
        this.Id    = id;
        this.Model = model;
    }
}
```

In C#, the base type looks like this:

```
public abstract class DigitalTwinBase
{
    public string Id { get; private set; }
    public string Model { get; private set; }

    public void Init(string id, string model)
    {
        this.Id    = id;
        this.Model = model;
    }
}
```

Messages

Messages sent to digital twin instances from data sources have an arbitrary, application-defined payload and are assumed to be serialized using JSON. They are addressed to a digital twin model and instance identifier, both of which are specified as strings. Once received by an application-defined message processor, messages may be deserialized as an instance of a language-defined type. It is the responsibility of the application to avoid de-serialization exceptions.

Messages may be sent from data sources to digital twins, from digital twins back to their respective data sources, and from digital twins to other digital twins. Messages can represent events, alerts, or commands. Their interpretation is defined by application-defined message components and messaging processing code.

Message Processor

A message processor describes the application code used to receive and analyze incoming messages for an instance of a digital twin. In particular, it specifies the set of parameters and return value for application-defined methods used by digital twin models to process messages.

In Java, the message processor abstract base type is defined as follows:

```
public abstract class MessageProcessor<T extends DigitalTwin, V>
    extends MessageProcessorBase<T> implements Serializable
{
    public abstract ProcessingResult processMessages(
        ProcessingContext context, T stateObject,
        Iterable<V> incomingMessages) throws Exception;

    @Override
    public ProcessingResult processMessages(
        ProcessingContext context, T twin, MessageListFactory factory)
        throws Exception {
            Iterable<V> incoming = factory.getIncomingMessages();
            this.processMessages(context, twin, incoming);
    }
}
```

In C#, it is defined as follows:

```
public abstract class MessageProcessor<TDigitalTwin, TMessage> :
    MessageProcessor where TDigitalTwin: class
{
    public abstract ProcessingResult ProcessMessages(
        ProcessingContext context, TDigitalTwin digitalTwin,
        IEnumerable<TMessage> newMessages);

    internal override ProcessingResult ProcessMessages(
        ProcessingContext context, DigitalTwinBase digitalTwin,
        IMessageListFactory messageListFactory)
    {
        IEnumerable<TMessage> newMessages =
            messageListFactory.GetIncomingMessageList<TMessage>();

        return ProcessMessages(context, digitalTwin as TDigitalTwin,
            newMessages);
    }
```

```
}
```

The `processMessages` method supplies the following parameters to the application's code:

- a processing context, which supplies a method that the application can use to send messages back to its respective data source
- the digital twin instance's state object
- a list of incoming messages from the data source to be handled by the message processor

This method returns a result of type `ProcessingResult` that indicates whether the state object or message list has been modified and should be updated in the in-memory data grid.

Processing Context

An instance of a processing context is passed to the application's message processor to provide context-specific features, in particular, a method for sending messages back to the digital twin's data source.

The Java version is as follows:

```
public abstract class ProcessingContext implements Serializable {
    public abstract SendingResult sendToDataSource(byte[] payload);
}
```

In C#, the processing context is defined as follows:

```
public abstract class ProcessingContext
{
    public abstract string DataSourceId { get; }
    public abstract string DigitalTwinModel { get; }

    public abstract SendingResult SendToDataSource(byte[] message);
}
```

Application Endpoint

Client applications can serve as data sources for digital twins and send messages to their counterparts running within the ScaleOut StreamServer. The ScaleOut Digital Twin Builder includes APIs in Java, C#, and REST for sending messages.

A Java application can send a message to a digital twin with the following static method defined in the `AppEndpoint` class, which is located in the digitaltwin-datasource library:

```
public static SendingResult send(String model, String id, byte[] jsonMessage)
```

The `model` parameter specifies the name of the digital twin model, and the `id` parameter specifies an identifier for the instance of the model, i.e., a specific data source. All messages are assumed to be serialized in JSON.

Likewise, a C# application can send a message to a digital twin as follows:

```
public static SendingResult Send(string model, string id, byte[] jsonMessage)
```

An application can also post a message to a digital twin using a web service, which is downloaded as a Docker image from docker.com. Here's a simple JavaScript sample that accesses the web service on the local system:

```
const serverUrl = "https://localhost";

