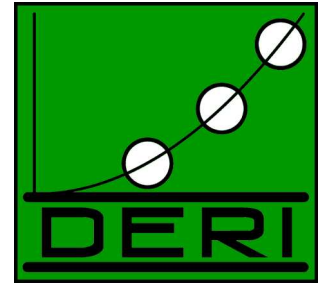


DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE



ENVOY: A PLATFORM FOR COOPERATING  
WIDGETS ON THE WEB

Ronan Fox Manfred Hauswirth

DERI Technical Report 26-01-2009

January 2009

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

**DERI Galway**

IDA Business Park

Dangan

Galway, Ireland

[www.deri.ie](http://www.deri.ie)

# Envoy: A Platform for Cooperating Widgets on the Web

Ronan Fox

*Digital Enterprise Research Institute, National University of Ireland, Galway,  
IDA Business Park, Galway, Ireland*

Manfred Hauswirth

*Digital Enterprise Research Institute, National University of Ireland, Galway,  
IDA Business Park, Galway, Ireland*

Abstract: Widgets have the potential to improve the way in which applications are developed on the web. With HTML 5 widgets can now communicate with one another and their hosting platforms. However, there exists the problem of how to enable this communication in a scalable, flexible, and robust manner. There is no existing framework which enables heterogeneous widgets to be used together, without a-priori knowledge of each widget's interface protocol. In this paper we propose Envoy - a semantic widget engine - which supports the reuse of functionality encapsulated by intercommunicating web widgets in web applications through the use of a new design methodology which incorporates the semantic description of interfaces, publishing those interfaces, searching for them, and combining them into fully functional, context-aware web applications. As the communication paradigm enables interface reuse, this is particularly well suited to the asynchronous communication style of typical widget-based applications.

*Widget, web 2.0, semantic widget engine, event processing*

## 1 Introduction

Component-based application development is well established in the software engineering community. Interface standards such as COM[1], CORBA[2], and REST[3] all have enabled the reuse of software components in various distributed settings[4]. At the back-end of web applications component-based software development has been used to provide the business logic and data access functionality. The task of programming such functionality is largely composed of combining existing components from well-established component libraries and customizing and extending them for application specific tasks. However, this

methodology has not made it out onto the web. Typical web applications are monolithic, specialized, and non-customizable. Beyond changing the look and feel of the user interface, there is not a lot the user can do to change the way an application works.

To access different functionality on the web the user must visit different websites. These websites have no relationship and user-entered information on one site will not make its way to a second site where it could be given a different context. Potentially important events on one site go unnoticed by another, when that second site could. There is a lack of integration and collaboration between these applications and their information sources. In addition, to access any of these information sources the user will probably have profiles on several web sites, which all have to be maintained and accessed separately.

Web widgets have emerged as a mechanism for encapsulating pieces of functionality within portable fragments of HTML and JavaScript code. Web widgets are an important enabler for Web 2.0 – customizing applications, enabling viral use, supporting reuse in different contexts for different users and allowing the introduction of dynamic content to sites with minimal effort. They represent standalone functionality that can be used in many contexts. The ability for the user to create personalized home pages through the use of web widgets has been enabled through sites such as iGoogle[5] with over 45,000 widgets, PageFlakes[6] with 236,000 widgets, and NetVibes[7] with over 144,000 widgets. Other sites act as libraries where users can import widgets into existing blogs or web pages. Sites in this category include Widgetbox[8] with over 82,000 widgets, and Spring Widgets[9] with over 3,700 widgets. Behind these sites lie widget engines which manage the storage, access, management, and installation of widgets into the chosen environments. Despite this large selection, the problems of enabling flexible intercommunication while decoupling interfaces still pertain and building portable, reusable widget applications is still not possible.

Web widgets currently make no attempt to automatically recognize the context into which they are deployed, nor do they make any attempt to integrate with either the web page on which they find themselves or with other widgets which

may be deployed on the same web page. This has limited their usefulness and the possibilities for meaningful integration. This widget isolation is potentially remedied with the publication of the HTML 5 [10] standard. Using this standard web widgets are provided with the means to intercommunicate with other widgets and the containing web pages.

However, none of the web widget engines identified to date support the exchange of data using the HTML 5 standard. There are several reasons for this:

1. The views hosting the widgets have no intelligence and do not understand the semantics of the data being exchanged.
2. Widgets do not conform to any “global standard” reflecting the semantics of the data being exchanged, and its format.
3. Intercommunicating widgets are developed by independent software developers and yet each widget developer must have a-priori knowledge of the exact environments into which his widget is installed and used.
4. The widgets in any specific view must be tightly coupled to one another, increasing dependencies, and reducing reusability.

It is possible, therefore, that the constructs provided as part of HTML 5 will remain unused in existing widget engines. Widgets will remain as “toys” with no interaction with the outside world represented by other widgets which may exist in the same view.

This paper describes Envoy, a semantic widget engine which is a design methodology aligned with an architectural style. Envoy provides a simple means to combine functionalities into applications, building on the simple mechanisms already provided by widgets to create graphical user interfaces and encapsulate functionality. We base our work on an architectural style which combines elements of workflows, Event Condition Action (ECA), Event Based Integration (EBI), and semantics to provide an inclusive environment for the semantic integration of web widgets and rapid application development.

Envoy requires several processes to support the ad-hoc deployment of widgets into user-specified views. For this part of our research we draw inspiration from

research into domain oriented design environments[11], agile software development[12] and collaborative development processes[13], where the design of applications is moved into the hands of the users, and where evolution of software is based on a collaboration between users and widget developers. These software design workflows allow the integration and reuse of decoupled components using ECA concepts.

This paper is structured as follows: Section 2 describes the fundamental models underlying Envoy. Section 3 describes the run-time architecture which builds upon these models. Section 4 details the software design methodology to be used by web widget developers in the description, publication, search, and combination of widgets. Section 5 describes the deployment workflow to install intercommunicating web widgets to create web-widget based applications. Section 6 describes problems associated with creating integrated, widget-based applications on the Internet using the current infrastructures and widget engines, and how our system solves them. Section 7 discusses related work in workflow, event based integration, event condition action, and semantic web services. Section 8 discusses Envoy and its implications for application development on the Internet, including its relationship to existing widget engines.

## **2 An Example Widget Scenario**

Widgets enable the simple reuse of existing functionality on the web and thus provide a suitable base for components in building personalized applications for individual users and/or groups of users with similar interests.

In this section we provide an overview of the shortcomings of the current widget infrastructure in enabling widget-based applications through an example so that the scope and context of Envoy can be understood.

### **2.1 The Use Case**

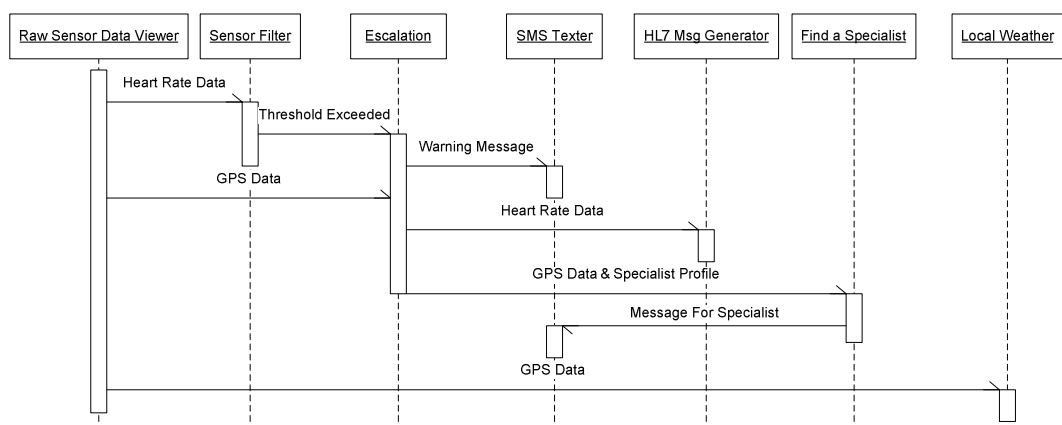
For our use case we draw upon the world of telehealth - the delivery of health related services using telecommunications technology. We can visualize a user having telehealth requirements, aligned with other non-medical requirements so

that a home page would reflect more clearly a user's needs – health within a social context.

Mary suffers from chronic heart failure (CHF) which means that she needs to be monitored remotely while she is in the community. Mary also has interests in outdoor pursuits, and occasionally goes hill walking. Mary is an active web user and uses it to find breaking news about her health, about sports, and about hill walking. When Mary is out hill walking she wants to know the local weather forecast and also wants to feel assured that if her heart rate jumps unexpectedly she will be able to receive appropriate treatment.

