



**PureWeb<sup>®</sup> STK 4.0**

Developer's Guide

**The information contained herein is proprietary and confidential and cannot be disclosed or duplicated without the prior written consent of Calgary Scientific Inc.**

Copyright © 2013 Calgary Scientific Inc. All rights reserved.

### **About Calgary Scientific**

Calgary Scientific Inc. is dedicated to providing advanced visualization, web enablement, and mobility enhancement solutions to industries looking for secure access and use of their data or graphics intensive applications, while using their existing systems. Visit [www.calgaryscientific.com](http://www.calgaryscientific.com) for more information.

### **Notice**

Although reasonable effort is made to ensure that the information in this document is complete and accurate at the time of release, Calgary Scientific Inc. cannot assume responsibility for any existing errors. Changes and/or corrections to the information contained in this document may be incorporated in future versions.

### **Your Responsibility for Your System's Security**

You are responsible for the security of your system. Product administration to prevent unauthorized use is your responsibility. Your system administrator should read all documents provided with this product to fully understand the features available that reduce your risk of incurring charges for unlicensed use of Calgary Scientific products.

### **Trademarks**

© 2013 Calgary Scientific Inc., ResolutionMD, PureWeb and the Calgary Scientific logo are trademarks and/or registered trademarks of Calgary Scientific Inc. or its subsidiaries. Any third-party company names and products are for identification purposes only and may be trademarks of their respective owners.

### **Released by**

Calgary Scientific Inc. [www.calgaryscientific.com](http://www.calgaryscientific.com).

**Document Version:** PW4.0\_Developers\_Guide\_07-2013\_v1.000.00

# Table of Contents

Chapter 1	<b>Introduction.....</b>	<b>7</b>
	The STK .....	8
	Documentation.....	8
Chapter 2	<b>PureWeb Fundamentals.....</b>	<b>9</b>
	Basic Architecture.....	9
	Service.....	10
	Client.....	10
	Server .....	10
	The Main Building Blocks .....	11
	Views .....	11
	Commands .....	12
	Application State .....	12
	Events.....	13
	Communication Flow .....	13
	How Application State Remains Synchronized.....	14
	How Images in Views Are Kept Up-to-Date.....	14
	How Events and Commands Are Communicated .....	15
	Additional Components .....	15
Chapter 3	<b>PureWeb Enablement in a Nutshell .....</b>	<b>16</b>
	Setting Up the Server Connection .....	16
	Setting Up the Views .....	17
	Adding User Input Events to Views .....	18
	Designing the Client Interface .....	18
	Sending Instructions Using Commands .....	18
	Manipulating Application State .....	19

Chapter 4	<b>Communicating with the Server</b> .....	<b>20</b>
	About the Server.....	20
	Server Communication Workflow .....	20
	Connecting the Service .....	21
	Graceful Disconnect .....	22
	Connecting the Client .....	22
Chapter 5	<b>Views</b> .....	<b>25</b>
	About Views.....	25
	Remoting a Service View .....	26
	The (I)RenderedView Interface.....	26
	Sample Implementation .....	27
	Displaying a Remoted View in a Client.....	29
	Handling User Input.....	29
	Keyboard and Mouse Events.....	29
	Converting Touch-Screen Input to Keyboard Events.....	32
	Defining Image Quality for Views .....	33
Chapter 6	<b>Commands</b> .....	<b>36</b>
	About Commands.....	36
	Setting Up Commands on the Service .....	37
	Registering and Unregistering Command UI Handlers.....	37
	Populating Callback Parameters on the Service .....	37
	Sending Commands from the Client.....	38
Chapter 7	<b>Application State</b> .....	<b>40</b>
	About Application State .....	40
	State Tree .....	41
	Initializing Application State .....	42
	Creating State Initialization Handlers.....	43
	Interacting With The Application State.....	43
	Direct Interaction.....	43
	Advanced Methods .....	44
	Change Handlers.....	45
	Value Change Handlers .....	45
	Child Changed Handlers.....	46

Chapter 8	<b>Designing the Client Interface</b> .....	<b>47</b>
	Feature Set and Appearance .....	47
	Adding a PureWeb Layer to UI Elements.....	48
	Using Commands .....	48
	Using Application State.....	49
Chapter 9	<b>The Resource Manager</b> .....	<b>50</b>
	Managed Access .....	51
	Workflow .....	51
	Example - Screen Captures .....	51
	Service Application Code (Storing Data).....	52
	Client Application Code (Retrieving Data) .....	53
	Resource Distribution Methods .....	54
Chapter 10	<b>Debugging</b> .....	<b>56</b>
	Platform-Specific Debugging .....	56
	Diagnostics Panel.....	57
	Adding a Diagnostics Panel to a Client.....	57
	Using the Diagnostics Panel.....	58
Index	.....	<b>62</b>

# Preface

Welcome to the *PureWeb STK Developer's Guide*, part of the PureWeb® Software Transformation Kit (STK) documentation suite.

## Intended Audience

This document is intended to be read by software developers who plan to install and develop applications using the PureWeb STK.

## Making Comments on This Document

If you especially like or dislike anything about this document, feel free to e-mail your comments to [techpubs@calgaryscientific.com](mailto:techpubs@calgaryscientific.com).

## Contacting Calgary Scientific Support

Use one of the methods in the table below to contact Calgary Scientific support.

Web Site	E-Mail
<a href="http://support.getpureweb.com">support.getpureweb.com</a>	<a href="mailto:support@getpureweb.com">support@getpureweb.com</a>

# 1 Introduction

PureWeb® is a platform that enables the rapid transformation of enterprise software into cloud-ready, web and mobile applications.

PureWeb-enabled applications are centrally hosted on a server and delivered to the end users, from workstations to hand-held devices, using standard web technologies. The application framework is built specifically to leverage HTTP(S) and XML for the highest level of flexibility, interoperability, consistency and performance. This offers several advantages:

- The PureWeb-enabled application and the PureWeb server can be deployed on existing high-performance infrastructure, eliminating the need to invest in new hardware. Alternately, the PureWeb-enabled application and its server can be deployed in a number of cloud-computing environments.
- For users, the client application gives a natural web or mobile experience. It feels like a tablet program on an iPad and like a smart phone program on an Android, but it is really the same software on the server.
- The client is very thin, since the rendering and heavy computation is executed remotely on the server.
- The data remains stored where it is; PureWeb does not move or copy it, and no application data ever persists on the device itself. This keeps the data secure, while still allowing compliant, authenticated access.
- Applications can be accessed through internet browsers capable of running Microsoft Silverlight and Adobe Flash, as well as from many smart phones and tablet devices. PureWeb also offers an HTML5 client that can be used in most HTML5-capable browsers and environments.

The PureWeb STK offers a number of powerful features, including synchronized event-based state management, command-response APIs, tools for collaboration, and a high performance image remoting pipeline. These allow the creation of mobile and web client applications that were traditionally difficult to bring to these platforms, such as those requiring the interactive visualization of large, secure data sets.

Although very powerful, the PureWeb software transformation technology is designed to be easy to master and provides a high degree of abstraction. PureWeb shields developers from the burden of managing the details of image and application state communication, allowing them to focus on creating the best experience for their users.

Re-purposing an existing application using the PureWeb STK (software transformation kit) does not require a costly code rewrite. Rather, developers integrate it directly with their existing code by adding a thin layer between the logic engine and the interface components.

This approach keeps the logic of the application intact. The client and the PureWeb-enabled application tiers are isolated from each other; developers maintain one code base, and can quickly implement clients as needed.

---

## The STK

The PureWeb STK (software transformation kit) provides developers the tools that they need to implement the PureWeb solution in their own applications, in particular:

- Service APIs, to modify existing C#, C++ and Java software to function as a PureWeb service.
- Client APIs, which can be used to create mobile and web versions of new or existing applications. These APIs are currently available in the following languages: Silverlight, Flex, Java, Android, iOS, and HTML5.
- Sample applications for each supported service and client platforms, which illustrate key concepts of PureWeb enablement. They can be used as a starting point for developers to create their own PureWeb-enabled applications.

## Documentation

In addition to the above, the STK includes a complete documentation suite that explains the PureWeb solution, including:

- this *Developer's Guide*, which describes how to PureWeb enable applications using the service and client APIs
- an *Installation Guide*
- a *Server Administration Guide*
- *Quick Start Guides* for each supported programming platform, that illustrate how to quickly get started using the sample applications as models
- Complete API reference material for service and client platform.

For more information about the product, visit [www.getpureweb.com](http://www.getpureweb.com).



Chapter

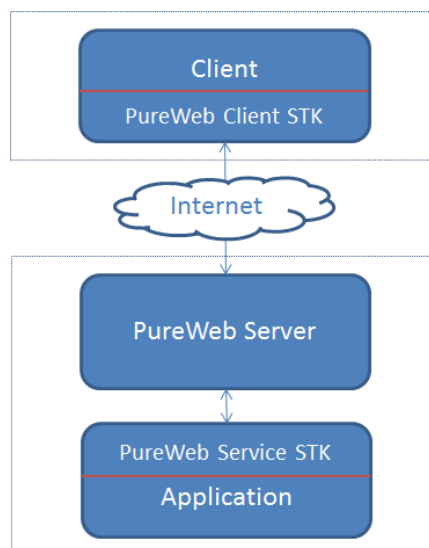
# 2 PureWeb Fundamentals

This chapter describes the basic architecture and main building blocks of PureWeb, as well as the flow of communication between these components. The concepts and terminology described in this chapter will help developers navigate the rest of this document.

---

## Basic Architecture

A PureWeb-enabled system consists of three components: a service application, a client application, and a server. The server and service run on the same system node, and the client runs on remote or mobile user systems.



## Service

A PureWeb service is a C#, C++ or Java application that has been transformed using the PureWeb service STK APIs to connect it to a PureWeb server, making it accessible to PureWeb client applications. It remains responsible for performing all of the application logic – this is what allows the clients in a PureWeb solution to remain so thin. The clients access this application logic through PureWeb “Events”, “Commands” and “Application State”.

The service is also responsible for remoting the rendering of views, which provides client applications the information they need to create user interfaces.

The service also manages the synchronization of the state of the application between itself and the clients applications.

## Client

A PureWeb client is a mobile or web-based representation of a service application.

The first client is usually created in tandem with the service. Once the application is PureWeb-enabled, the STK makes it simple to develop additional clients, with little or no modification to the service.

The STK supports several languages and platforms to develop client applications: Silverlight, iOS, Java Swing, Android, Flex, and HTML5.

The client functionality does not have to be an exact replica of what’s available on the service. It is possible to expose only a subset of features, and to add custom features not available in the original application, assuming the necessary handling logic is added to the service. It is also possible to change the look and feel of each client. For more information, see “Designing the Client Interface” on [page 47](#).

## Server

The PureWeb server leverages existing web server technology (Apache Tomcat) to broker communications between the client and service. It serves as a session manager, fields connections from clients, as well as launches and load-balances service instances. The server manages collaborative sessions, allowing two or more users to view and interact with the same service using independent clients.

The server coordinates client disconnection and server process termination. It also provides a variety of server diagnostic and application information.