/// Send a message to a specified Digital Twin state object.
/// The message must be in JSON format -- e.g., JSON.stringify(message)
function sendMessage(twinModel, twinId, message) {
    let xhttp = new XMLHttpRequest();
    xhttp.open("POST",`${serverUrl}/api/messages/${twinModel}/${twinId}`,
               true);
    xhttp.setRequestHeader("Content-type", "application/json");
    xhttp.send(message);
}
```

Sample Programs

The following sample programs illustrate the use of these concepts in building a simple digital twin model to process streaming events. These samples illustrate how digital twins can maintain parameters and dynamic state information that are unique to their respective data sources. This information enables deeper introspection on the contents of incoming messages than would otherwise be possible.

Heart-Rate Tracking Sample (Java)

This sample tracks heart-rate telemetry from a wearable device called a heart-rate tracker and detects incidents of heart-rate spikes that might indicate a risk to the user. The device sends periodic messages reporting heart-rate with a timestamp to its digital twin. The digital twin's state object includes user-specific parameters, such as age, body mass index, and a maximum allowed heart-rate that is not considered a spike. It also includes dynamic state variables that track the duration of an incident in which heart-rate spikes are occurring. The message processor code looks for heart-rate spikes reported by incoming messages and then tracks the duration of each incident in which they are occurring. It also alerts the device when the duration exceeds allowable limits based on the user's parameters.

A message from the device is defined as follows:

```
public class HeartRate {
    private int _hr;   // heart-rate
    private long _ts;  // timestamp

    public HeartRate(int hr, long timestamp) {
        _hr = hr;
        _ts = timestamp;
    }

    public long getTimestamp() {
        return _ts;
    }

    public int getHeartrate() {
        return _hr;
    }

    public byte[] toJson(){return …}  // serialize to JSON
}
```

An alert message back to the device can be defined as follows:

```
public class HeartRateAlert {
```

```
    private String _msg;  // alert message
    private long   _ts;   // timestamp

    public HeartRateAlert(String msg, long timestamp) {
        _msg = msg;
        _ts  = timestamp;
    }

    public byte[] toJson() {return …}  // serialize to JSON
}
```

A very simple digital twin state object for the heart-rate tracker which includes user parameters and dynamic state information can be defined as follows. Some of the details regarding helper classes are omitted.

```
public class HeartRateTracker extends DigitalTwinBase {
    // user's parameters:
    private AgeRange      userAgeRange;
    private BodyMassIndex userBmi;
    private Medication    userMedication;

    // dynamic state for tracking heart-rate spikes:
    private int           heartRateSpikeCount;
    private long          maxHeartRate;
    private boolean       heartRateSpikeInProgress;
    private long          heartRateSpikeStartTime;
    private List<Alert>   alertList;

    public HeartRateTracker() {
        // handle tracker initialization
    }

    public AgeRange getUserAgeRange() {
        return userAgeRange;
    }

    public BodyMassIndex getUserBmi() {
        return userBmi;
    }

    public Medication getMedication() {
        return userMedication;
    }

    public int getHeartRateSpikeCount() {
        return heartRateSpikeCount;
    }

    public long getMaxHeartRate() {
        return maxHeartRate;
    }

    public boolean isSpikeInProgress() {
        return heartRateSpikeInProgress;
    }
```

```
    public long getSpikeStartTime() {
        return heartRateSpikeStartTime;
    }

    public void stopSpikeTracking() {
        heartRateSpikeCount      = 0;
        heartRateSpikeInProgress = false;
    }

    public void startSpikeTracking(long timestamp) {
        heartRateSpikeCount++;
        heartRateSpikeInProgress = true;
        heartRateSpikeStartTime = timestamp;
    }

    public void recordAlert(Alert alert) {
        // use the TimeWindowing library to add to list in time-order:
       Utils.addTimeOrdered(alerts, HeartRateAlert::getTimestamp, alert);
}}
```

This digital twin's message processor is as follows:

```
public class HeartRateMessageProcessor extends
    MessageProcessor<HeartRateTracker, HeartRate> {

    @Override
    public ProcessingResult processMessages(
               ProcessingContext processingContext,
               HeartRateTracker tracker,
               Iterable<HeartRate> incomingMessages) throws Exception
    {
        ProcessingResult result = NoUpdate;
        long unixTimeNow = System.currentTimeMillis();

        // analyze incoming messages:
        for(HeartRate msg : incomingMessages)
        {
            // if the message indicates a heart-rate spike, track it:
            if(incomingMessage.getHeartrate() > tracker.getMaxHeartRate())
            {
                // if this is the first spike, start tracking an incident:
                if (!(tracker.isSpikeInProgress()))
                {
                    tracker.startSpikeTracking(msg.getTimestamp());

                    result = UpdateDigitalTwin;
                }

                // alert user if spike reached duration that exceeds
                // limit allowed by user's age, bmi, or medication:
                long duration = unixTimeNow - tracker.getSpikeStartTime();

                if (duration > tracker.getUserAgeRange().getThreshold() ||
                    duration > tracker.getUserBmi().getThreshold() ||
                    duration > tracker.getMedication().getThreshold())
                {
                     // send alert message to data source:
```