From a technical standpoint we know that Mary is different to every other sufferer of CHF and hill walker. Some are more active, they all have differing interests, and their social circumstances differ. An application delivered on the web will need to be different (customizable) for each user. Widgets provide a mechanism for encapsulating functionality from a variety of web sites and thus can act as building blocks for web applications. Combining them together in novel ways will enable the satisfaction of users' diverse requirements in a more adaptable way than current "customizable" web applications. Current widget engines go a certain distance, but as pointed out above, they do not extend far enough to allow intercommunicating, heterogeneous web widgets be reused in various contexts by various users.

The scenario above can be represented in the UML Sequence Diagram below.



**Fig. 1** Combined Widget Telehealth/Lifestyle Application

In all cases described below, the widgets use HTML 5 constructs to send data and register as “Sources”. Those widgets listening for data also use HTML 5 constructs to register as “Sinks”. All data originating from Sources reach the Sinks via the viewer hosting the widgets.

The communication “chain” starts with a “Raw Sensor Data Viewer” widget. This widget gets regular updates from two on-body sensors worn by Mary – a heart beat monitor on her wrist and a GPS sensor in her PDA. Thus, the “Raw Sensor Data Viewer” widget acts as a Source. Mary’s PDA is capable of serving the sensor data in such a way that the viewer widget can display (either through a web service mechanism, or through the direct placement of files in specific locations). The Sensor Viewer widget then uses the HTML 5 constructs to send the sensor data to the viewer – for simplicity we assume that the sensor viewer uses separate constructs for each sensor.

The hosting viewer retains a “hard-coded” list of widgets waiting for input and forwards the data to the relevant widgets using another series of HTML constructs. The “Sensor Filter” receives the heart beat and the “Weather Forecast” widget receives the GPS coordinates – both acting as Sinks. The “Weather Forecast” widget adjusts the forecast to match Mary’s current location and gives her instant feedback. The “Sensor Filter” widget has a threshold – if the heart beat rate goes over this threshold value (say 110 beats per minute) then it again uses HTML 5 constructs to send data (acting as a Source) to the “Escalation” widget via the viewer.

The “Escalation” widget then sends data (via the viewer) to the “SMS Texter” widget, the “HL7 Generator” widget, and the “Find a Specialist” widget. Each of these widgets performs a very specific function:

1. The “SMS Texter” widget parses the message to find the message to send and the phone number to which it should be sent. It then sends an SMS text to the number (Mary’s) with potentially a warning message included.
2. Health Level 7 (HL7) [14] is the most widely adopted standard for the exchange of information between electronic patient record systems. The

“HL7 Generator” widget formulates a HL7 Version 3 message and converts it to HL7 Version 2 for subsequent forwarding to the receiving GP/hospital.

3. The “Find a Specialist” widget uses the combination of the GPS coordinates from Mary’s PDA with the warning (as received in the message received from the “Escalation” widget), to locate the nearest cardiologist. It then posts a message to the “SMS Texter” widget with a warning message for that specialist, along with Mary’s contact details.

These interactions place the following requirements on the widget engine infrastructure:

1. The viewer must establish which one of three sources a particular message is coming from. Thus, the viewer must have a-priori knowledge of the widgets which act as Sources.
2. The viewer must know the destination of each message to which it wishes to send. These destinations act as Sinks in this context.
3. The viewer must understand the requirements of each widget acting as Sinks. This is shown clearly when the Source is the Escalation web widget. When a message from the Escalation widget reaches the viewer, the viewer must then send data to three other widgets – in the manner required by those widgets. The “SMS Texter” widget does not want the data to be just forwarded, but requires a phone number and a string representing the warning message, whereas the “Find a Specialist” widget requires both the GPS coordinates of the patient in emergency situations. Thus the viewer must mediate between the widgets so that the data expected by each widget is sent in the format required.

## 2.2 Issues

This example scenario, though simplistic, is sufficient to uncover the faults with current mechanisms which could be used to manage the flow of data between widgets and the hosting viewer.

There are a number of requirements to enable these widgets to work together in existing widget engines:

1. The widget developers have agreed a set of parameters, and exchange protocols for the postMessage calls.
2. Any new widgets must adhere to the established “standards” in use in the view.

The disadvantages of this approach are:

1. The containing web page must have knowledge of the widgets to be hosted, the data to be transferred, and the specific role requirements of each widget (Sinks and Sources). This clearly makes it impossible for existing widget engines to server these specific, user-focused pages in a sustainable, scalable way. The widgets have a-priori knowledge of one another.
2. The widgets are tightly coupled and thus the overall application is resistant to change. Any one widget changing could break the entire application.
3. The widgets cannot be used in different contexts, if those contexts have different “standards”.
4. The widgets will only interact with other widgets in a predefined manner.
5. No reuse of the integration effort in this view is possible in another view.

### **2.3 Solution Requirements**

The overriding requirement we have is to enable dynamic, customized application development. This means enabling software developers and indeed casual users to create applications based on reusable, intercommunicating widgets, for different uses, in different contexts.

A number of key concepts are required to decouple the widgets, to make them flexible, to allow dynamic, customized application development:

1. A decoupled run time which will enable multiple widgets to communicate without a-prior knowledge of each others existence.
2. A setup environment which allows the developer register widgets and which will also allow a casual user specify how the widget will interact

with other widgets in the view under construction. This environment must be simple and intuitive to use.

### 3 Widget Models

This section details the models which are used to describe generic, intercommunicating widgets. Based on these models, the widget developers can describe a new widget in enough detail in order that Envoy can access it and provide a grounding mechanism for interoperation. Domain models, or ontologies, are employed as part of the model and these will enable users to specify the context for the widgets deployed in individual views.

To successfully enable a decoupled environment, where independently created, heterogeneous widgets can interoperate, two sets of models are required:

1. A model for describing individual widgets and their interfaces as defined by their inputs and outputs. This model will allow for discovery, data type mediation, and other lower level requirements.
2. A model for describing the context of the widgets in terms of the user and his requirements for the view in which the widget has been installed. This model will allow for heterogeneous publications and subscriptions, workflow, and analysis of the communication between the widgets installed in the view.

Ontologies will give us an explicit means of defining widgets and their interactions, and their subsequent relationship within user or domain-based views. They provide a higher-level linkage between ontologies where sinks and sources are defined in terms of concepts, attributes, and relationships.

The models described in the following section address the following requirements:

1. Search and location of widgets based on semantic definitions.
2. Automated application building through user profile analysis matched to available semantically-defined templates.
3. Decoupled communication amongst widgets through semantic mediation

4. Enabling context for widgets applied to semantic event processing.
5. Knowledge encapsulation for adaptive views.

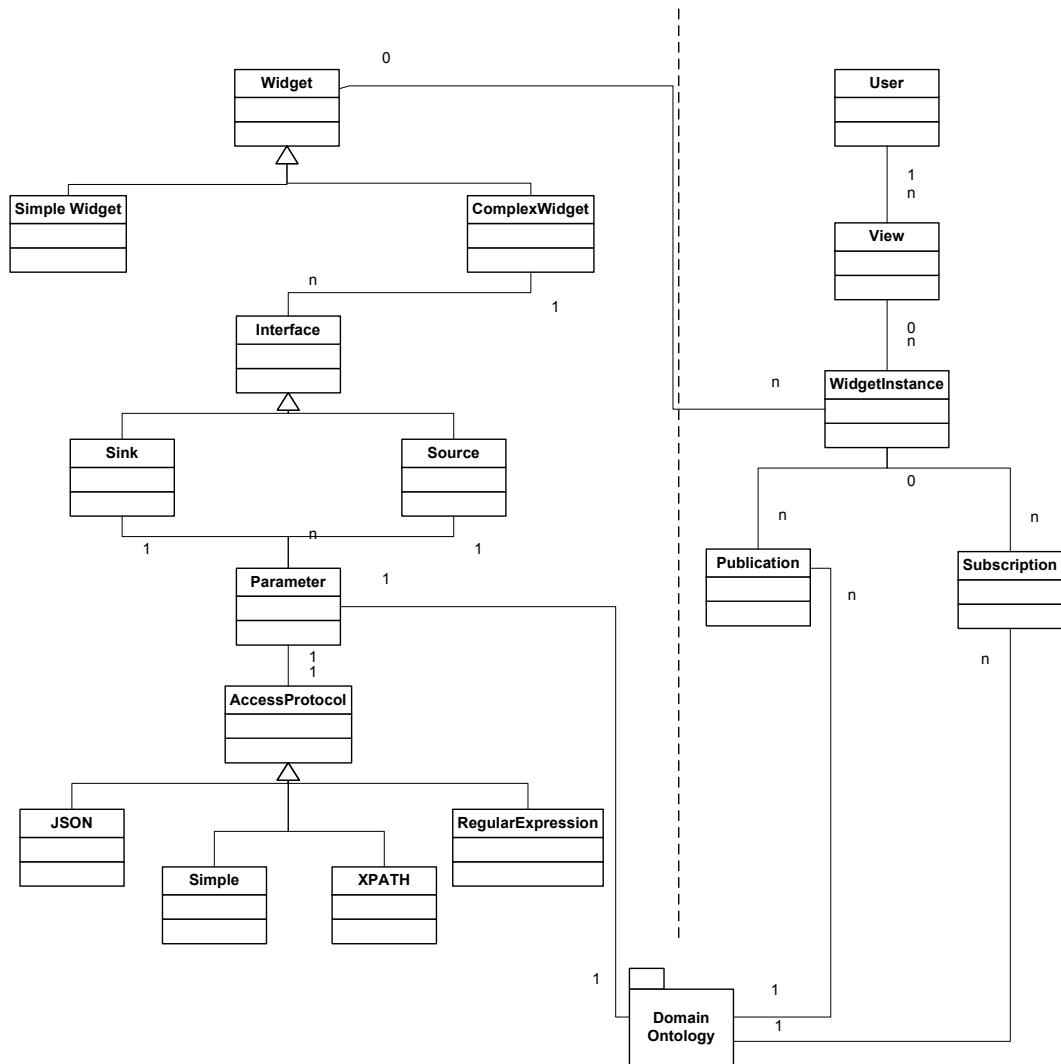
### 3.1 Modeling Widgets

Fig 2 outlines the classes and concepts necessary as part of a model which represents widgets and their use by individual users.