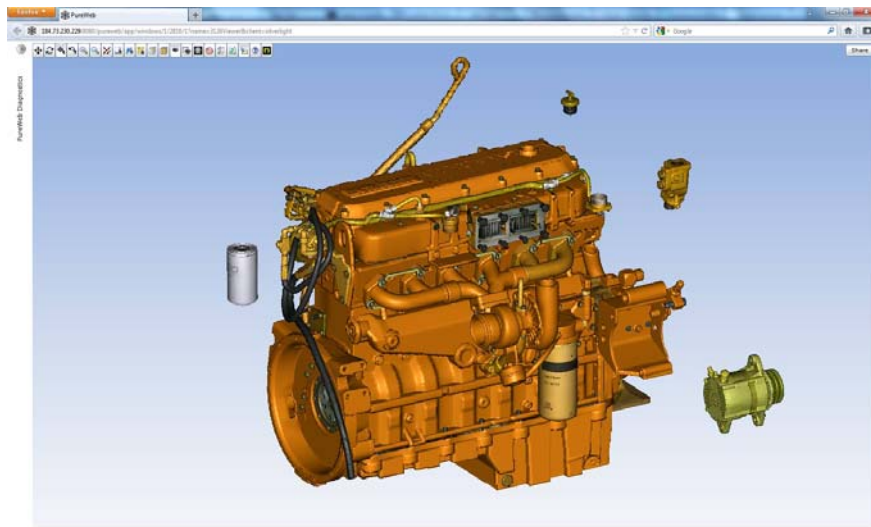
For detailed information about managing and customizing the server, refer to the *PureWeb Server Administration Guide*.

# The Main Building Blocks

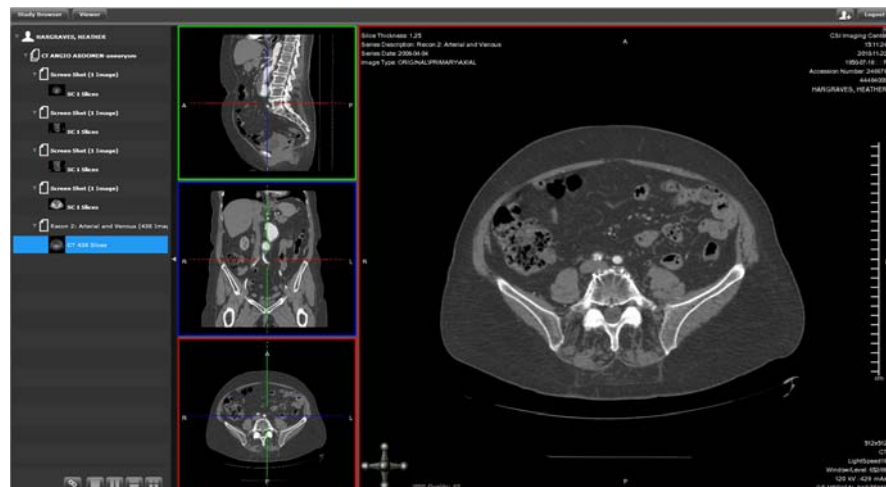
Interactions between client, service and server happens through four main building blocks: views, commands, application state, and events.

## Views

In its most simple form, a PureWeb view is an area on the screen that contains an image. Developers inject mouse and keyboard events into views to make them interactive.



Developers can convert almost any portion of the original application's visible user interface into a view. Additionally, they may have several independent views on the same client screen, each with its own set of mouse and keyboard events.



A common use for PureWeb views is the remoting of 3D rendered images. The service application handles the intensive image rendering, while clients simply display the remoted images within the views. The data used to generate the images remains securely stored on the server.

In the PureWeb STK, the interface used to create views is called either `IRenderedView` or `RenderedView`, depending on the programming language. This interface handles image encoding, quality and rate, sheltering developers from having to worry about image transmission bandwidth and latency.

For more information, see the chapter “Views” on [page 25](#).

## Commands

A command is an instruction sent from the client for the service to execute a given function.

For example, the application could have a “Clear” function that erases the scribbles on a white board panel. To enable this functionality on a client application, add an “Erase All” button which, when clicked, queues a Clear command; the service would then execute the Clear function.

For more information, see the chapter “Commands” on [page 36](#).

## Application State

Application state is essentially a hierarchical set of properties and values which is automatically shared and synchronized between the service application, and any clients using or collaborating with that service application.

The property values are stored in XML format, and reside on both the service and the client. When creating the service application, developers, decide which properties are stored in this data structure.

For example, if the user of an application has the ability to change the color of an element on the screen, this color is a property value that could be stored in application state.

Unlike commands, which are one-way communications from a single client to the service, application state can be used to make changes known to several collaborating clients automatically.

For more information, see the chapter “Application State” on [page 40](#).

## Events

Events in PureWeb work in a similar fashion as events on other user interface frameworks.

PureWeb uses events in conjunction with views, commands, and application state to capture user input on the client.

For instance, when the user interacts with the client, for example by clicking on a button or pressing a key, the handler for that event could perform one or more of the following:

- change the value of a state property (for example, the color of an element on the screen),
- send a command to the service (for example, a command to connect when the user clicks the Connect button),
- transmit an updated image to display in a view.

The PureWeb STK APIs also provide the functionality to convert hand gestures on mobile devices to mouse events. For more information, see “Converting Touch-Screen Input to Keyboard Events” on [page 32](#).

---

## Communication Flow

Communication between the service, client, and server is handled via an HTTP(S) connection. In the HTML5 client, the connection occurs via web sockets..

The sections below provide a general overview of how these communications take place. This is for information purposes only, as all this is handled directly by PureWeb, and developers do not need to worry about managing image and application state communication.

## How Application State Remains Synchronized

The state of an application is stored on both the client and service as XML. Changes to the state can be introduced from either the service or the client, with the differences being transmitted to the other side to ensure state synchronicity.

Specifically, the service creates an instance of the `StateManager` class (responsible for maintaining application state) and passes it to an instance of `StateManagerServer`, which handles communication with the PureWeb server over a TCP socket. Messages arriving from the client via the server as XML text are converted to lists of command objects that are executed by the `StateManager`.

In executing the commands, the application will generate response objects, including changes to the application state. The `StateManagerServer` converts the response objects to XML and multipart messages that are passed back to the clients via the PureWeb server.

Once the `StateManager` instance exists in the service, it is possible to create event handlers that are invoked when specific nodes in the application state change.

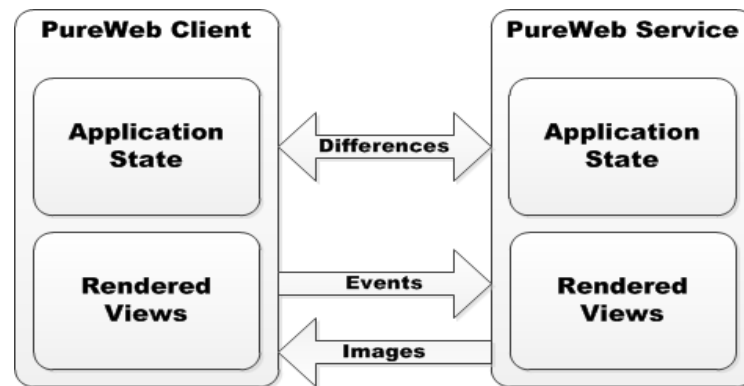
For more information, see the chapter “Application State” on [page 40](#).

## How Images in Views Are Kept Up-to-Date

PureWeb ensures that clients views receive images from service views by overriding the handler for the service's image drawing event and requesting an update to be sent to the client.

it is also possible to use these same PureWeb methods to request image updates outside of the standard application image updating scenarios.

For example, let's assume a client application has a Clear button which, when pressed, sends a `ClearPage` command to the service. The service receives the command and responds by deleting drawing on a page. It then informs the server that an updated image (a blank page) is available, and publishes the new image to the client view.



## How Events and Commands Are Communicated

As a user interacts with a PureWeb client, this might be generating keyboard or mouse events, in which case the client sends the events as XML through the PureWeb server, to the service application. The PureWeb APIs then translate these items into system-level events, allowing the service application to receive them as though the user was interacting directly with it.

Developers can also instrument native client user interface components, such as buttons and other controls, to generate application-specific commands. When a user presses such a button, a developer-defined command is sent from the client, to the PureWeb server, to the service application, which responds accordingly.

---

## Additional Components

The PureWeb STK also contain tools that provide functionality for data storage, diagnostics, logging and collaboration, in particular:

- **Resource Manager:** An interface that allows users to store and retrieve files on the service machine.
- **Diagnostics Panel:** A set of built-in tools that developers can use to help configure and troubleshoot PureWeb client applications during the development phase. The panel allows developers, for instance, to display trace messages going to and from the server, to view the XML representation of the application state, and to measure bandwidth and latency.
- **Acetate:** An interface which allows users in a collaborative session to draw and write on an overlay over rendered views.

Chapter

# 3 PureWeb Enablement in a Nutshell

This chapter gives a high-level overview of the steps involved in PureWeb enabling a service and client application. More detailed explanations and code samples are provided in the other chapters of this *Developer's Guide*.

Although this chapter describes the process in a linear fashion, in a typical PureWeb project, the order would be less rigid and more iterative: for instance, developers could add buttons to the client and hook them to the service using commands or state, add more interface elements and hook these up to the service, decide that another view is needed, and so on.

---

## Setting Up the Server Connection

The PureWeb server acts as an communication intermediary between the client and service application. Consequently, the first step in PureWeb-enabling an application is setting up connections to ensure that both the service and the client can communicate to the server.

### On the Service

Setting up the connection between an existing application and the server effectively converts that application into a service.

This is done with a few simple lines of code, an example of which is shown in the section “Connecting the Service” on [page 21](#).



## On the Client

The code to set up the connection between the client and the server first uses event handlers to determine if the service is connected.

Typically, a different handler is used for each possible state that the service may be in: active, disconnected, stalled, etc.

If the service is connected, then the client can connect. Sample code for this is provided in section “Connecting the Client” on [page 22](#).

---

## Setting Up the Views

Once there is an established connection to the server from the service and the client side, the next step typically involves choosing which views to display on the client.

As mentioned earlier, PureWeb-enabled application can remote all, or just a subset, of the view interfaces in the existing application. Which interfaces a developer chooses to remote depends entirely on the functionality that needs to be exposed in the web or mobile client. It is recommended to start by remoting one view at a time.

## On the Service

Since the service application is responsible for generating and processing the images, most of the effort involved in remoting a PureWeb view comes down to implementing the `RenderedView` interface from the service STK.

Sample code illustrating how to work with views on the service side are provided in the chapter “Views” on [page 25](#).

## On the Client

The PureWeb STK provides a native view for each client platform, exposing both the native language functionality in addition to the relevant PureWeb APIs.

Displaying a remoted view in a client application is quick and easy. Use the `view` element, available in the client STK, to add the view where it should be displayed. For a code sample of this task, see section “Displaying a Remoted View in a Client” on [page 29](#).

---

## Adding User Input Events to Views

PureWeb developers allow users to interact with views by adding events that capture keyboard and mouse input. It is also possible to simulate keyboard input for touch-screen devices.

The `RenderedView` interface provides the necessary methods for injecting mouse and keyboard events into the views.