```
                    HeartRateAlert alert = new HeartRateAlert(
                       String.format(
                      "Heart-rate exceeded (%d, %d) - current hr %d for %d ms.",
                       tracker.getUserAgeRange().getThreshold(),
                       tracker.getUserBmi().getThreshold(),
                       msg.getHeartrate(),
                       duration), unixTimeNow);

                    processingContext.sendToDataSource(alert.toJson());

                    // record the alert for time-window analysis:
                    tracker.recordAlert(alert);

                    result = UpdateDigitalTwin;
                }
            }

            // otherwise, if the incident has ended, reset the tracking state:
            else if (tracker.isHeartRateSpikeInProgress())
            {
                tracker.stopSpikeTracking();
                result = UpdateDigitalTwin;
            }
        }
        return result;
    }
}
```

A client application can deploy this digital twin model to ScaleOut StreamServer's in-memory data grid by creating an execution environment which is deployed to all servers in the grid:

```
ExecutionEnvironment environment = new ExecutionEnvironmentBuilder()
    .addDependencyJar("HeartRateSample.jar")
    .addDigitalTwin("HeartRateTracker", new HeartRateMessageProcessor(),
        HeartRateTracker.class, HeartRate.class)
    .build();
```

Likewise, the application can deploy a connection to Kafka for exchanging messages with data sources as follows:

```
KafkaEndpoint HearRateMessageSource = new KafkaEndpointBuilder(
    new File("server.properties"))
    .addTopic("HRTdataSources", "device_to_dt", "dt_to_device")
    .build();
```

For test purposes, a client program can send a message to instance "tracker_1717" of a HeartRateTracker digital twin as follows:

```
byte[] msg = new HeartRate(130, System.currentTimeMillis()).toJson();

SendingResult result = AppEndpoint.send("HeartRateTracker",
                           "tracker_1717", msg);
switch (result) {
    case Handled:
        System.out.println("Message was delivered and handled.");
        break;
```

```
    case NotHandled:
        System.out.println("Message was not handled.");
        break;
}
```

Wind Turbine Tracking Sample (C#)

This sample demonstrates how telemetry from a wind turbine can be tracked and analyzed. As with the previous sample, it illustrates how a digital twin can track both parameters and dynamic state variables for each data source and use this information to introspect on the significance of incoming messages.

In this sample, messages from a wind turbine report the turbine's temperature with a timestamp. If a high temperature ("overtemp") condition is detected, its duration is tracked. If the wind turbine is in a pre-maintenance period, an alert is sent to the data source after a shorter duration than is permitted under normal conditions. The length of the pre-maintenance period is determined by the wind turbine's model. A list of incidents is maintained for future analysis. This list records when the turbine enters an overtemp condition, is alerted, and when the overtemp condition is resolved.

Telemetry messages from the wind turbine are defined as follows:

```
public class DeviceTelemetry
{
       public int Temp { get; set; }
       public DateTime Timestamp { get; set; }
}
```

Here are some other helper data structures used in the state object and message processor:

```
public enum IncidentType
{
   OverTempDetected,
   OverTempAlert,
   OverTempResolved
}

public enum WindTurbineModel
{
    Model17,
    Model18,
    Model19
}

public class Incident
{
       public Description IncidentType { get; set; }
       public DateTime TimeStamp { get; set; }
       public int MetricValue { get; set; }
       public bool InPreMaintPeriod { get; set; }
}

public class Alert
{
       public Description IncidentType { get; set; }
       public DateTime TimeStamp { get; set; }
       public TimeSpan Duration { get; set; }
       public bool InPreMaintPeriod { get; set; }
```

```
}
```

The wind turbine's digital twin state object contains the following parameters and state variables:

```
public class WindTurbine : DigitalTwinBase
{
    // physical characteristics:
    public const string DigitalTwinModelType = "windturbine";
    public WindTurbineModel TurbineModel { get; set; }
        = WindTurbineModel.Model7331;
    public DateTime  NextMaintDate { get; set; }
        = new DateTime().AddMonths(36);
    public const int MaxAllowedTemp = 100; // in Celsius
    public TimeSpan  MaxTimeOverTempAllowed
        = TimeSpan.FromMinutes(10);
    public TimeSpan  MaxTimeOverTempAllowedPreMaint
        = TimeSpan.FromMinutes(2);

    // dynamic state variables:
    public bool TrackingOverTemp { get; set; }
    public DateTime OverTempStartTime { get; set; }
    public int NumberMsgsWithOverTemp {get; set;}

    // list of incidents and alerts:
    public List<Incident> IncidentList { get; } = new List<Incident>();
}
```

The message processor code for this digital twin is defined as follows.