We can see the widget as it is developed and made available by the widget developer: Simple widgets are those widgets which do not require interaction with any other widgets or their views. Complex widgets are those that have the ability to act as event sources or event sinks ("Source"/"Sink" respectively)

Events for Sources and Sinks can contain more than one piece of information, for example, a message might contain a temperature, and the time at which it was taken.

Given that many parameters can be contained in a single message, there must be a means to extract those parameters. The data in the message payload may be formatted as JSON or XML. It might be accessed using a regular expression, or indeed, it might be a simple single parameter. Some flexibility is required here, so this part of the overall model is based on the ClassFactory pattern[15]. In the model above we are allowing for multiple styles to be used in the one message – this could mean that an XML document could be followed by a JSON object with multiple named properties.



**Fig. 2** Model for Developed Widgets

**Fig. 3** Model for Widgets in Use

Envoy needs a mechanism for decoupling the widgets and this can be achieved through the use of a message bus. In [16] it is noted that the shared infrastructure between the message bus and the sources and sinks can be achieved using a message router or through a publish/subscribe system. A publish/subscribe based system with event processing services provides maximal decoupling between widgets and their hosting container. The static model required by Envoy to enable the instantiation of widgets by a user is shown in Fig 3. To give this context we have included the User and View class. Any user can have multiple views. Each view can have one or more widgets in it, with various user settings – **WidgetInstance**. **WidgetInstance** is linked to the base widget installed in Envoy by the widget developer and has zero or more publications and subscriptions. These publications and subscriptions are identified through the domain ontology specified by the widget developer during the registration process. This separation

of Sink and Generator from Subscriptions and Publications is required as an individual user may not want to publish a particular message, or may wish to add specific constraints to a publication or subscription. For example, if a widget represents a sensor and it acts as a Source for heart reading, subsequent Sinks may only accept that output if the heart rate as measured by the sensor exceeds 110 beats per minute.

This model is typical of publish/subscribe systems[17, 18] which focus on subject and predicate filters on single simple events, such as the example given above. In [19] it is noted that more complex publish/subscribe systems are required, especially in peer-to-peer systems on which the widget model of operation is modeled. The publish/subscribe model described in Fig. 3 will, however, have difficulty processing complex events where two or more events combined may result in an exception. For example, a heart rate sensor reading 110 beats per minute may not need to be acted upon, but if it is combined with another sensor reading which indicates impending heart failure then action will need to be taken. These sensors and widgets may be provided by different suppliers and thus cannot be modeled effectively until the user combines them in a single view. This implies that a model must exist independently of the widgets which reflects the user/view and is capable of reacting when a combination of events occur within a time window.

At the bottom of the model is a package called Domain Ontology and exists independently of the widgets (though the widgets event generators and sinks are mapped to the model). It is included here as an abstract concept to demonstrate the way in which widgets are grounded, how semantic mediation is achieved, and how simple and complex event processing can be modeled. The domain ontology will be different in each view created by the user, given that a view in turn will be an amalgamation of aligned ontologies specified by community of web widget developers. The relationships noted in the diagram refer to individual concepts and attributes within specific ontologies. More detail on the domain ontology package is included below.

### 3.2 The Domain/View Ontology

The purpose of the domain/view ontology is to provide a shared understanding of the concepts used in the view, by a specific user. This will enable more flexibility, robustness, and scalability in the following user and widget developer activities:

1. Describing widgets. Developers will describe their widgets in terms of the ontologies already in existence. Common concepts and keywords will be provided to enable others find the widgets. On a lower level the interaction requirements of the widgets will be described and related to concepts and attributes within the defined ontologies.
2. Discovering widgets. Using concepts and keywords from the defined ontologies will give users and developers alike a very efficient mechanism of testing and building customized views.
3. Automated application/view build. A user will have a profile and will be able to specify requirements which will drive the widget search process.
4. Allowing the decoupling of widgets. The domain/view ontology will provide a point of integration of the widgets, without individual widgets having to integrate with one another.
5. Enabling context for widgets. By providing an ontology which reflects both the structure of a view and the user profile we can add more intelligence to event processing
6. The domain/view ontology provides the control point for complex event processing. Through the use of the ontologies the user/widget developer can now provide a means to define complex events and how they should be dealt with, especially when those events could originate from multiple decoupled widgets.
7. Knowledge encapsulation. Adaptive views can be created by adding and removing widgets depending on information flowing between them, altering the look and feel of widgets so that the user's attention is drawn to those parts of the view most relevant given the current information flow.

The domain/view ontology will be composed of a number of modularized ontologies representing the various interests of the user within the current view. A user may decide to create thematic views in which case the domain ontologies will tend towards a single representation. For example, a user may chose "My

Health" as a theme. In this case we could see a patient ontology to model the user, a Friend of a Friend ontology to represent those involved in patient care, along with a drug ontology which will help the user and his caregivers establish the safest drug regime for treating his illness. For another user, who may wish a more integrated view of his life, he may choose to include widgets which relate to his family – photographs, blogs, his hobbies, the weather, sports results, and so on. His choices will influence the domain ontologies which will be aligned to represent his view of the world.