Keyboard and mouse inputs that occur on the client application are captured by the `postKeyEvent` and `postMouseEvent` functions and sent to the original application. For more information, see the section “Handling User Input” on [page 29](#).

---

## Designing the Client Interface

After creating the views, a logical next step is to add buttons, dialog boxes and other user interface elements to the client application. These are built using the native toolkit for the client platform.

The feature set and appearance of the clients created using PureWeb do not need to match that of the original application. When designing new clients, developers can modernize the look and feel, reduce the feature set to simplify a mobile client application, or add new features that were not in the original application. It is also possible to combine PureWeb views with other native UI elements to create an entirely new application user interface for the client. For more information on these possibilities, see the chapter “Designing the Client Interface” on [page 47](#).

In large part, the remainder of the PureWeb enablement process then consists of hooking these interface elements to the service-side functionality using PureWeb commands and application state.

---

## Sending Instructions Using Commands

Commands are often used with event-driven user interface components such as buttons as a way to hook up the service-side functionality. They are simply instructions from the client requesting the service to execute a specific function.

Commands are one-way communications between a single client and the service. For communications from the service to the client, as well as for communication between multiple client applications, developers should use the application state manager instead.

Setting up commands in PureWeb is quick and easy, as described below.

## On the Service

Before a client can send commands, the service must be setup to handle them correctly. This consists in a single line of code that adds a UI handler, specifies the command string, and defines the handling function.

For an example of this code, see section “Setting Up Commands on the Service” on [page 37](#).

## On the Client

The command is expressed as a string and uses the client-side `queueCommand` function; it can have arguments and, optionally, a callback can be triggered on the client, completing the communication loop.

Several code examples are provided in the section “Sending Commands from the Client” on [page 38](#).

---

# Manipulating Application State

Once the client contains interface elements, the service needs to know if the properties of these elements change, for example if a user changes the color of a pen, so that these properties can remain synchronized between the client and the service.

Since commands are a one-way communication between a single client and the service, state is used in most other situations.

This is done through the `StateManager` class, which provides the methods to write, read, edit and delete content from the state tree.

Code samples for working with state can be found in the chapter “Application State” on [page 40](#).

Chapter

# 4 Communicating with the Server

This chapter describes how to establish server connections in the service application and the client application.

---

## About the Server

The PureWeb server is the intermediary that manages communications between a service and its clients. The PureWeb service and client applications cannot function properly if their connection to the server is not properly established. The PureWeb server also acts as a session manager, coordinates client disconnection and service process termination, and provides a variety of diagnostic and application information.

The server must be on the same system node as the service application, and must be configured correctly. This chapter assumes that the server is correctly set up. For information on performing these tasks, please refer to the *PureWeb Server Administration Guide*.

---

## Server Communication Workflow

In a typical PureWeb application, the workflow when establishing server connections is as follows:

- The client application initiates the connection process by requesting a session from the server.
- The server responds by launching the service application.
- The service, upon starting up, also connects to the server.
- The client then connects with the established session.

---

## Connecting the Service

Connecting the service to the server is the first step of the PureWeb enablement process. The code for this task does more than just establishing that connection: it effectively sets up the desktop workstation application to behave as a service. This can be accomplished using just a few lines of code. Below is an example of the service-side code in C#:

```
1  static class Program
2  {
3      public static PureWeb.Server.StateManager StateManager;
4      /// <summary>
5      /// The main entry point for the application.
6      /// </summary>
7      [STAThread]
8      static void Main()
9      {
10         StateManager = new PureWeb.Server.StateManager("MyApp",
11             Dispatcher.CurrentDispatcher);
12         StateManager.Uninitialized += new
13             EventHandler(StateManager_Uninitialized);
14
15         StateManagerServer server = new StateManagerServer();
16         if
17             (!string.IsNullOrEmpty(System.Environment.GetEnvironmentVariable("PUREWEB_PORT")))
18         {
19             server.Start(StateManager);
20         }
21     }
22     static void StateManager_Uninitialized(object sender, EventArgs
23     e)
24     {
25         Application.Exit();
26     }
27 }
```

This code accomplishes the following:

- Declares a static instance of the PureWeb `StateManager` class (line 3). This class can be globally used by the rest of the application. It is responsible for creating and updating application state, responding to input events and commands sent from the client, and generating responses such as update images to send back to the client application.
- Initializes the `StateManager` class with the service application's name (line 10).

- Adds an event handler to the PureWeb session uninitialization event, to allow for graceful disconnect (line 11).
- Creates and initializes an instance of the `StateManagerServer` class (line 13). This implements an input/output thread to receive events and commands from the client. Assuming the `PUREWEB_PORT` environment variable is defined, this start the PureWeb session.
- Starts the server using the `server.start` method (line 17).
- Provides the uninitialization event handler, which shuts the application down when the PureWeb session is terminated (line 20).

## Graceful Disconnect

The sample code above simply allows the session to terminate if the service application stops responding.

However, it is common practice for service applications to handle a termination command in response to the PureWeb session ending. It is also recommended to initiate a PureWeb session shutdown upon exiting the application.

To explicitly end a session upon termination, call the `Stop()` function on the `StateManagerServer` instance in the application exit event handler.

---

## Connecting the Client

This section deals with the connection process of a single client to a service application, and doesn't touch upon the case of multiple collaborative users.

The code to connect a client application should cover the changes in the state of the service session:

- Add event handlers that listen for changes in the service state (the client will connect to a session only when it is notified that the service is connected).
- Create a URL which targets the correct server application and specifies the client platform.
- Connect to the server using an instance of the `PureWeb Framework` class and the constructed URL.
- Gracefully disconnect from the server.

Here's an example of what the state listening part of this code might look like in a Silverlight client:

First, add handlers for server state changes

```
1 Framework.Instance.Client.IsStalledChanged += new
  EventHandler(Client_IsStalledChanged);
2 Framework.Instance.Client.SessionStateChanged += new
  EventHandler(Client_SessionStateChanged);
```

Then, define the handler for stalled connections on the server. In this example, the handler is named `Client_IsStalledChanged` and it simply logs the event.

```
1 void Client_IsStalledChanged(object sender, EventArgs e)
2 {
3     string message = Framework.Instance.Client.IsStalled ? "Stalled" :
4     "Not stalled";
5     Trace.WriteLine(message);
6 }
```

Refer to the state change handler (`Client_SessionStateChanged`) to find out what state the session is in, and handle each state appropriately. For the sake of keeping the example simple, the code for each case (failed, disconnecting, etc.) alerts the user of the current state with message boxes, but actual applications would be expected to properly handle changes in session state.

```
1 void Client_SessionStateChanged(object sender, EventArgs e)
2 {
3     if (Framework.Instance.Client.SessionState == SessionState.Failed)
4     {
5         Framework.Client.Disconnect();
6         MessageBox.Show("The connection to the server has been closed.");
7     }
8     {
9         MessageBox.Show("You are now logged out and disconnected.");
10    }
11    else if (Framework.Instance.Client.SessionState ==
12    SessionState.Connecting)
13    {
14        MessageBox.Show("You are now logged out and disconnected.");
15    }
16    else if (Framework.Instance.Client.SessionState ==
17    SessionState.Active)
18    {
19        if (Framework.Client.IsConnected)
20        {
21            // Bring the user to your application's main page.
22        }
23    }
```

Next, connect to the PureWeb server. This code must be placed after the initialization code.

```
1 System.Diagnostics.Debug.Assert(HtmlPage.IsEnabled);
2 // Get the current URL from the browser window
3 var href = (string)HtmlPage.Window.Eval("document.location.href");
4 // Start a new session with default credentials
5 Framework.Instance.Client.AuthorizationInfo = new
  BasicAuthorizationInfo() { Name = "admin", Password = "admin" };
6 }
7 // Call the PureWeb framework's connect method using the URL you
  created
8 Framework.Instance.Client.Connect(href);
```

Note also that in the above example, the server authentication credentials are hard-coded into the application for illustration purposes. In real life, you would likely handle authentication by using a custom login page.



# 5 Views

This chapter describes how to use the `IRenderedView` interface to make image elements of the service application visible to client applications in the form of PureWeb views.

Depending on the programming language, in some of the PureWeb service APIs this interface is called `RenderedView`; this is why in the rest of this chapter, it is referred to as `(I)RenderedView`.

---

## About Views

In its most simple form, a PureWeb view is an area on the screen that contains an image; this image is typically a portion of the original application's visible user interface. It is possible to have several independent views displayed on the same client screen.

PureWeb views are at the heart of the PureWeb STK. Here are some reasons why:

- They allow clients to be very thin, since the service application remains responsible to display the views displayed on the client application.
- They allow web and mobile users to interact seamlessly with complex graphical information, such as 3D images or animations, as if the entire application was native to their device, even across low bandwidth networks.

PureWeb allows developers to combine views with other native UI elements to create an entirely new application user interface for each client. For more information, see the chapter “Designing the Client Interface” on [page 47](#).

Developers can inject mouse and keyboard events into views to make them interactive; each view can have its own set of events. For more information, see “Displaying a Remoted View in a Client” on [page 29](#).

Developers also can also set parameters that control the quality of graphics displayed in views. For more information, see “Defining Image Quality for Views” on [page 33](#).

## Remoting a Service View

Since service applications are responsible for generating and processing the views, most of the effort involved in remoting a PureWeb view comes down to implementing the `(I)RenderedView` interface from the service STK.

There are two major approaches to implementing this interface:

- Implement it directly in each class where a view must be rendered. This method may work when the number of views is low, but in most cases, it is not the best option.
- Create a generic PureWeb view handling class. This is more efficient when there are several views; it is the approach discussed in the rest of this chapter.

In the Java service STK, such a generic class can be found in the samples; it is called `RemotedPanel.java`; in the C# service STK, it is called `RemotedControl.cs`. We encourage developers to use or reference these generic classes in their own PureWeb projects, and we recommend using the adapter design pattern for this.

### The `(I)RenderedView` Interface

The `(I)RenderedView` interface requires the implementation of five functions, summarized in the table below.

Function	Description
<code>setClientSize</code>	Specifies the desired rendering size of a view in pixels.
<code>getActualSize</code>	Returns the actual size of the view.
<code>renderView</code>	Handles the actual rendering of the view. It is called by the PureWeb server when requesting an updated image.
<code>postKeyEvent</code>	Communicates incoming keyboard events from the client, making them available to the service application.
<code>postMouseEvent</code>	Communicates incoming mouse events from the client, making them available to the service application.

The actual content of each of these functions will usually reflect the behavior of the application before it was PureWeb-enabled. PureWeb is structured in a way that allows the original logic and functionality to remain essentially unchanged.

## Registering Views

Views are identified by a unique name, which must be registered with the PureWeb state manager. For more information on state, see “Application State” on [page 40](#).

Where a view is registered depends on whether or not the `(I)RenderedView` interface is implemented in a generalized view handling class.

- If a generic view handling class is not used, register the view where desired within the view class – that may be the constructor of the class or another initializing function.
- If a generic view class is used, it is recommended to register the view upon construction of the view handling class – this ensures this handler class’ lifespan equals the registration period of the view it handles.