```
public class WindTurbineMessageProcessor : MessageProcessor<
    WindTurbine, DeviceTelemetry>
{
    static Dictionary<WindTurbineModel, TimeSpan> _preMaintPeriod;

    public override ProcessingResult ProcessMessages(
        ProcessingContext context, WindTurbine dt,
        IEnumerable<DeviceTelemetry> newMessages)
    {
        var result = ProcessingResult.NoUpdate;

        // determine if we are in the pre-maintenance period for this
        // wind turbine model:
        var preMaintTimePeriod  = _preMaintPeriod[dt.TurbineModel];
        bool isInPreMaintPeriod = ((dt.NextMaintDate
                 - DateTime.UtcNow) < preMaintTimePeriod) ? true : false;

        // process incoming messages:
        foreach (var msg in newMessages)
        {
            // if message reports a high temp indication, track it:
            if (msg.Temp > WindTurbine.MaxAllowedTemp)
            {
                dt.NumberMsgsWithOverTemp++;

                if (!dt.TrackingOverTemp)
                {
                    dt.TrackingOverTemp  = true;
```

```
            dt.OverTempStartTime = DateTime.UtcNow;

            // add a notification to the incident list:
            dt.IncidentList.Add(new Incident() {
                Description = IncidentType.OverTempDetected,
                TimeStamp   = dt.OverTempStartTime,
                MetricValue = msg.Temp,
                InPreMaintPeriod = isInPreMaintPeriod });
             }
        }

        TimeSpan duration = DateTime.UtcNow -
                                 dt.OverTempStartTime;

        // If we have exceeded the max allowed time for an over-
        // temperature condition in either a normal or pre-maint.
        // condition, send an alert:
        if (duration > dt.MaxTimeOverTempAllowed ||
               (isInPreMaintPeriod && duration >
                dt.MaxTimeOverTempAllowedPreMaint))
        {
            var alert = new Alert();

            alert.Description  = IncidentType.Overtemp_alert;
            alert.TimeStamp    = DateTime.UtcNow;
            alert.Duration     = duration;
            alert.IsInPreMaintPeriod = isInPreMaintPeriod;

            // send message back to data source:
            context.SendToDataSource(Encoding.UTF8.GetBytes(
                JsonConvert.SerializeObject(alert)));

            // add a notification to the incident list:
            dt.IncidentList.Add(new Incident() {
                Description = IncidentType.Overtemp_alert,
                TimeRegistered = DateTime.UtcNow,
                MetricValue =
                        dt.NumberMsgsWithOverTemp,
                InPreMaintPeriod = isInPreMaintPeriod });
        }

        result = ProcessingResult.DoUpdate;
    }
    // stop tracking the condition and reset our state:
    else if (dt.TrackingOverTemp)
    {
        dt.TrackingOverTemp       = false;
        dt.NumberMsgsWithOverTemp = 0;

        // Add a notification to the incident list:
        dt.IncidentList.Add(new Incident() {
            Description = IncidentType. OverTempResolved,
            TimeStamp   = DateTime.UtcNow,
            MetricValue = msg.Temp,
            InPreMaintPeriod = isInPreMaintPeriod });

        result = ProcessingResult.DoUpdate;
```

```
            }
        }
        return result;
    }
}
```

This sample program can be deployed by a client application as follows:

```
ExecutionEnvironmentBuilder builder = new ExecutionEnvironmentBuilder()
    .AddDependency(@"WindTurbine.dll")
    .AddDigitalTwin<WindTurbine, WindTurbineMessageProcessor,
        DeviceTelemetry>(WindTurbine.DigitalTwinModelType);

ExecutionEnvironment execEnvironment = await builder.BuildAsync();
```

A client application can then send a message to wind turbine instance “WT17” as follows:

```
SendingResult result = AppEndpoint.Send("WindTurbine", "WT17", jsonMessage);
if (result == SendingResult.Handled)
    Console.WriteLine("Message was delivered and processed successfully");
else
    Console.WriteLine("Failed to process message");
```

Next Steps

The ScaleOut Digital Twin Builder toolkit provides powerful, easy to use APIs for building digital twin models, deploying them for execution on ScaleOut StreamServer, and using them to provide deep introspection on the dynamic behavior of data sources. This toolkit was designed to dramatically simplify the use of ScaleOut’s in-memory data grid for running digital twin models, while enabling applications to take full advantage of the platform’s power: fast message processing, scalability, and integrated high availability.

Here are the steps you should follow to get started using ScaleOut Digital Twin Builder:

- Download ScaleOut StreamServer from scaleoutsoftware.com and install it using a license key obtained from ScaleOut Software.
- If you are developing a Java digital twin, you can find an open source, Apache-licensed library for building digital twin models on GitHub here. The other Java libraries you will need are located in ScaleOut StreamServer’s installation folder within the JavaAPI folder (Windows) or java_api directory (Linux).
- If you are developing a C# digital twin, please download a NuGet package from nuget.org here.
- If you want to write an application which posts messages to ScaleOut digital twin web service, please download the web service from docker.com here.

We invite your questions and feedback! Please contact us at support@scaleoutsoftware.com.