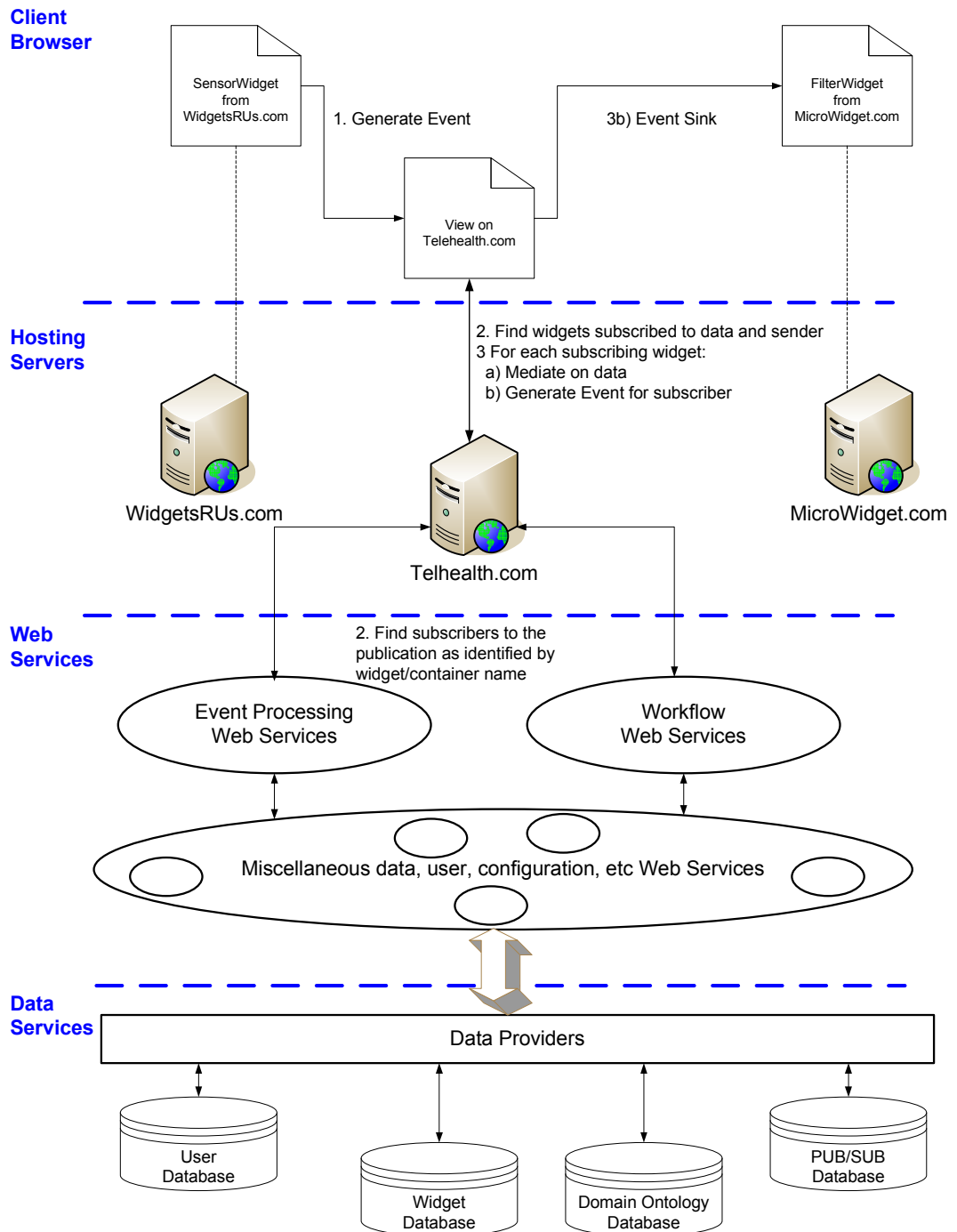
## 4 Run-time Architecture

### 4.1 Overview

This section describes an architecture which will support the models and workflows required for Envoy. Fig 4 illustrates the architecture.

To describe the architecture we have included a number of textual descriptions in the diagram which illustrate the flow of information and the interaction of the various high-level elements identified.

**Step 1.** On the top left of the picture SensorWidget raises an event to send sensor information to the hosting view on Telehealth.com. The only requirement specified by HTML 5 is that the data "payload" is a string or binary data, the content of which is freely structured. We suggest that it be left to the widget developer to choose the way in which he defines and implements the semantics of the event and the mechanism he uses to identify multiple events. During the widget registration process to be described in a later section, the developer will describe the widget inputs and outputs and the mechanisms to be used in accessing the payloads.



**Fig. 4** Envoy Architecture

**Step 2.** The hosting view is listening for events from the widgets. The event data arrives in a free structure requiring an understanding of how to unpack the data. The hosting view delegates all event data processing to web services. In our example the hosting view establishes those widgets interested in outputs from SensorWidget. It interacts with a publication/subscription web service. This publication/subscription web service will be aware of the widgets installed in a view, the outputs (if any) from each widget, and the required inputs (if any) for

each widget. Each widget will be identified by a unique identifier, and each input and output will also have unique identification.

**Step 3.** The hosting view, having received a list of widgets interested in the output of SensorWidget, will interact with a Workflow web service. The workflow web service will determine the flow of events and will work with the Event Processing Web Service and other services such as a Mediator Web Service. The mediator web service will, through models of the view, the widgets and their interfaces, map the outputs of SensorWidget with the specified input requirements from FilterWidget.

Data will be stored in one or more data stores. The data stores shown are logical in nature. The physical data store may blend all of these into one data store. Types of data required to be stored and accessed include user, widget, ontologies, and publication and subscription.

Envoy is based on an Event Driven Architecture (EDA). We chose EDA as it provides some key, desirable attributes when enabling inter widget communication:

1. Loose coupling. Widgets are loosely coupled as they are developed by independent organizations.
2. Responsiveness. With the nature of a widget base application comes uncertainty in the way that they will be combined. Through the use of EDA we can allow for this uncertainty in the event processing agents.
3. Asynchronous domain. Widgets will raise events randomly, and not wait for a response. Such is not typical of other architectural patterns such as a pure Service Oriented Architecture which relies more on a predefined command-and-control style.

In an EDA there are typically six basic components[20-22]:

1. Sensors or Sources. External events coming into the system.
2. Event Processing Agent (EPA). Events flow through the EPA which is also known as a matching engine. It takes streams of events and

potentially runs simulation and queries to determine if significant events have taken place.

3. Sink. Also known as responders, these are services which respond to the significant events within the system
4. Channel. This is a service which disseminates message throughout the system.
5. Administration Services. These services will manage the components of the system.
6. Transformation Services. Also known as brokers, these are services which consume, process, and create events.

The following table indicates how the core components of a typical EDA map to Envoy:

Table 1 EDA Concepts as implemented in Envoy

EDA Concepts	Envoy Components
Sensors or Sources	Widgets
Event Processing Agent	Event Processing Web Services
Sink	Widgets
Channel	The View
Administration Services	The Registration Processes (Following Sections)
Transformation Services	Mediation Models

In Envoy simple and complex events are sourced by widgets. Those events are processed by an Event Processing Web Service. This web service will be based on elements of Complex Event Processing where combinations of events, the timing of events, more complex events can be dealt with than traditional publish/subscribe systems where single events are treated as important, rather than a sequence of events. Widgets will also act as sinks in that they will receive events and will act upon them within the integrated view. The view itself will act at the entry point to the channel, where events are distributed, using HTML 5 constructs, to sinks. Widget registration and deployment services will enable the administration of widgets by both widget developers and end users. Events from widgets will be heterogeneous and in most cases will require mediation (transformation), both on the structure and on the semantics of the event payload.

Events received by sinks will not necessarily contain the same payload as sent by the source.

Using Envoy widgets can be developed independently of each other and without reference to the hosting viewer. They can be updated independently, and they can be used in different contexts.

## **4.2 Web Services**

For simplicity only two high-level web services are shown – the Event Processing and Workflow web services. These are provided as entry points to Envoy. Other web services are identified through the descriptions of the widget registration process and the widget deployment process.

### *4.2.1 Event Processing*

The Event Processing Service (EPS) is first of two core web services that enable the decoupling of widgets from one another. The EPS allows each widget act independently to determine if and when it should receive updates as the data underlying the domain-based ontology is updated by other widgets. It will also allow the publishers specify exactly what is to be published and what that publication means. This service will take input from the other services such as the Ontology Management Service and the Ontology Alignment Service.

The EPS will also allow for more complex event processing where the events from two or more widgets can be used to establish if the combined event is of interest to other widgets. This will be based on temporal, spatial, and relationships between those events. A semantic model will provide meaning to these events and their relationships to allow for a declarative mechanism combined with reasoning over the streams of event data coming from the various widgets in the view.

### *4.2.2 Workflow*

The workflow service will work with the event processing service to determine the sink to which the current events should be forwarded. In the majority of cases this will be a simple one to one relationship. However, in conjunction with the

Event Processing service it may be that the workflow will resemble a many to one where many events are combined, or where there is a temporal relationship between events incoming to a sink.

#### *4.2.3 Mediation*

A fallout from heterogeneous widgets deployed in a single view is the requirement for mediation between the events and their payloads as they pass from Source to Sink. The mediation web service gives Envoy the flexibility to accept heterogeneous widget registrations from independent software vendors and service providers. In cooperation with the event processing service it enables the full decoupling of the widgets in Envoy.

#### *4.2.4 Ontology Alignment*

The purpose of the Ontology Alignment Service will be to establish the fit of the widget inputs and outputs with the ontology representing the view the user has created in Envoy.

Over time the widget developer community will work with the user community to develop domain ontologies which reflect their needs. By necessity these ontologies will be modular, but will need to be decoupled themselves for ease of maintenance, flexibility, and robustness. Bearing this in mind, users will design views which will use more than one of these ontologies which may overlap in their meaning and context. In this case an alignment of ontologies will be required for specific users to ensure that widgets communicate effectively across the domains represented by the ontologies, based on the shared understanding of concepts between different ontologies as developed by different sets of widget developers and their users.

#### *4.2.5 Ontology Management*

The purpose of the Ontology Management Service is to allow users extend and redefine the domain ontology they are using in their view(s). It could be through of in terms of allowing the user to create on-the-fly sub classes of already existing classes in the domain model, making the semantic meaning more explicit. For

example, the widget may be outputting the ambient temperature in a specific room, while the domain ontology only has a single temperature attribute attached to a person. The user will then decide whether the temperature output by his widget fits into the existing temperature attribute or whether he should extend the domain ontology to allow his widget be used appropriately.

#### *4.2.6 Ontological / Syntactic Mapping*

Ontological / Syntactical bidirectional mapping will be required to enable the users to attach semantic meaning to the widget inputs and outputs.

Initially this functionality will extend to the semi-automatic identification of potential mappings for the widget sources and sinks to the View ontology representing the context in which the widget will operate. For example, the alignment process will establish that an event containing temperature data is in fact a body temperature reading, and thus that event payload can be mapped to a particular concept in a person- or medical patient- related ontology. Over time this alignment process will extend to other more complex identifications to include units of measure calculations, more complex data types, and many-to-many parameter splits.

There will be a significant level of overlap between this service and the Ontology Alignment Service, but we have kept it separate for now to allow for a “separation of concerns” within the architecture

## **5 Registering Widgets**

The widget registration process will create a semantically described entry in the registry for easier discovery and integration by potential users. Given that the widget will be connected to other widgets and views about which the newly-minted widget has no a-priori knowledge a mechanism must be provided by which the developer can describe his widget, and annotate it with metadata for successful integration with other heterogeneous widgets and views.