In either case, the command to register the view is the same. Below is an example of this command in C#:

```
1 Program.StateManager.ViewManager.RegisterView("MyView", this);
```

## Sample Implementation

Below is a basic example of an implementation of the `(I)RenderedView` interface in a Java Swing panel class.

```
1 public class PWPanel extends JPanel implements RenderedView{
2     protected final String viewName;
3
4     public PWPanel(String viewName){
5         this.viewName = viewName;
6     }
7
8     public void renderView(RenderTarget target){
9         paintComponent(target.getImage().getBitmap().getGraphics());
10    }
11    public Dimension getActualSize(){
12        return getSize();
13    }
14    public void setClientSize(Dimension clientSize){
15        if (!getSize().equals(clientSize)){
16            setSize(clientSize);
17            invalidate();
18        }
19    }
//continued on next page
```

```
20     public void postKeyEvent(PureWebKeyboardEventArgs keyEvent) {
21         //Discussed later in this chapter
22     }
23
24     public void postMouseEvent(PureWebMouseEventArgs mouseEvent) {
25         //Discussed later in this chapter
26     }
27 }
```

In the above code, `PWPanel` is a reduced form of the `RemotedPanel` included in the Java service STK.

The `renderView` function makes a call to the Java Swing `paintComponent` function. The contents of this function depends on what will be drawn to the screen and how.

The definitions of `getActualSize` and `setClientSize` simply make use of the `JPanel`'s `setSize()/getSize()` functions to get and set the dimensions of the panel.

The definitions for `postKeyEvent` and `postMouseEvent` are intentionally left blank, as they are be discussed further in this chapter (see “Handling User Input” on [page 29](#)).

This code illustrates the implementation of `(I)RenderedView` in its most basic form. For more advanced example, refer to the sample applications provided with the STK.

- Java:

```
[PureWeb_directory]\SDK\Samples\Java\src\server\pureweb\samples\RemotedPanel.java
```

- C#:

```
[PureWeb_directory]\SDK\Samples\Scribble\ScribbleAppCsharp\RemotedControl.cs
```

---

## Displaying a Remoted View in a Client

Displaying a remoted view in a client application is quick and easy. Use the `view` element, available in the client STK, to add the view where it should be displayed. Below is an example of a view called `viewName` being added to a client written in HTML/Javascript:

```
1 <div id="MyViewDiv" class="purewebview" style="width: 100px;
  height:100px;"></div>

  <script type='text/javascript'>
    pureweb.listen(pureweb.getClient(),
    pureweb.client.WebClient.EventType.CONNECTED_CHANGED,
    attachViewOnConnect);

    var attachViewOnConnect = function(event) {
      if (event.target.isConnected()) {
        myView= new pureweb.client.View({id: MyViewDiv', 'viewName':
        'MyView'});
      }
    };
  </script>
```

A few points to note:

- The view is attached only after the client has an active connection. Typically the listener for the `CONNECTED_CHANGED` event will be included alongside the rest of the client initialization code.
- The client view will not attach to the service application unless the name of the view being added to the client matches the name that was registered for the service view.

---

## Handling User Input

PureWeb allows developers to add events that capture keyboard and mouse input. It is also possible to simulate keyboard input for touch-screen devices.

### Keyboard and Mouse Events

Keyboard and mouse events that occur on the client application are captured by the PureWeb view, and sent to the service application where they are handled by the `(I)RenderedView` interface's `postKeyEvent` and `postMouseEvent` functions. Developers define how the service application responds to input events from the client based on the content of these events.

The actual content of `postKeyEvent` and `postMouseEvent` depends on the framework used for the GUI (Windows Forms, Microsoft Foundation Class (MFC), QT, Swing, etc.).

## Keyboard Input Example

Below is an example of how to capture and handle a keyboard event in C# using the Windows Forms framework:

```

1  //...
2
3  const int WM_KEYDOWN = 0x100;
4  const int WM_KEYUP = 0x101;
5  const int WM_SYSKEYDOWN = 0x104;
6  const int WM_SYSKEYUP = 0x105;
7  [return: MarshalAs(UnmanagedType.Bool)]
8  [DllImport("user32.dll", SetLastError = true)]
9  static extern bool PostMessage(IntPtr hWnd, UInt32 Msg, IntPtr
wParam, IntPtr lParam);
10
11 //...
12
13 public void postKeyEvent(PureWebKeyboardEventArgs keyEvent){
14     bool isAltDown = 0 != (keyEvent.Modifiers &
Modifiers.Alternate);
15
16     int wParam = (int)keyEvent.KeyCode;
17     int lParam = isAltDown ? (1 << 29) : 0; // "context code";
18     int message;
19
20     if (isAltDown || keyEvent.KeyCode == KeyCode.F10){
21         message = keyEvent.EventType == KeyboardEventType.KeyDown ?
WM_SYSKEYDOWN : WM_SYSKEYUP;
22     }else{
23         message = keyEvent.EventType == KeyboardEventType.KeyDown ?
WM_KEYDOWN : WM_KEYUP;
24     }
25
26     PostMessage(this.Handle, (UInt32)message, new IntPtr(wParam),
new IntPtr(lParam));
27 }

```

In the above example, `postKeyEvent` starts by determining if the `Alt` key is down (line 14), as this will impact the key code that gets sent.

Next, the key code and context code are determined (lines 16 - 18).

Then, the message is interpreted based on whether `Alt` or `F10` are pressed.

Finally, the message, key code and context code are passed to `PostMessage` (line 26), a part of Windows made available through `user32.dll` (line 8).

The `PostMessage` function will generate the keyboard event in the appropriate window, just as though the client had actually interacted with the service application directly without the `PureWeb` layer.

## Mouse Input Example

The next example shows how mouse events would be captured by the same C# application and using the Windows Forms framework:

```
1 public void PostMouseEvent(PureWebMouseEventArgs mouseEvent) {
2     System.Windows.Forms.MouseButtons buttons =
3         System.Windows.Forms.MouseButtons.None;
4     if (0 != (mouseEvent.Buttons & PureWeb.Ui.MouseButtons.Left))
5         buttons |= System.Windows.Forms.MouseButtons.Left;
6     if (0 != (mouseEvent.Buttons & PureWeb.Ui.MouseButtons.Right))
7         buttons |= System.Windows.Forms.MouseButtons.Right;
8     if (0 != (mouseEvent.Buttons & PureWeb.Ui.MouseButtons.Middle))
9         buttons |= System.Windows.Forms.MouseButtons.Middle;
10    if (0 != (mouseEvent.Buttons &
11        PureWeb.Ui.MouseButtons.XButton1)) buttons |=
12        System.Windows.Forms.MouseButtons.XButton1;
13    if (0 != (mouseEvent.Buttons &
14        PureWeb.Ui.MouseButtons.XButton2)) buttons |=
15        System.Windows.Forms.MouseButtons.XButton2;
16
17    switch (mouseEvent.EventType) {
18        case MouseEventType.MouseEnter:
19            OnMouseEnter(EventArgs.Empty);
20            break;
21
22        case MouseEventType.MouseLeave:
23            OnMouseLeave(EventArgs.Empty);
24            break;
25
26        case MouseEventType.MouseMove:
27            OnMouseMove(new MouseEventArgs(buttons, 0, (int)mouseEvent.X,
28                (int)mouseEvent.Y, (int)mouseEvent.Delta));
29            break;
30
31        case MouseEventType.MouseDown:
32            OnMouseDown(new MouseEventArgs(buttons, 0, (int)mouseEvent.X,
33                (int)mouseEvent.Y, (int)mouseEvent.Delta));
34            break;
35
36        default:
37            Trace.WriteLine("Received unknown mouse event type {0}.",
38                (int)mouseEvent.EventType);
39            return;
40    }
41 }
```

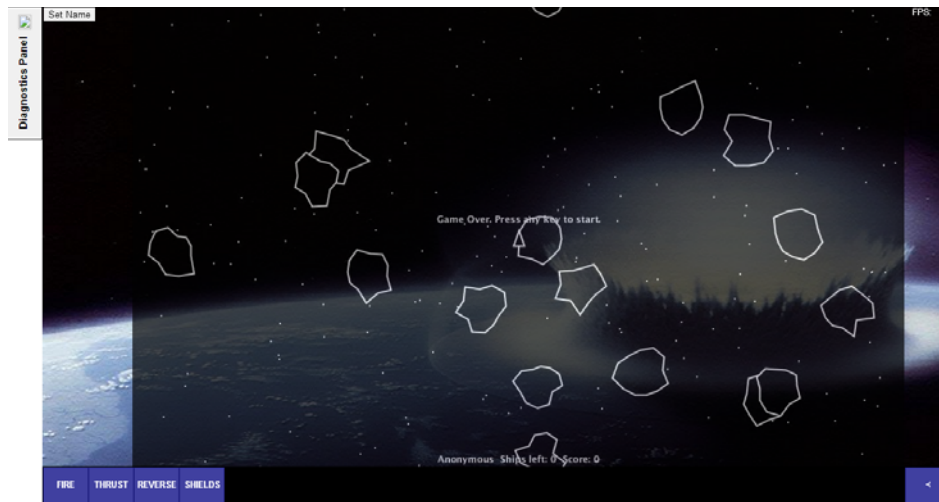
The above example boils down to two key operations. The first set of conditional statements takes the `PureWebMouseEventArgs` argument and converts them to a WinForms button object based on which buttons were clicked.

The `switch` statement determines which type of mouse event was reported by the PureWeb client, and then fires the corresponding WinForms mouse event with the computed buttons object.

## Converting Touch-Screen Input to Keyboard Events

It is possible for PureWeb clients to simulate keyboard presses for touch-screen devices. Fundamentally, this is achieved by creating a key-down command when certain parts of the screen are touched.

An example of this approach is provided in the HTML5 Asteroids sample application, though a similar interface could be achieved in Android or iOS. Notice, when the application runs (for instructions on how to build the application, refer to the *Quick Start Guide: HTML5 Client*), that a button panel at the bottom of the screen allows users to play the Asteroids game on touch-screen devices:



The code used to implement this button panel is described below. The code can be found in the following file:

```
[PureWeb_directory] \SDK\Samples\Asteroids\AsteroidsClientHTML5
```



When the user touches a button on the panel, the client application simulates a key press by sending a command to PureWeb (for more information on commands, see section “Commands” on [page 36](#)):

```
1 function simKeyDown(e, keyCode) {
2   //Simulate a keyboard event
3   var eventObj = queueKeyboardEvent('KeyDown', keyCode);
4   //Suppress the default action e.preventDefault(); //Send a
   keyboard event using a PureWeb commandfunction
5   queueKeyboardEvent(eventType, keyCode) {
6     //Create the keyboard event as a JS object
7     var parameters = {'EventType': eventType, 'Path': 'AsteroidsView',
8                       'KeyCode': keyCode, 'CharacterCode': 0, 'Modifiers': 0 };
9     //Send the PW command
10    pureweb.getClient().queueCommand('InputEvent', parameters);};
```

---

## Defining Image Quality for Views

Developers can control the quality and encoding format of the views displayed in a client application.

Quality refers to the clarity and number of visual artifacts present in the view. When quality is low, the fidelity of the images generated by the service is low, and the size of each view update is smaller. Lowering view quality can therefore be used to conserve bandwidth, reduce latency, and improve performance.

Encoding format refers to the format (mime type) in which the image is transmitted, such as JPEG, PNG, or tiles.

PureWeb allows developers to set image quality and encoding format for two image transmission modes: interactive and non-interactive.

- Interactive mode: the user is interacting with the view, for example when clicking and dragging the mouse to rotate a 3D model or pan a 2D diagram.
- Non-interactive mode: the view is not processing any input event. Because the quality of the images sent in non-interactive mode is usually higher than in interactive mode, this mode is also referred to as “FullQuality”.

A PureWeb application’s encoder configuration is therefore the combination of quality and encoding format for each of these modes. For example, it is possible to set the application to transmit JPEG images at 30% quality when the user is interacting with the view, and to transmit full-quality PNG images when the user is not. Every client, for each view, can have a different configuration.

The sample applications offer an easy way to manipulate the encoding configuration settings and see their effect on the fly. For more information, refer to the section “See It In Action” on [page 35](#).

Using the encoder configuration API is entirely optional. The default configuration is tiles at 30% quality for interactive mode and JPEGs at 70% quality for non-interactive mode.

---

Note: A few notes about encoding format:

- PNG images, unlike JPEG and tiles, are always transmitted at 100% quality, because they use loss-less encoding. If the configuration is set to send PNGs at 40% quality, the PNGs will still be sent at 100% quality.
- When using JPEG or PNG, the application updates the entire view. Tiles, on the other hand, are small header-less JPEG images which only update the parts of the view that have changed since the last update; this sends noticeably less data to the clients.
- Tiles require more computational effort on the part of the computer running the service application. Although this encoding format is supported in all client platforms, tiles perform best when the client application is Flex or Silverlight; on other platforms, JPEGs perform best.

---

## Setting the Encoder Configuration

The Diagnostics Panel's graphics interface makes it easy to change the encoding configuration and see the impact of these changes on the fly. For more information, refer to the sections "See It In Action" on [page 35](#) and "Diagnostics Panel" on [page 57](#). However, changing the configuration in the Diagnostics Panel is not a permanent operation; for the changes to be permanent, it is necessary to change the encoder configuration programmatically as described in this section.

---

Note: The PureWeb APIs used for encoding configuration have changed between version 3.1 and 4.0. If you developed applications using 3.1, refer to the 4.0 release notes for more information about what changed and for backward compatibility considerations.

---

Each PureWeb view exposes a reference to an `EncoderConfiguration`. Changing the encoder configuration is simply a matter of creating or modifying two `EncoderFormat` objects within this `EncoderConfiguration`. The code below illustrates how to accomplish this in Silverlight.

```
1 MyView.EncoderConfiguration.InteractiveFormat.Quality = 45;
2 MyView.EncoderConfiguration.InteractiveFormat.MimeType =
  "image/tiles";
3 MyView.EncoderConfiguration.InteractiveFormat.Quality = 90;
  MyView.EncoderConfiguration.FullQualityFormat.MimeType =
    "image/jpeg";
```

The `EncoderFormat` further exposes a parameters dictionary as an extension point. These parameters are also made available on the service side, as `EncoderParameters` on the `RenderTarget` interface. This allows developers to make potential rendering decisions based on the encoding format and quality that will eventually be used.

To determine whether to use the settings for interactive or non-interactive modes, the application relies on the value for the `SetViewInteracting` parameter on the `ViewManager`. If this value is set to `true`, the application will use the interactive mode settings, if it is set to `false`, the application will use non-interactive (full quality) mode settings.

In the example below, the `SetViewInteracting` parameter is set to `true`.

```
1  protected override void OnMouseDown(MouseEventArgs e)
   {
       if (e.Button == MouseButton.Left || e.Button ==
           MouseButton.Right)
       {
           // Tell the StateManager we are interacting with this view
           Program.StateManager.ViewManager.SetViewInteracting(ViewName,
               true);
           Capture = true;

           BeginStroke();
           m_currentStroke.Add(e.Location);
           DrawCurrentStroke();
           RemoteRender();
       }
   }
```

## See It In Action

The sample Scribble application provides a quick way to see interactive quality in action. The example below is based on the Silverlight client. For instructions on how to build and run this client, refer to the *Quick Start Guide: Silverlight Client* document.

To view the Diagnostics Panel, launch the client with the `_diagnostics=true` parameter in the URL. With the diagnostics panel expanded, set **Non-interactive mode** quality at its highest and **Interactive mode** quality at its lowest, and apply the changes. Then, in the application's main screen, change the color to blue (the change in quality is most noticeable with the color blue). Notice that the image quality is lower while drawing, and is restored to higher quality when the user stops drawing.

# 6 Commands

This chapter describes how to generate commands on the client application, and how to handle these commands in the service application.

---

## About Commands

A command is an instruction sent from the client for the service to execute a specific function.

The command is expressed as a string and uses the client-side `queueCommand` function; it can have arguments and, optionally, a callback can be triggered on the client, completing the communication loop.

In PureWeb, commands can only originate from clients, and they are always handled by the service.

When communications need to occur from the service to the client, the best course of action is to post the information to the application state tree. For more information, see “Application State” on [page 40](#).

Here are some scenarios from the sample applications illustrating the use of commands:

- In Scribble, when a user clicks on the **Erase All** button to restore the canvas to its blank state, the client application sends a `Clear` command.
- In both Scribble and Asteroids, when the user clicks on the **Share** button, the client sends a `generateShareURL` command.

---

## Setting Up Commands on the Service

Before a client can send commands, the service must be set up to handle them correctly. This consists in a single line of code that adds a UI handler, specifies the command string, and defines the handling function.

Here's an example, taken from the Scribble sample application.

When a user clicks on the **Erase All** button to restore the canvas to its blank state, the client application sends a `Clear` command.

The code to set up this `Clear` command in a C++ service application would look like this:

```
1 CScribbleApp::StateManager().CommandManager().AddUiHandler("Clear"
  , Bind(this, &CScribbleView::OnExecuteClear));
```

In this example, the command string is “Clear”, and the user-defined function that will run when the command is received from the client application has been named `OnExecuteClear`.

### Registering and Unregistering Command UI Handlers

To ensure that all commands are correctly caught and handled by the service application, command handlers should be added early on in the program initialization.

The PureWeb client APIs also include `RemoveUiHandler()` functions. Using these is not mandatory, but it is recommended to unregister handlers when terminating the service application.

### Populating Callback Parameters on the Service

Client applications can register a callback function in the `queueCommand` call. For an example of how this is done, see “Example 3” on [page 38](#).

If the client has specified a callback, the service application can define (populate) response parameters for that callback. Below is an example from a C# command handler:

```
1 private void OnExecuteClear(Guid sessionId, XElement command,
  XElement responses)
2 {
3     //Add responses to the provided response object.
4     responses.Add("Foo");
5 }
```

In this example, any content added to `responses` will be made available to the client callback function in the form of an XML element.

---

# Sending Commands from the Client

In the client application, commands are sent to the service application using the `queueCommand` function. This function adds the command, including any of its arguments, to the outgoing command queue, which delivers the commands.

The command string registered on the client side must match the string as set up in the service side precisely, otherwise the command will not be correctly handled.

The examples below illustrate `queueCommand` instruction, some straightforward, some complex with arguments and callbacks.

All are examples of the Clear command from the sample Scribble application discussed earlier in this chapter.

## Example 1

This is a simple command in Silverlight:

```
1 Framework.Instance.Client.QueueCommand("Clear");
```

In this example, the Clear command was already registered in the service application, allowing it to respond accordingly.

## Example 2

This is a command with arguments but no callback, using the Android API:

```
1 Map<String, Object> parameters = new HashMap<String, Object>();
2 parameters.put("argumentName", "firstArgumentValue");
3 framework.getWebClient().queueCommand("Clear", parameters);
```

## Example 3

This is a command with both arguments and a callback, using the JavaScript API:

```
1 pureweb.client.getClient().queueCommand('Clear',
2     {argumentName:'argumentValue'},
3     function (sender, args) {
4         //Callback
5         console.log('Command completed: ` + args);
6     });
```

In this last example, the callback is in-line, but this doesn't have to be the case.

The callback sends a logging message to the console, including the callback arguments: the object that sent the callback, and the callback parameters expected from the service application.

## Example 4

Here is another, more practical, example of a command with a callback; in this Javascript/HTML5 example, the callback is used to alter the user interface:

```
1  <script type='text/javascript'>
2      function clearCanvas() {
3          pureweb.client.getClient().queueCommand("Clear", null,
4              clearCallback);
5      }
6
7      function clearCallback(){
8          document.getElementById('clearButton').innerHTML = 'Cleared';
9      }
10 </script>
11 <button id="clearButton" onclick="clearCanvas();">Clear</button>
```

In this example, the `queueCommand` function takes a callback, which locates the relevant button on the page and changes its label to indicate that the command was successfully processed by the service.

# 7 Application State

This chapter describes how to use application state as a means of communicating information between the service and clients.

---

## About Application State

The state of the application is essentially a hierarchical set of properties and values which is automatically shared and synchronized between the service application, and any clients using or collaborating with that service application. The property values are stored in XML format.

For example, there are many properties that can describe the state of a specific button, such as whether it is enabled or even visible, and what text or image it contains. Another example of a property value that could be stored in application state would be the color of an element on the screen, if the user has the ability to change that color.

The main purpose of application state is to keep the values of these properties synchronized between the service and its clients (there is one application state for each instance of a service). Developers decide which properties should be stored in the application state of their own application.

The Diagnostics Panel's graphic interface offers a user-friendly mode of viewing the state of an application. For more information, see "Diagnostics Panel" on [page 57](#), and in particular the section "AppState Tab" on [page 60](#).



## When to Use Application State

When developers need to communicate information between client and service applications, they can use either application state or commands.

The method chosen in a particular case depends on what will be sent and where. Although there are no hard and fast rules dictating when to use one method over the other, below are general guidelines:

- Application state is best used for information that is relevant to every client and service in a session, as it ensures that all clients can be notified.
- Application state can also be used to communicate from a single client to the service, if it is important that all other clients are aware of this communication.
- A command is useful for sending messages between a service and a single client.

---

## State Tree

The application state is managed using the `StateManager` class. The information is stored as a hierarchical XML tree.

This makes application state very flexible: it can contain basically anything that can be stored as an XML element or collection of elements.

When referring to a value in the tree, use a path notation, which corresponds to the hierarchy of XML elements enclosing that value, for example:

```
/PureWeb/InteractiveQuality  
/PureWeb/FullQuality
```

A few of the elements in the PureWeb application state tree have particular rules to follow, as described below.

### The `<ApplicationState>` Root Element

All application state elements are children of the root `<ApplicationState>` element. However, when setting or accessing elements within application state, this first level will not be included in the path.

### The `<PureWeb>` Element

The `<PureWeb>` element is created by the `StateManager` class upon initialization.

It contains information about the service application, image quality properties and session information which is primarily used when multiple clients are collaborating. This information is entirely maintained by PureWeb.

To prevent conflicts, it is strongly recommended that users not write to the `<PureWeb>` element or its children.