One of the key components of Envoy is the registry of semantically described widgets. Ontology-based semantic registries have been researched heavily. [23-

26] are some examples of research and implementation to this issue of making semantically described software components (generally web services) available for search and discovery in the process of creating software applications. The registries used for web services in the research carried out to date can be reused for widgets.

Aligned with the architecture and the registry is the process of putting the widgets in the registries. In [27] the authors provide general guiding principles to be used by application designers when developing mobile applications. The methodology they propose entails the combination of workflow techniques with event based models to enable fine grained adaptation of applications. In [28] the authors propose a framework for the rapid integration of presentation components. These are related works, but, none of them define a process to be used by the developer to publish the component in the registry. Indeed these research works have no reference to the semantic interoperability challenges inherent in composing applications from components built by third parties. Also, they refer to generic presentation components with complete, heterogeneous interfaces which do not take advantage of HTML 5 standards used in Envoy.

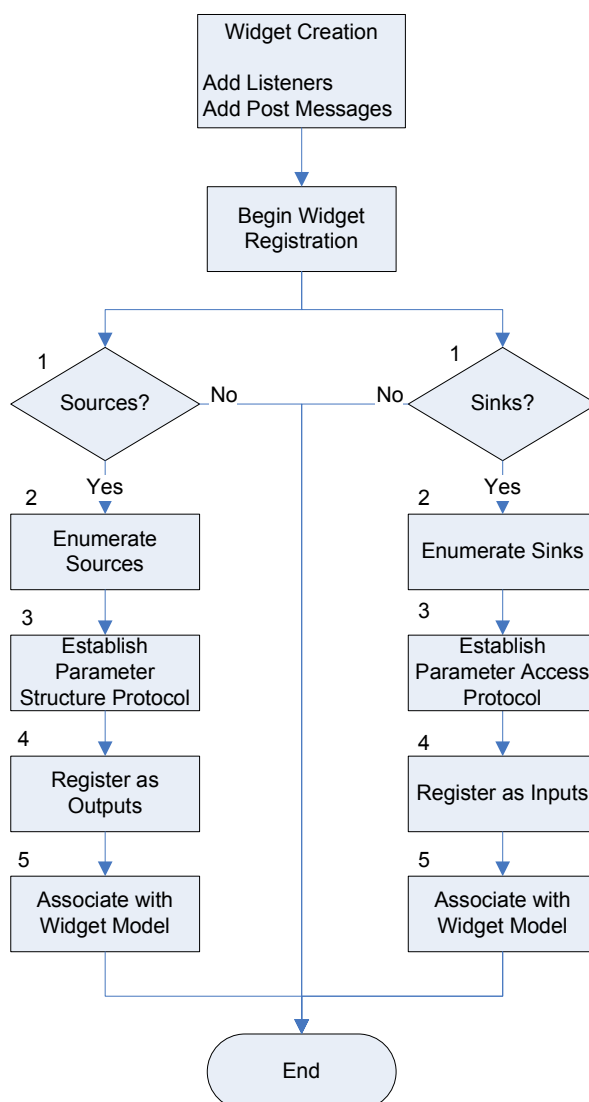
There are two phases to the “Widget Registration” process:

1. The first phase entails the widget creator making the widget available to the wider community, and specifying the input and output parameters and its access mechanisms. This involves an interaction with a low-level ontology which represents the widget independent of the ultimate domain(s) in which it will be used.
2. The second phase entails the widget developer specifying the meaning of the inputs and outputs which form the widget interaction requirements. This interaction is at a higher level and requires the developer to work with domain ontologies. For example, a domain ontology could be a patient and a temperature sensor widget would specify that the temperature gathered relates to the body temperature and not an ambient reading.

## 5.2 Registering a Widget – Phase I

The widget creator has a newly minted widget which will communicate with other widgets, but those widgets are currently unknown. Fig 5 illustrates the process to be used when registering a widget in Envoy described above.

The registration process assumes that there is an already created ontology which describes widgets, their inputs and their outputs.



**Fig. 5** Registering a Widget Phase I

Widgets act as Sources and Sinks as they communicate between themselves and their hosting view. The HTML 5 constructs allows generic, string or binary,

messages to be exchanged between widgets and their views. As with the intelligence and functionality of the widgets, we will see a wide variety in the capabilities of widgets to exchange messages, from the traditional display messages, through XML encoded data types, to semantically described attributes and processes.

**Step 1 & 2.** Enumerate any Sources and Sinks evident in the widget. As noted earlier, any widget can act as multiple Sources and Sinks. If none of these HTML 5 constructs are found in the widget then the widget does not require interaction with other widgets, and the process ends.

**Step 3.** Establish the parameter access protocol. The widget developer will identify how various parameters are structured within the Source and the sequencing of parameters for the Sink. Widgets will vary to the point where one widget could emit an event consisting of a single number representing a temperature, while another could emit an event consisting of a temperature reading, along with a timestamp, all within the same string. The developer will need to specify a method whereby each of those parameters are accessed, be that through a regular expression[29], jQuery[30], or through JSON[31] (JavaScript Object Notation), or indeed through a direct access method where the parameter just contains the value of, in our case, the temperature sensor. The above is particularly relevant when a widget acts as a Source. In the case of a Sink the developer must be in a position to provide a template for its total input, broken into each of the composite input parameters. For example, our widget might act as a Sink for the output of a Source and the associated duration. The developer could specify a template structure which could be a combination of string literals and values similar to the following:

```
"HeartBeat="<AverageHeartRateValue>";Duration="<DurationInMinutes>".
```

**Step 4.** Register the Sources and Sinks and their parameters as outputs and inputs in Envoy. The widget specification will be stored in RDF[32] to allow for flexible storage.

**Step 5.** This step follows on naturally from Step 4 and relates to the mapping of the widget specification to the widget model. The widget developer will store the widget specification in terms of a concrete realization into the Envoy data store. The widget model described in Fig 2 and Fig 3 will act as a shared repository of widgets in Envoy and will allow for significant flexibility when it comes to mediating between widgets on a view and with the view itself.

### 5.3 Registering a Widget – Phase II

Now that the widget developer has created a specification of the widget in terms of a formalized widget model he is now in a position to further specify that widget within the context of a domain. The questions to be addressed in this phase of the widget registration process include: What exactly am I expecting as part of the interaction with my widget? What is the meaning of the data to be exchanged?

The purpose of this phase is to relieve the burden from the user of understanding the specific, technical semantics associated with the widget. The user knows what he wants to do with the widget but will most likely not be in a position to specify it using semantics or ontologies.

Fig 6 illustrates the process to be used during Phase II of the widget registration process – aligning the widget with the available domain ontologies:

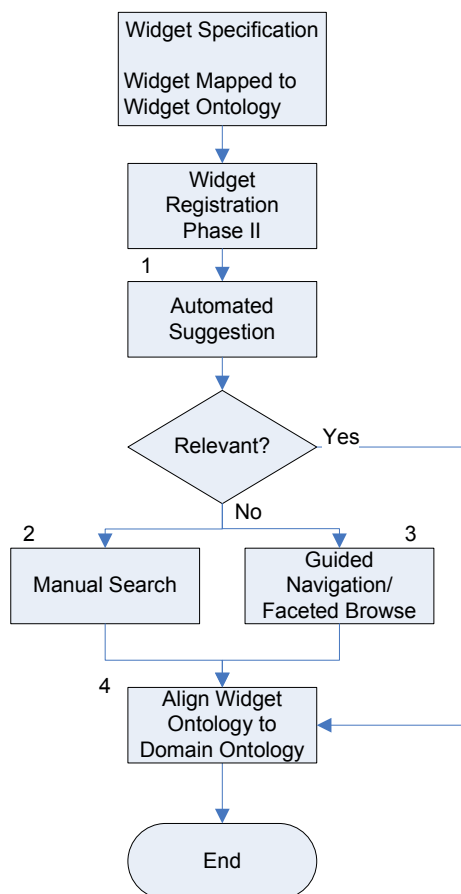
Phase II of the widget registration process picks up where the widget developer completes the specification of the newly minted widget in terms of the widget model. This phase relates to the way in which developers make their communicating widgets easy for casual users to deploy on their web pages.

**Step 1.** The process starts with Envoy using the named parameters of the widget inputs and outputs to find related concepts in the domain ontologies currently stored in Envoy. For example, if the widget developer specified a parameter name such as "Temperature" the search facility will use a combination of keyword search along with NLP to determine a number of suggestions. The suggestions will present so that the widget developer can select the concept and complete the mapping process with minimum effort.

**Step 2.** If the widget developer is unhappy with the suggested list of concepts returned by the named parameter search he then can use other search terms to find the correct concept from the established ontologies.