Below is an example of an application state tree.

```

1  <ApplicationState
   xmlns:typeless="http://calgaryscientific.com/typeless/2008">
2    <PureWeb>
3      <Application>ScribbleApp</Application>
4      <ClientCommandFiltering>1</ClientCommandFiltering>
5      <InteractiveQuality>30</InteractiveQuality>
6      <FullQuality>70</FullQuality>
7      <Sessions>
8        <SessionId-01000000-0000-0000-0000-000000000000 />
9      </Sessions>
10     <Collaboration>
11       <OwnerSession>01000000-0000-0000-0000-000000000000</OwnerSession>
12       <Sessions>
13         <SessionId-01000000-0000-0000-0000-000000000000>
14           <DefaultColor>#FFFF0000</DefaultColor>
15           <UserInfo />
16         </SessionId-01000000-0000-0000-0000-000000000000>
17       </Sessions>
18     </Collaboration>
19   </PureWeb>
20   <ScribbleColor>White</ScribbleColor>
21 </ApplicationState>

```

In the above example, which is taken from the Scribble sample application, the `<ScribbleColor>` element is the only user-provided value in the state tree.

---

## Initializing Application State

The application state must be initialized before a PureWeb application can start reading from and writing to it.

This initialization needs to occur when the service application is started. Typically, the state initialization code should be placed early on in the initialization of the service.

To initialize application state, create new `StateManager` and `StateManagerServer` objects in the service application. Here is an example of how this was done in the Scribble C# sample application

```

1  StateManager = new PureWeb.Server.StateManager("ScribbleApp",
   Dispatcher.CurrentDispatcher);
2  StateManagerServer server = new StateManagerServer();
3  server.Start(StateManager);

```

## Creating State Initialization Handlers

The `StateManager` object fires an event when it has been successfully initialized. A handler can be chained to this event, so that the application can perform actions, for example load initial values for some properties, when this occurs. Such handlers can be on either the service or the client application. They can include any command which interacts with application state: add, remove, modify, or read. For information about writing these commands, see “Interacting With The Application State” on [page 43](#).

Here’s an example of adding a user-defined handler called `ModifyMyAppState` in a C# service application.

```
1 StateManager.Initialized += new EventHandler(ModifyMyAppState);
```

Client applications can listen for the event which is fired when the application state is initialized. Here’s what this might look like in JavaScript:

```
1 pureweb.client.listen(pureweb.client.getClient(),
2 pureweb.client.Framework.EventType.IS_STATE_INITIALIZED,
3 function(){
4     //Things to do once state has been initialized
5 }
6 });
```

Alternatively, to allow a client application to interact with application state when it did not listen for the state initialization event, query for the `isStateInitialized` status, as shown below in JavaScript:

```
1 if (pureweb.client.framework.isStateInitialized()){
2     //Interact with state
3 }
```

---

## Interacting With The Application State

Once state is initialized on the service, or the client has been notified that state has been initialized, the application can interact with `StateManager`.

This interaction can be direct, or through change handlers.

### Direct Interaction

The primary purpose of application state is to provide a synchronized data store between a service and its clients; it is therefore not surprising that the most common use of application state is to read and write values directly to it.

Recall that values in application states are referenced by a path which corresponds to the tree hierarchy, and that the `<ApplicationState>` element is not included in the path.

## Writing to Application State

The examples below shows how to write a string into state.

In these particular examples, the value would be accessible in the path `/ScribbleColor`.

In Silverlight:

```
1 Framework.Instance.State["ScribbleColor"] = "red";
2
3 //OR... (where m_stateManager is an instance of StateManager
4 m_stateManager.XmlStateManager.SetValue("/ScribbleColor", "red");
```

In Java (in this case, the framework object is just an instance of Framework):

```
1 framework.getState().setValue("ScribbleColor", "red");
```

## Reading from Application State

Values can be read from state in a similar way.

In Silverlight:

```
1 var color = Framework.Instance.State["ScribbleColor"];
2
3 //OR...
4 Framework.getState().GetValue("/ScribbleColor");
```

In Java (as above, the framework object is just an instance of Framework):

```
1 framework.getState().getValue("/ScribbleColor");
```

## Advanced Methods

PureWeb supports a variety of methods to accomplish more complex application state reading and writing tasks.

### getValueAs()

This method allows developers to retrieve parsed values from the application state; a data type must be specified. Refer to the PureWeb APIs reference material for more information on this method.

### setTree() and getTree()

These methods allow developers to get or set a section of application state. They take and return XML elements which can be read from or inserted into the path specified in the arguments.

The objects that store XML trees are platform-specific, represented in the native XML data type for that language, for instance `XElement` (C# and Silverlight) or `Element` (Java). Refer to the PureWeb API reference material to find out the exact object type used.

The only PureWeb client API to deviate from this model is HTML5. In this API, `setTree()` and `getTree()` take and return a JSON object, because this is a more intuitive format for storing complex data in JavaScript. However, the API does provide `getTreeAsXml()` and `setTreeAsXml()` if XML is the desired format.

## stateLock

In mutli-client environments, the application state should not be assumed to be perfectly synchronized, due to network latency or other factors. For this reason, it is possible to request a lock on the `StateManager`, which ensures that no changes to the application state take place while the lock is held.

Below is an example of how to acquire a state lock in Objective C. Once acquired, the `stateLock` provides an API similar to `StateManager`:

```
1 PWXmlStateLock *stateLock = [_stateProvider acquireLock];
```

---

# Change Handlers

Change handlers allow developers to respond to a particular change in application state by registering a change handler. These handlers come in two varieties, value change handlers and child change handlers.

## Value Change Handlers

Value changed handlers are triggered when a specific value changes. They can be added to any path in the application state tree. If the value at that path changes, the associated handling function will be called.

Typically, value change handlers are defined inside the application state initialization function. Below is an example using the C++ API:

```
1 stateManager.XmlStateManager().AddValueChangedHandler("/PureWeb/Profiler", Bind(this, &MyApp::OnProfilerStateChanged));
```

In this example, when the value at `/PureWeb/Profiler` changes, the service application will execute `OnProfilerStateChanged`.

The function that handles a state change event (in this case `OnProfilerStateChanged`), provides a `ValueChangedEventArgs` argument. This argument contains information about the change that triggered the event, including the path at which the change occurred and the new value.

## Child Changed Handlers

Child changed handlers are triggered if any changes occur at or below the specified path in the XML tree.

Below is an example similar to the above, using Objective C:

```
1  [[PWFramework sharedInstance].state.stateManager
    addChildChangedHandler:@"/PureWeb/Profiler" target:_myApp
    action:@selector(OnProfilerStateChanged)];
```

# 8 Designing the Client Interface

This chapter describes the options available to developers when creating user interfaces (UI) for their PureWeb-enabled client applications, and provides practical examples for adding interface elements such as buttons and linking them to the service-side functionality.

---

## Feature Set and Appearance

When PureWeb enabling an application, developers link PureWeb to their service application's logic, and not to its user interface. They then create the client application separately, using primarily the native interface elements (buttons, dialogs, etc.) present on the client platform (Android, iOS, HTML5, etc.). This is part of what makes PureWeb so flexible and powerful and offers several advantages:

- Developers can have several client platforms created for the same service, for example one optimized for viewing in web browsers, and one for tablet. Creating the clients does not require changes to the service.
- Each client has its own user interface, making the application feel like a native application on the end user's device.
- The user interfaces for the PureWeb clients are not limited by the look and feel of the original application. This is the perfect opportunity to modernize the appearance of dated legacy software, for example by adding graphics or transition animations.

- Developers can change the application feature set depending on the client: it is not uncommon, for example, for mobile clients to expose only a subset of features compared to the desktop version. It is also possible to create entirely new functionality that is not available in the interface of the service application.

Even the workflow can be changed, if it makes sense for the target audience, as long as the handling logic is added to the service application. Consider for example an image manipulation service application which gives users the option to switch into a zoom mode and only then magnify/miniaturize their image. On an iOS PureWeb client, this sequence of events could happen automatically when a pinch zoom in/out is initiated.

The rest of this chapter describes how the native UI elements of a client platform can be enabled with the PureWeb API to drive the functionality of the service.

The only part of the client's user interface which is handled differently is the view, which is essentially a wrapper around the native UI components. For more information on views, see chapter "Views" on [page 25](#).

---

Note: Client applications are built using the platform's native functionality, and therefore the code for coupling these element to PureWeb is unique to each platform. However, the general pattern remains the same. The examples in this chapter are written in a combination of HTML and JavaScript.

---

---

## Adding a PureWeb Layer to UI Elements

In PureWeb, developers have two options available for communicating information between the service and its client applications: commands and application state. For more information on these options, see their respective chapter: "Commands" on [page 36](#), and "Application State" on [page 40](#).

The UI elements in a client application can use either commands or application state to tell the service how to respond to user interactions.

### Using Commands

Commands are a form of communications sent from the client to the service, requesting the execution of a specific function. PureWeb developers often use them to tell the service what to do when the end user interacts with an interface element.

In the example below, a PureWeb command is coupled with the **Erase All** button, from the Scribble sample application, to queue the `Clear` command. This



example shows both the HTML and JavaScript necessary to enable this functionality:

```
1 <script type='text/javascript'>
2   function clearCanvas() {
3     pureweb.client.getClient().queueCommand("Clear");
4   }
5 </script>
6
7 <button onclick="clearCanvas();">Erase All</button>
```

This is a simple case: the button is created with an `onclick` event handler (line 7). The event handler points to the `clearCanvas` function (line 2), which queues up the desired command, in this case, `Clear` (line 3).

## Using Application State

It is often preferable to use application state instead of commands, for example when communication needs to flow from the service to the client, or in a collaborative session where multiple client applications using the same service need to be made aware of a change in a property value.

In the example below, also taken from the HTML5 sample Scribble application, a select box is tied to the PureWeb application state to trigger a change in the user's pen color.

```
1 <script type='text/javascript'>
2   function changeScribbleColor(e) {
3     pureweb.client.getFramework().getState().setValue('ScribbleColor', document.getElementById('color').value);
4   }
5 </script>
6
7 <select onChange="changeScribbleColor();" id="color">
8   <option value="White">White</option>
9   <option value="Red">Red</option>
10  <option value="Blue">Blue</option>
11  <option value="Green">Green</option>
12 </select>
```

In this example, the developer created a select box for the pen color (lines 7 - 12). When the user changes the value in this box, the `changeScribbleColor` function is called (line 7). This function (line 2) simply gets the value of the selected option and sets it in application state at the path 'ScribbleColor' (line 3).

To extend this example even further, it would be possible to store the list of possible pen colors in application state, and when this list changed in the application state, the options in the select box could be dynamically adjusted accordingly.

# 9 The Resource Manager

The Resource Manager interface provides functionality which allows users to store and retrieve files on the service machine. This feature is commonly used in a collaborative session, when multiple users are connected to the same PureWeb session.

The server handles storing the files and providing a temporary access link to them. Service and client applications can then access the files.

Not all of the methods available in the resource manager interface are described in detail in this chapter. For additional information, consult the API reference material. For the service, the information can be found in the following path:

- C++: `CSI::PureWeb::Server::IResourceManager`
- C#: `PureWeb::Server::IResourceManager`
- Java: `pureweb.server.ResourceManager`

For the client, the description for the methods `RetrieveObject` and `GetResourceURL` can be found in the `WebClient` class of the `pureweb.client` package.