**Step 3.** Another option is to use the guided navigation feature / faceted browse feature will enable the developer browse the library of ontologies to select the ontology and concept most applicable to the widget input and output parameters.

**Step 4.** The final step in the process is where Envoy aligns the parameters with the domain ontologies. This is a combination of grounding the domain ontology and aligning that ontology with the lower level widget model.



**Fig. 6** Registering a Widget Phase II

The widget is now available for use by casual users in Envoy.

## 6 Deploying a Widget

### 6.1 Overview

The key requirement from Envoy is to ensure that the non technical user can create personalized applications dynamically, based on widgets published in the semantically enhanced registry. While research has been focused on component registries[23-26] and the techniques used to model components and their interactions[33-35] in various contexts, little has been focused on the processes a non technical user would employ to create personalized applications using these components.

As with the widget registration process used by the widget developer, we have specified a two-phased process for deploying the widget.

1. The first phase entails the user adding the widget to his view in Envoy. It involves simple addition of the widget and verification that the new widget performs as expected by the user.
2. The second phase is optional and will be executed by the more technical of users. It entails specifying more complex relationships between the widgets than would be assumed by the interfaces they expose as Sources and Sinks. This phase may also be entered by those who wish to use generic utility widgets in a domain-specific view. The user may wish to specify how that interaction should happen in their view, which may be different than that envisaged by the community creating the domain models behind the view, or indeed the potential usage foreseen by the widget developers.

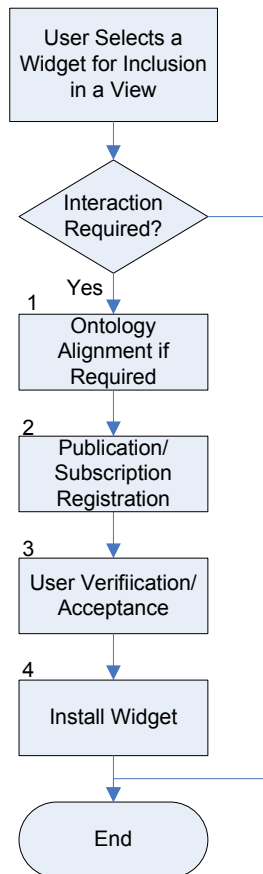
### 6.2 Deploying a Widget – Phase I

In Envoy each view will have a “domain” ontology which maps back to the Widget Ontology used during the registration of individual widgets. This Domain or View ontology could be a representation of a person or a patient, or indeed could reflect the interests a user might have and would include sports, news, weather, and other miscellaneous pieces of descriptive information.

When the user decides to install a widget into his view he will understand it in terms of his own requirements – and each user’s requirements will be different. However, a starting point is required for a user to start creating a View within Envoy. This starting point will be a series of templates or typical profiles from which the user will choose. These profiles could refer to things like health, diabetes, sport, news, and so on – templates created and made available by other users. The user would choose one which is the closest match and then modify it implicitly through the addition/removal of widgets.

The diagram shown below illustrates the process to be used when adding a adding a widget to a view.

Once it is established that the widget has interaction requirements (extracted from the specifications noted in the Widget ontology) the process of integrating the widget into the current view begins.



**Fig. 7** Deploying a Widget Phase I

**Step 1.** The widget developer will have already mapped the widget with both the widget model and a domain ontology. This step evaluates the existing ontology in use representing the view and establishes the fit of that ontology with the arriving ontology representing the new widget. In some cases it may be that the existing view ontology subsumes the ontology referenced by the incoming widget in which case the Ontology Alignment step is a simple one. In the case where they are different then Envoy refers back to the core ontologies created and establishes the linkage and alignment points of the two ontologies. The new ontology is brought into the view/domain ontology.

**Step 2.** The widgets already in the view will have existing publications and subscriptions. The arriving widget inputs and outputs as defined through the widget and domain ontologies will register with this service. The service will provide an entry point for mediation based on the aligned domain ontologies in use in the domain.

**Step 3.** Once the widget has been provisionally inserted into the view the user will verify that it doesn't break anything already working in the view. A signoff process will help here allowing the user to agree that the new widget is performing as expected and fits into the view in terms of its functionality, inputs, and outputs.

**Step 4.** At this point the user accepts the widget and it is released from its sandbox to the main view.

### **6.3 Deploying a Widget – Phase II**

There are times that the default deployment as offered by the widgets and the domain model do not satisfy a specific user requirement. In those cases where more complex interactions between widgets are required, an additional phase of deployment is required. It is envisaged that Phase II of the deployment process can be initiated at any time by the user.

Instances where this phase of deployment are used include:

1. The user is deploying a previously unused generic utility widget into the view.
2. The user wants to customize the way that a template view operates.
3. The user wants to combine events in specific temporal sequences so that a more complex event process can occur.
4. The user wants to make unobvious transformations of event payload data as the event passes from one widget to another.

Implicit in these examples are requirements for elements of workflow creation and complex event processing, and it is from these two fields of research that we draw upon the knowledge required for the deploying widgets (Phase II) in Envoy.

Implicit in the examples given above are the following impacts:

1. Events will be held/delayed in some cases where two or more events must fire before being sent to a Sink. The first event will be held, updating the data store, but when the second event comes through the event can fire.
2. Envoy must maintain the state of the view so that events can be split or joined as defined by the user.

In Envoy there is no direct interaction between the widgets. The activities are directed towards modeling the interaction – the implementation is through the use of publications and subscriptions, and event based integration.

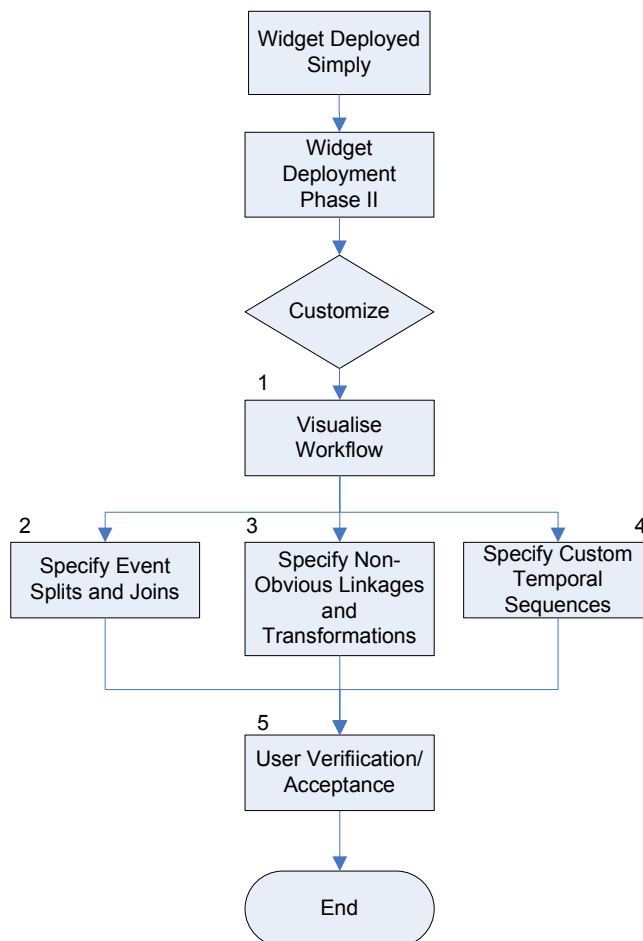
The figure below indicates the process a user will employ in the second phase of widget deployment on a view.

When a user decides to customize the view they have just created he is presented with a visualization of the workflow:

**Step1.** To generate a workflow visualization from a widget-based view, the workflow engine will analyze the publications and subscriptions associated with each of the widgets in the view, along with any customizations made to date by the user. It must then make the connections between the widgets based on those

publications and subscriptions much as it would do at run time, except that no data will be moving at this point.

**Step 2.** Events can be split or joined, depending on the Source and Sink being connected. For example, a Source might be modeled as sending its payload to two Sinks. This can be an 'AND' (both Sinks accept), an 'XOR' (only one Sink accepts), or an 'OR' (any one or both Sinks accept). Likewise, a widget might act as a Sink to two or more Sources. As for a split, a join could be an 'AND' (both Sources raise events), an 'XOR' (only one Source raises an event), or an 'OR' (any one or both Sources raise an event).



**Fig. 8** Deploying a Widget Phase II

**Step 3.** Linkages between widgets not picked up automatically through the first phase of the deployment process can be specified. This would normally occur if a generic utility widget was dropped into the view. This utility widget could be a

communications widget or a newsfeed-based widget which is not domain specific. In the context of a specific view these types of widget will be interested in the occurrence of certain types of events, but not in the data payload contained in them. When an event is raised through the system this widget could need some demographic data, or some element of the users profile other than the specific data changed as part of the event processing functionality.

**Step 4.** Single events happening in one temporal sequence may not be of interest to a sink. However, when the temporal sequence changes in those same events the sink may need to be notified with a data payload. These custom temporal sequences will not be automatically implemented as part of the first phase of widget deployment in a view.