Once files are stored using the resource manager, they remain available for as long as the application session remains active.

---

Note: The Resource Manager currently has a 2 GB limit, due to how the `RetrieveObject` method in the various APIs retrieves the resources.

---

---

## Managed Access

Whenever a service application adds a resource to its resource manager, the resource manager creates a unique identifier (GUID key) for that particular resource, which can then be used to retrieve it.

If the requesting user is not given the GUID key for a resource, that user will not be able to access it. A service can therefore decide which GUID key to send to which client, and hence manage which clients have access to what files.

The service also has the ability to write the GUID keys into PureWeb application state, thereby allowing clients to access any currently stored files, provided the clients are made aware of these particular application state paths.

Clients can construct a URL from the GUID key and use this URL to access the resource directly (in iOS, Java and Android clients), or with a web browser.

---

## Workflow

Typically, the need to store a resource is communicated from the client to the service through a command. This implies adding to the client a method that sends the command, and adding to the service a method to handle that command.

The command handler on the service should include the following:

- create the resource,
- store the resource using the `Store()` method of the resource manager interface,
- return a GUID key for that resource as a response.

The method sending the command on the client should include the following:

- queue the command,
- wait for a callback that provides the GUID key,
- retrieve the resource using the provided GUID key,
- perform whatever action is needed on the resource, if applicable.

---

## Example - Screen Captures

This section illustrates how the resource manager stores and retrieves screen captures. The code is in C++ for the service, and in Silverlight for the client.

To try this example, edit the Scribble sample application for these programming languages. The necessary files are in the installed PureWeb directory on the server machine:

Service: [PureWeb\_location]\SDK\Samples\Scribble\ScribbleAppCpp

Client: [PureWeb\_location]\SDK\Samples\Scribble\ScribbleClient

## Service Application Code (Storing Data)

Assuming that the service will receive a command from the client application to save a screen capture, the service will first need a handler for that command. In the Scribble sample application, add the following code to the constructor in the `ScribbleView.cpp` file:

```
1  CScribbleApp::StateManager().CommandManager().AddIoHandler("Save",
   Bind(this, &CScribbleView::OnExecuteSave));
```

This command handler responds to the PureWeb command `Save` by executing the `OnExecuteSave` method, shown below.

```
1  void CScribbleView::OnExecuteSave(CSI::Guid sessionId,
   CSI::Typeless command, CSI::Typeless response)
2  {
3      PureWeb::Image image(m_pPixelBits, m_Width, m_Height,
   PureWeb::PixelFormat::Bgr24, PureWeb::ScanLineOrder::TopDown, 4);
4      ByteArray jpeg = PureWeb::JpegEncoder::JpegCompress(image, 80);
5      ContentInfo content("image/jpeg", jpeg.AsByteArray());
6      Guid key =
   CScribbleApp::StateManager().ResourceManager().Store(content);
7      response["ResourceKey"] = key;
8  }
```

This code:

- creates a PureWeb image given the current view's pixel bits and dimensions (line 3) -- the variables `m_pPixelBits`, `m_Width` and `m_Height` are defined elsewhere in the Scribble sample code
- converts the PureWeb image into a JPEG (line 4),
- packs the JPEG into a `ContentInfo` object (line 5),
- stores the `ContentInfo` object into the application's resource manager and returns a GUID (line 6),
- returns the GUID as `ResourceKey` to the calling application (line 7).

Next, add a reference to this new method in the header file (`ScribbleView.h`):

```
1  void OnExecuteSave(CSI::Guid sessionId, CSI::Typeless command,
   CSI::Typeless response);
```

## Client Application Code (Retrieving Data)

On the client side, first add a button which, when clicked, will run the method to request a screen capture of the service application's view; in this example, the method is called `SaveButton_Click`.

To create the button in the sample Silverlight client, add line 2 in the file `MainPage.xaml`.

```

1  <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="2" >
2    <Button x:Name="save" Content="Save" Margin="4,3,0,3" Width="54"
      Click="SaveButton_Click"/>
3  </StackPanel>

```

Then, add the `SaveButton_Click` method to the `MainPage.xaml.cs` file. In this example, the code includes both the `RetrieveObject` and the `GetResourceURL` methods, for illustration purposes. The `RetrieveObject` method would be used internally by the program and does not display the retrieved resource to the end user. However, with the `GetResourceURL` method the user will be able to see the saved screen capture when the program runs:

```

1  private void SaveButton_Click(object sender, RoutedEventArgs e)
2  {
3      Framework.Instance.Client.QueueCommand("Save", (o, args) =>
4      {
5          if (args.Exception != null)
6          {
7              Trace.WriteLine("Except during RPC: " + args.Exception);
8              return;
9          }
10         var key = args.Response.GetTextAs("ResourceKey", Guid.Empty);
11         //section illustrating the the retrieveObject method
12         var binaryObject =
13             Framework.Instance.Client.RetrieveObject(key);
14         var bitmapImage = new
15             System.Windows.Media.Imaging.BitmapImage();
16
17         using (var stream = new
18             System.IO.MemoryStream(binaryObject.Object))
19         {
20             bitmapImage.SetSource(stream);
21         }
22
23         var image = new Image { Source = bitmapImage };
24         //section illustrating the getResourceURL method
25         var resourceUrlText = new TextBox { Text =
26             Framework.Instance.Client.GetResourceUrl(key), IsReadOnly = true };
27
28         //code continued on next page

```

```
23     var stackPanel = new StackPanel();
24     stackPanel.Children.Add(resourceUrlText);
25     stackPanel.Children.Add(image);
26
27     var screenshotWindow = new ChildWindow { Title = "Screenshot
    image", Content = stackPanel };
28     screenshotWindow.Show();
29     });
30 }
```

This method:

- queues the `Save` command and waits for a callback (line 3)
- verifies, when the callback is received, that the storage was successful (line 5)
- looks for the GUID saved in the `ResourceKey` response path, from the `OnExecuteSave` method on the service application (line 10)
- uses the GUID to retrieve the binary representation of the saved image (line 12)
- formats the binary object into a bitmap image (lines 13 - 20)
- generates a unique URL which can be used to access the file indirectly with a web browser and places it into a text box (line 22)
- creates a new Silverlight stack panel which contains the generated URL path in its text box and the saved image itself (lines 23- 26)
- creates a child window which pops up and displays the new stack panel (lines 27 - 28)

After editing the sample Scribble application, when it runs next, try the new screen capture functionality:

There will be a new **Save** button in the Silverlight client. Click on it; the application will open a view on top of the current window, with the URL to the file and a presentation of the JPEG image. Copy and paste this link in a new window to retrieve the screen capture; this will work as long as the session remains active. Close the new view using the X in the top right corner.

---

## Resource Distribution Options

Using the Resource Manager, nearly any file can be saved on the service and retrieved by any user who has been given the unique identifier for that resource, which gives developers a lot of flexibility with regards to how they distribute resources. Here are a few examples:

- **Single file to single client:** a single client requests a file; the service stores the file and sends the client its GUID key, which the client can then use to retrieve the requested file whenever desired (within the duration of the service application session).

- **Single file to selected clients:** this is basically the same scenario as for a single client, except the service application would also send the GUID key to other selected clients.
- **Any file to any client:** instead of simply assigning a GUID key to the files at time of storage, the service could write that GUID key into a descriptive path in the PureWeb application state, which would allow any and all clients to access the files.
- **Different files to different clients:** this would be useful, for instance, to handle operating system differences. Consider for example the case of a collaborative text editing application. When saving the file in the resource manager, the service would save it in two different formats, let's say a Windows-targeted .docx file and a .pages file for iOS-based clients. Each file format would be assigned its own GUID key; the key for the .docx files would be sent to the Windows clients, and the key to the .pages would be sent to the iOS-based client.

Chapter

# 10 Debugging

When developing PureWeb applications, the following tools are available to help with debugging:

- the debugging features of the development platform (Eclipse, Microsoft Visual Studio, etc.)
- PureWeb's Diagnostics Panel
- PureWeb logs

This chapter focuses on the Diagnostic Panel, but also provides some information on the other debugging options.

---

## Platform-Specific Debugging

PureWeb-enabled applications can attach to the debugging functionality of the development platform.

For example, below are the steps to follow in Xcode for debugging the iOS version of the sample Asteroids client:

1. Open Xcode and select the relevant project from the Welcome dialog.
2. Open the `AppDelegate.m` implementation file in the Classes folder and navigate to the `didFinishLaunchingWithOptions` function.
3. Click on the side panel beside the editor window to set a breakpoint at the desired location.
4. Click the **Run** icon in the top left-hand corner to run the client code and trigger the breakpoint.

Similar instructions are provided in the Debugging section of the *Quick Start Guide* for each of the supported development platforms.



---

# Diagnostics Panel

The Diagnostics Panel contains a set of built-in tools to help configure and troubleshoot PureWeb client applications during the development phase.

The panel allows developers, for instance, to change on the fly the image encoding configuration to help decide which one is optimal. This panel is also useful to see at a glance what happens within application state during user interactions with the client application.

## Adding a Diagnostics Panel to a Client

As discussed in chapter “Designing the Client Interface” on [page 47](#), when adding user interface elements to a PureWeb client, developers use primarily the native elements present on the client platform. The same applies when adding the Diagnostics Panel. For example, in HTML5, the panel is added as a div; in Silverlight, it is added as a grid.

### HTML5

```
1 <div id="pwDiagnosticsPanel"></div>
```

### Silverlight

```
//in header
1 xmlns:Diagnostics="clr-namespace:PureWeb.Client.Diagnostics;assembly=PureWeb.Client.Silverlight"

//in body of code
2 <Diagnostics:DiagnosticsView Grid.Row="0" Grid.Column="0"
  HorizontalAlignment="Stretch" VerticalAlignment="Stretch"/>
```

Most sample client applications include a Diagnostics Panel; review their code for examples of how this panel has been added. Sample applications are described in the *Quick Start Guide* for each supported client platform.

---

Note: Once the panel is added, how it is accessed will vary based on the platform. For example, in Silverlight or HTML5, add the `_diagnostics=true` parameter to the client application’s URL to view the panel. In the iOS client, a Diagnostics button will appear in the interface.

---

## Using the Diagnostics Panel

Once added, the panel will have four tabs or five tabs, depending on the programming language. This section describes the functionality available in each tab.

### Options Tab

The **Options** tab, shown below, is used to define options that impact image bandwidth and quality.

The screenshot shows the 'Options' tab of the Diagnostics Panel. It features three tabs: 'Options', 'Trace', 'App State', and 'Bandwidth'. The 'Options' tab is active. Below the tabs, there is a checkbox for 'Client side filtering' which is currently unchecked. Under the heading 'Encoding Configuration:', there are two sections: 'Interactive mode:' and 'Non-Interactive mode:'. Each section has a 'Mime type:' dropdown menu and a 'Quality:' slider. The 'Interactive mode' slider is set to 30 (0 - 100), and the 'Non-Interactive mode' slider is set to 70 (0 - 100). An 'Apply' button is located below the sliders. At the bottom, a note states: 'Note: changes will only apply to the currently visible view(s).'