**Step 5.** Once the customizations have been made to the view the user will verify that the application performs as expected. A signoff process will help here allowing the user to agree that new custom application is performing as expected and meets the user's requirements.

The storage and representation of this interaction will be given a home at the level of the view. This will allow the context of the view to be built and defined semantically which will allow more reasoning about the view and the data being exchanged by the widgets. This implies the development of a model of the conceptual interaction possible between widgets. The instance data then represents the individual views.

## **7 Related Work**

### **7.1 Event-Based Software Architecture**

Research into mechanisms enabling the reuse of components within the context of event-based, or messaging, systems finds its roots in the latter half of the 1990's. It was applied to graphical user interfaces[36] and event-based integration (EBI)[37].

In the 1990's widgets were considered to be the buttons, input boxes, labels, etc that one would see on a dialog box, or similar UI component. In the case of event based architectural styles for GUI software, the objective was to support larger granularity reuse of software components. Interoperability between components is enabled through the use of domain translators, where each component would have a mechanism for explicitly translating incoming events into a representation that the component could use and understand. This would enable the maximum reuse of a software component. Connectors are employed to route, broadcast, and filter messages.

However, the domain translators limit the reuse of the components. Since, for complete semantic interoperability, each component will have to transform incoming messages into a form it understands, which assumes a-prior knowledge of the incoming formats, at the time of component creation. Compounding the problem here is that human intervention is most likely required to enable interoperability of components.

With the addition of Envoy as a semantic widget engine, interoperability can be abstracted from the widget (component) to a domain model. This model is then mapped to by each widget developer. Mediation is enabled through the use of a "third party" which understands semantically the domain in which the widget is to be deployed, and publish/subscribe web services built on top of the semantic model ensure the events get routed to the right receiving widgets.

In [37] the authors put forward a framework for Event-Based Integration (EBI). In that framework informers and listeners are registered with a registrar. Once messages are exchanged then a series of predefined Message Transformation Functions (MTFs) are used to mediate upon the messages so that the listeners can understand the structure and the meaning of the message. Delivery constraints act as filters on the messages and can define such things as the ordering of messages, the priority of messages, etc.

Envoy conforms to such a framework, where registration of widgets is performed as a two-phased process (to the generic widget model and then to the domain). In

Envoy the global domain ontology mediates upon the messages through mappings provided by widget developers. Semantic mediation carried out through modeling the domain equates to the MTFs described as part of the EBI framework. The publication/subscription services act in a similar manner to the delivery constraints from EBI where a subscriber could constrain the receipt of messages to specific times of the day, or only when some other event has also happened, and so on.

## **7.2 Mashups**

A mashup is defined as “a Web site or application that combines content from more than one source into an integrated experience.”[38] In [39]the authors apply semantics to web pipes to build RDF-based mashups.

Mashups rely on the users understanding the data sources, even if those data sources are presented in a “standardized” fashion. Thus the inputs to a mashup and their output through a widget are tightly coupled. Mashups are good for ad-hoc widget development and are typically shared amongst small, specialized user groups. They focus on content, not workflow.

Mashups are not to be confused with widgets, but they can be used together. For example, a widget can be used to display the output of a mashup. And the output of that widget can be used in Envoy.

## **7.3 Semantic Service Oriented Architecture**

Standards such as WSDL have been applied to the definition and description of web services. Widgets have no such grounded definition. Widget interaction models are more light weight and just rely on inputs and outputs.

Similarly WSMO[40] and OWL-S[41] are the results of efforts into the semantic definition of web services. These ontologically base languages allow the modeling of web services so that their function and meaning is explicit and provide a mechanism where machine-to-machine interaction is possible, even if the web service interaction requirements are heterogeneous.

In the Web services environment decoupling is achieved through the use of such tools as WSMX[42] and METEOR-S[43]. Through the creation of ontologies representing service descriptions within a domain, semantic mediation of both the data and the process requirements of the various web services involved in a workflow to complete a business transaction ensures that the web services can evolve independently with minimal impact on the overall transaction processing system.

In Envoy we use a similar concept for widgets rather than for web services. We have described a domain ontology representing the view and the user's profile. This, matched with an event processing framework, provides decoupling between the widgets, and enables their independent evolution.

#### **7.4 Workflow**

Workflow has been researched over the decades and has found implementation in a variety of products[44-46] and standards[47, 48].

These tools evolved over the years to become complex, all-encompassing environments and have mainly focused on the needs of professional software developers and business analysts. In Envoy we will take a simplified view of workflow, make it available on the web for users with limited software development experience. We will focus on allowing the customization of connections between widgets allowing XOR, AND, and OR splits and joins so that multiple Source Widgets can feed into a single Sink Widget and that one Source Widget can feed into multiple Sink Widgets.

Starting from this base, we will integrate the workflow engine with the event-based integration paradigm which forms a core component of Envoy.

#### **7.5 Widget engines**

Existing widget engines will not be in a position to handle intercommunicating engines in a scalable, flexible manner as in those engines the widgets will have to be tightly coupled to one another and to the view which hosts them.

The following table lists the high-level functionality a widget engine will need to enable applications to be built using heterogeneous widgets developed by independent software vendors:

Table 2 Functionality Map to Widget Engines

Functionality	NetVibes	iGoogle	PageFlakes	Spring Widgets	Widget Box	Envoy
1. Custom Home Page Creation	X	X	X		X	X
2. Widget Library	X	X	X	X	X	X
3. Syntactic Widget Search	X	X	X	X	X	X
4. Categorized Browsing	X	X	X	X	X	X
5. Themes	X	X	X			X
6. Sharing Content With Others	X	X	X			X
7. Home Page on your Mobile	X					X
8. DIY Widgets	X	X	X	X		X
9. Tabs or Views	X	X	X			X
10. Public Views / Pagecasting	X		X			X
11. Private Views	X	X	X		X	X
12. Export Widgets to other pages				X	X	X
13. Import Widgets	X	X	X	X	X	X
14. Decoupled Widget Connection						X
15. HTML 5 Compliance						X
16. Event Processing						X
17. Widget-based Application Design						X
18. Workflow						X
19. Semantic Widget Search/Discovery						X
20. Semantic Interoperability						X

Functionality implemented by widget engines in general are shown above. In addition we have included some high level items of functionality required by any widget engine purporting to support application development based on web widgets.

The type of functionality supported by existing widget engines limits the usefulness of widgets to standalone entities, unaware of anything else with which they share screen real estate. This is indicated by functionality items 1 to 13 and existing web engines and libraries, in general, support enough functionality to allow users create home pages (1), enable storage, search, and browsing for widgets (2,3, 4), apply color and layout themes to pages created (5), share content with others (6), use a mobile phone to access your page (7), create widgets (8), create tabbed views within the page (9), make your page public for others to use (10), or chose to keep it private (11). They also allow to varying degrees the export of widgets for use in other environments and web sites (12), along with the ability to import widgets (13). Only one engine reviewed supported the use of mobile phones as a delivery mechanism.

Functionality items 14 to 20 are required at a minimum to enable widget-based application development. The widgets must be able to communicate even though they are decoupled from one another and from the view hosting them (14). Based on the fact that the HTML 5 constructs are event based and thus allow event based integration there needs to be functionality which will process these events, both in the simple case, and in the more complex cases where temporal, spatial, and sequencing interaction is required (15,16). There must be functionality whereby users can design their applications based on widgets (17). Users will also need to visualize, enact, and manage workflows as part of the application design process and then in the running of the application (18). Users need to be able to discover web required web widgets (19). Semantic interoperability is required to make the widget interaction as robust as possible (20).

## 8 Conclusions

Creating and using applications on the web has typically involved interacting with monolithic applications, where one size fits all. Widgets provide an ideal mechanism for encapsulating functionality but to date they have been limited to the satisfaction of user requirements such as clocks, RSS feed readers and news aggregators. Widget engines have provided a platform for these mute widgets through simple display mechanisms, but have not provided an infrastructure which allows customized applications to be built. Envoy enables integrated widget based applications where the user can build his own customized application.

With the release of HTML 5 as a working draft inter-widget communication has been enabled. Using HTML 5 constructs, widgets can act as Sources for data within an event-based application, and they can also act as Sinks for data. This has opened the door to more meaningful widget-based application development. Firefox 3, Internet Explorer 8, and Opera 9.5 browsers support the standard. Thus, widgets employing the standard HTML 5 constructs will find support on all major web browsing platforms.

However, to use this messaging infrastructure in a scalable, flexible manner within a heterogeneous widget environment presents a significant engineering challenge. Key structural questions to be addressed include:

- How do we build communicating widgets independently?
- How can heterogeneity between widgets be handled?
- How can views host intercommunicating widgets forming part of a workflow, without having a-priori knowledge of the widgets or the workflow?

Adding a publish/subscribe infrastructure provides a means for decoupling the widgets – providing that participating widgets understand the structure and meaning of the messages to be exchanged, without knowing one another.

Complete decoupling is achieved through a widget model and a domain ontology. Widgets acting as Sources pass data using HTML 5 event constructs through an instantiation of the domain ontology. Updates to the data based on this ontology are interpreted by a publish/subscribe system for complex events to determine the widget Sinks that should receive the data. Neither Source nor Sink knows about each other, and the view has no a-priori knowledge of any of the widgets it hosts. All of this knowledge is encapsulated and held through the models described above, and reference to it is maintained through a series of web services.

The widget model defines the concrete mechanisms to be used to access the information elements within a specific message type sent and expected to be received by the various widgets. A domain model is used as a central point of mediation between widgets. The domain model is a series of lightweight ontologies up to which the specific parameters within messages are mapped. The combination of the widget model and the domain ontology provides a means to get from the conceptual description of widgets, their roles, and interfaces to the concrete realization of the widget.

With that infrastructure in place the widget developer will be given the means to register widgets with Envoy. The developer process is two-phased. The first phase involves understanding how to access the core content of the messages – effectively the message structure – and how the Sinks expect to receive data. During the second phase the widget developer specifies the semantic meaning of the information contained in the message or expected by a Sink. A side effect of the second phase of the process is that, through the identification and semantic specification of the data in the messages to be sent and received, the developer suggests the context in which that widget should be used. It also enables the publication/subscribe service work effectively across the widgets without concern about the structure of the exchanged messages. This in turn eases the process for a user to create a view based on these registered widgets.

The process used by to deploy widgets into individual views is also two-phased. In the first phase of the process the widget end-user adds the widget to the view and verifies that the widget works as expected in the view. The second phase is

optional and will be executed by the more technical of users. It entails specifying more complex relationships between the widgets than would be assumed by the interfaces they expose as Sources and Sinks. This phase may also be entered by those who wish to use generic utility widgets in a domain-specific view. The user may wish to specify how that interaction should happen in their view, which may be different than that envisaged by the community creating the domain models behind the view, or indeed the potential usage foreseen by the widget developers. This deployment process also integrates with the domain model/ontology. Through this integration the user knows that the meaning attached to Sources and Sinks is as he expects, and through the use of semantics can build applications in context.

With Envoy interconnected applications can be built by independent widget developers, without a-priori knowledge of the views in which they will be deployed or of the other widgets with which they will communicate.

## Acknowledgements

The work presented in this paper has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

## References

1. The COM: Component Object Model Technologies Website. <http://www.microsoft.com/com/default.aspx>. Accessed 09-November-2008
2. The CORBA/IIOP Specifications Website. [http://www.omg.org/technology/documents/corba\\_spec\\_catalog.htm#ccm](http://www.omg.org/technology/documents/corba_spec_catalog.htm#ccm). Accessed 09-November-2008
3. Fielding, R., Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine (2000)
4. Nenad, M., On the role of middleware in architecture-based software development, in Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM: Ischia, Italy (2002)
5. The iGoogle Website. <http://www.google.com/ig>. Accessed 01-October-2008
6. The PageFlakes Website. <http://www.pageflakes.com/>. Accessed 01-October-2008
7. The netvibes Website. <http://www.netvibes.com/>. Accessed 01-October-2008
8. The Widgetbox Website. <http://www.widgetbox.com/>. Accessed 01-October-2008

9. The SpringWidgets Website. <http://www.springwidgets.com/>. Accessed 01-October-2008
10. HTML 5. A Vocabulary and associated APIs for HTML and XHTML. <http://www.w3.org/html/wg/html5/>. Accessed 01-October-2008
11. Fischer, G., Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments. *Automated Software Engineering*, 54. 447 (1998)
12. Raman Ramsin and R.F. Paige, Process-centered review of object oriented software development methodologies. *ACM Comput. Surv.* 401. 1-89 (2008)
13. Jonathan Ostwald, et al. Organic Perspectives of Knowledge Management. in I-KNOW'03 Workshop on (Virtual) Communities of Practice within Modern Organizations. 2003. Graz, Austria.
14. The HL7 Website. <http://www.hl7.org>. Accessed 18-November-2008
15. Gamma E, et al., *Design patterns: elements of reusable object-oriented software* Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1995)
16. Gregor Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* Addison-Wesley Longman Publishing Co., Inc. 480. (2003)
17. A. Carzaniga and A.L. Wolf. Forwarding in a content-based network. in *In SIGCOMM*. 2003.
18. Zbigniew Jerzak and C. Fetzer, Bloom filter based routing for content-based publish/subscribe, in *Proceedings of the second international conference on Distributed event-based systems*, ACM: Rome, Italy (2008)
19. Patrick Th Eugster, et al., The many faces of publish/subscribe. *ACM Comput. Surv.* 352. 114-131 (2003)
20. K. Mani Chandy and D. Gawlick, Event processing using database technology, in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ACM: Beijing, China (2007)
21. Michelson, B.M., *Event-Driven Architecture Overview*, Patricia Seybold Group (2006)
22. Luckham, D.C., *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* Addison-Wesley Longman Publishing Co., Inc. 376. (2001)
23. Dan Song, et al., *Ontology Application in Software Component Registry to Achieve Semantic Interoperability*, in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, IEEE Computer Society (2005)
24. Bing Li, et al. *Research on Semantic-Based Web Services Registry Federation*. in *GCC'2005*. 2005: Springer.
25. Simon Miles, et al., Towards a protocol for the attachment of metadata to grid service descriptions and its use in semantic discovery. *Sci. Program*. 124. 201-211 (2004)
26. Alessio Bechini, Andrea Tomasi, and J. Viotto, Enabling ontology-based document classification and management in ebXML registries, in *Proceedings of the 2008 ACM symposium on Applied computing*, ACM: Fortaleza, Ceara, Brazil (2008)

27. Tore Fjellheim, et al., A process-based methodology for designing event-based mobile composite applications. *Data Knowl. Eng.* 611. 6-22 (2007)
28. Jin Yu, et al., A framework for rapid integration of presentation components, in *Proceedings of the 16th international conference on World Wide Web*, ACM: Banff, Alberta, Canada (2007)
29. The Regular-Expressions.info Website. <http://www.regular-expressions.info/>. Accessed 10-October-2008
30. The jQuery Website. <http://jquery.com/>. Accessed 02-October-2008
31. The JSON Website. <http://www.json.org/>. Accessed 07-October-2008
32. The Resource Description Framework (RDF) Website. <http://www.w3.org/RDF/>. Accessed 10-October-2008
33. Zoltán Fiala, et al., Design and implementation of component-based adaptive Web presentations, in *Proceedings of the 2004 ACM symposium on Applied computing*, ACM: Nicosia, Cyprus (2004)
34. Behzad Bastani and H. Bastani, High-level open evolvable systems design by process-oriented modeling: application to DNA replication mechanism. *SIGSOFT Softw. Eng. Notes.* 326. 3 (2007)
35. Chris Lüer and D.S. Rosenblum, WREN---an environment for component-based development, in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM: Vienna, Austria (2001)
36. Taylor, R.N., et al., A component- and message-based architectural style for GUI software. *Software Engineering, IEEE Transactions on.* 226. 390 (1996)
37. Daniel, J.B., et al., A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.* 54. 378-421 (1996)
38. The Mashup definition - Wikipedia. [http://en.wikipedia.org/wiki/Enterprise\\_mashups](http://en.wikipedia.org/wiki/Enterprise_mashups). Accessed 13-October-2008
39. Christian Morbidoni, et al. *Previewing Semantic Web Pipes*. in *5th European Semantic Web Conference*. 2008. Tenerife, Canary Islands, Spain: Springer.
40. The WSMO Website. <http://www.wsmo.org/>. Accessed 13-October-2008
41. The OWL-S Website. <http://www.w3.org/Submission/OWL-S/>. Accessed 13-October-2008
42. Armin Haller, et al., WSMX - A Semantic Service-Oriented Architecture, in *Proceedings of the IEEE International Conference on Web Services*, IEEE Computer Society (2005)
43. Abhijit A. Patil, et al., Meteor-s web service annotation framework, in *Proceedings of the 13th international conference on World Wide Web*, ACM: New York, NY, USA (2004)
44. Oracle BPEL Process Manager. <http://www.oracle.com/technology/products/ias/bpel/index.html>. (2008) Accessed 19-November-2008
45. Active Endpoints. *SOA Process Orchestration and Business Process Management*. <http://www.activevos.com/products.php>. (2008) Accessed 19-November-2008

46. Parasoft BPEL Engine & Toolkit: BPEL Maestro.  
<http://www.parasoft.com/jsp/products/home.jsp?product=BPEL>. (2008) Accessed 19-November-2008
47. Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>. (2007) Accessed 19-November-2008
48. Business Process Modeling Notation (BPMN) 1.1. <http://www.omg.org/spec/BPMN/1.1/>. (2008) Accessed 19-November-2008