### Client Side Filtering

When client-side filtering is enabled, the system automatically filters out duplicate or redundant commands sent from the client to the service, thereby reducing bandwidth. For example, when drawing in Scribble, if this option is enabled, PureWeb will send enough commands to the service to ensure the shape of the scribbles is displayed correctly, but will filter out the rest.

Client-side filtering is enabled by default and is set using the `<ClientCommandFiltering>` element in the application state. Depending on the client API, the value can be set to either true (enabled) or false, or it can be set to 0 (disabled) or 1.

The **Options** tab of the Diagnostics Panel provides a shortcut for this parameter, which eliminates the need to manually change it in application state. To enable or disable this feature, simply add or remove the checkmark in the **Client side filtering** checkbox.

## Encoding Configuration

In PureWeb applications, it is possible for the service to send images of a different quality and mime type when the user is interacting with a view, and when the user is not.

The main intent of this feature is to conserve bandwidth and reduce latency by sending images of lower quality during user interaction.

How this feature is implemented in the code is described in detail in the section “Defining Image Quality for Views” on [page 33](#).

The **Options** tab of the Diagnostics Panel provides a user-friendly visual interface for manipulating these encoding configuration parameters. It also allows developers to see on the fly the impact of their changes.

The values for Interactive mode and Non-interactive mode can be set completely independently of each other.

- **Mime type:** select a value from the drop-down box.
- **Quality:** slide the cursor to the right or to the left to set the quality of the images higher or lower. Accepted values range from 0 (lowest quality) to 100.

Click **Apply** to commit the changes. Interact with the view to see the impact of the changes.

## Trace Tab

The **Trace** tab will display any information that developers choose to trace within a client application. In the example below, the developer has chosen to trace color changes within the sample Scribble application.

	Options	Trace	AppState	Bandwidth	Profiling
Diagnostics Panel	<input checked="" type="checkbox"/> Autoscroll Trace	<input type="text" value="Reset Trace Size"/>	<input type="text" value="1000"/>		
	[149.783s] [Trace]	4/1/2013 10:26:42			
	Color is now:	SaddleBrown			
	[152.153s] [Trace]	4/1/2013 10:26:45			
	Color is now:	SkyBlue			
	[153.911s] [Trace]	4/1/2013 10:26:46			
	Color is now:	SteelBlue			
	[161.432s] [Trace]	4/1/2013 10:26:54			
	Color is now:	SkyBlue			
	[163.984s] [Trace]	4/1/2013 10:26:56			
	Color is now:	Tan			
	[166.377s] [Trace]	4/1/2013 10:26:59			
	Color is now:	Thistle			
[169.016s] [Trace]	4/1/2013 10:27:1				
Color is now:	Turquoise				
[173.312s] [Trace]	4/1/2013 10:27:6				
Color is now:	SeaShell				
[176.696s] [Trace]	4/1/2013 10:27:9				
Color is now:	Salmon				
[184.263s] [Trace]	4/1/2013 10:27:17				
Color is now:	PowderBlue				
[218.735s] [Trace]	4/1/2013 10:27:51				
Color is now:	Wheat				

By default, no information is traced. To set a trace, use the client API's `PureWeb.Diagnostics.Trace` namespace.

For example, the code for implementing the above trace in an HTML5 client would be as follows:

```
1 pureweb.client.diagnostics.trace("Color is now: " +
  e.getNewValue());
```

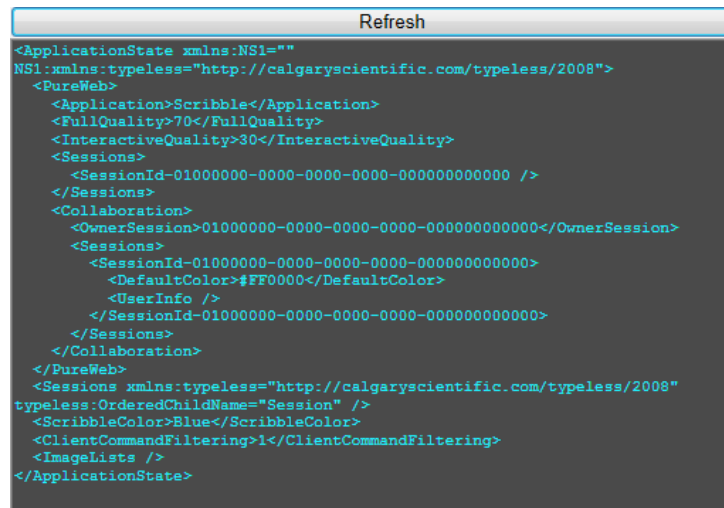
The Diagnostics Panel's **Trace** tab also offers some additional functionality:

- To change the trace size (number of traced transactions stored in memory), type the desired size in the text box provided.
- To reset the trace size back to the default value, click the **Reset Trace Size** button.
- If the trace size is a value larger than can be displayed in the Trace tab, the system will display a scroll bar on the side, which can be used to navigate to the bottom and see the latest lines added to the trace. When the **Autoscroll Trace** check box is selected, the system will automatically scroll to the bottom.

In addition to this trace feature, the PureWeb server and service application also write messages to log files, which are displayed on the server. For more information, see the Logs chapter of the *PureWeb Server Administration Guide*.

## AppState Tab

The **AppState** tab of the Diagnostics Panel allows developers to see the application state at a glance.



```
Refresh
<ApplicationState xmlns:NS1=""
NS1:xmlns:typeless="http://calgaryscientific.com/typeless/2008">
  <PureWeb>
    <Application>Scribble</Application>
    <FullQuality>70</FullQuality>
    <InteractiveQuality>30</InteractiveQuality>
    <Sessions>
      <SessionId>01000000-0000-0000-0000-000000000000 />
    </Sessions>
    <Collaboration>
      <OwnerSession>01000000-0000-0000-0000-000000000000</OwnerSession>
      <Sessions>
        <SessionId>01000000-0000-0000-0000-000000000000>
          <DefaultColor>#FF0000</DefaultColor>
          <UserInfo />
        </SessionId>01000000-0000-0000-0000-000000000000>
      </Sessions>
    </Collaboration>
  </PureWeb>
  <Sessions xmlns:typeless="http://calgaryscientific.com/typeless/2008"
typeless:OrderedChildName="Session" />
  <ScribbleColor>Blue</ScribbleColor>
  <ClientCommandFiltering>1</ClientCommandFiltering>
  <ImageLists />
</ApplicationState>
```

Since the application state is stored in memory only, this tab is the easiest way to quickly determine how user interactions in the client are reflected in application state.

Some changes will require a refresh before they are displayed in this tab.

## Bandwidth Tab

The **Bandwidth** tab of the Diagnostics Panel is used to test the network connection between the PureWeb server and the PureWeb service application to measure latency and bandwidth.

Test Iterations: 5 Test Latency

Payload Bytes: 60000 Test Bandwidth

Latency (min/avg/max): 0.000 / 0.000 / 0.000

Bandwidth (min/avg/max): 0.000 / 0.000 / 0.000

To test latency, enter the number of iterations that should be included in the test and click the **Test Latency** button. The system will return results that include minimum, average, and maximum latency.

To test bandwidth, enter the number of payload bytes and click the **Test Bandwidth** button. The system will return results that include minimum, average, and maximum bandwidth.

## Profiling Tab

The **Profiling** tab is available only in the Diagnostics Panel for HTML5 client applications.

Click the **Enable Profiling** checkbox to display a profiling report in XML format. This report contains measurements on how long it takes the PureWeb application to perform certain actions, such as building requests and parsing responses. An example of this report is shown below.

```

Enable Profiling: 
<Session-01000000-0000-0000-0000-000000000000>
  <View-ScribbleView>
    <Mbps>0.00</Mbps>
    <FPS>0.00</FPS>
    <Interaction>
      <DutyCycle>0.000</DutyCycle>
      <Frequency>0.0</Frequency>
    </Interaction>
    <EncodingType>image/jpeg;base64</EncodingType>
    <Renderer>ImageRenderer</Renderer>
  </View-ScribbleView>
  <XmlStateManager>
    <ProcessingChangeHandlers>
      <DutyCycle>0.003</DutyCycle>
      <Frequency>1.0</Frequency>
    </ProcessingChangeHandlers>
    <Merging>
      <DutyCycle>0.000</DutyCycle>
      <Frequency>0.0</Frequency>
    </Merging>
    <Diffing>
      <DutyCycle>0.009</DutyCycle>
      <Frequency>1.0</Frequency>
    </Diffing>
    <Getting>
      <DutyCycle>0.000</DutyCycle>
      <Frequency>1.0</Frequency>
    </Getting>
    <Setting>
      <DutyCycle>0.000</DutyCycle>
      <Frequency>0.0</Frequency>
    </Setting>
  </XmlStateManager>

```

# Index

## A

- application state
  - child changed handlers . . . . . 46
  - description . . . . . 12
  - get and set methods . . . . . 44
  - hierarchichal XML tree . . . . . 41
  - initializing . . . . . 42
  - keeping synchronized . . . . . 14
  - overview . . . . . 40
  - PureWeb element . . . . . 41
  - reading from state . . . . . 44
  - requesting a lock . . . . . 45
  - retrieving parsed values . . . . . 44
  - root element . . . . . 41
  - value changed handlers . . . . . 45
  - viewing in the Diagnostics Panel . . . . . 60
  - when to use . . . . . 41
  - with UI elements . . . . . 49
  - writing to state . . . . . 44

## C

- child changed handlers in application state . 46
- client
  - adding a command to a UI element . . . . . 48
  - adding a PureWeb layer to UI elements . . . 48
  - description . . . . . 10
  - feature set and appearance . . . . . 47
  - modify a UI element using application state. 49
  - supported languages . . . . . 10
  - UI elements and a platform's native
    - functionality . . . . . 48
- client side filtering . . . . . 58
- command
  - adding to a UI element . . . . . 48
  - description . . . . . 12
  - example with arguments and callback . . . 38
  - registering a UI handler . . . . . 37
  - response parameters for callback . . . . . 37
  - sending from client . . . . . 38

- setting up on the service . . . . . 37
- UI handler . . . . . 37
- when to use . . . . . 41
- communication flow . . . . . 13

## D

- debugging . . . . . 56
  - attaching to a debugger . . . . . 56
  - Diagnostics Panel . . . . . 57
- Diagnostics Panel . . . . . 57
  - adding to a client . . . . . 57
  - AppState tab . . . . . 60
  - Bandwidth tab . . . . . 61
  - client side filtering . . . . . 58
  - encoding configuration . . . . . 59
  - Options tab . . . . . 58
  - Profiling tab . . . . . 61
  - Trace tab . . . . . 59
  - using to measure bandwidth . . . . . 61
  - using to measure latency . . . . . 61

## E

- encoder configuration for images . . . . . 33
- encoding configuration
  - setting from the Diagnostics Panel . . . . . 59
- event
  - capturing a keyboard event . . . . . 30
  - capturing a mouse event . . . . . 31
  - converting touch-screen input to keyboard
    - events . . . . . 32
  - description . . . . . 13

## G

- getActualSize function . . . . . 26
- GUID key . . . . . 51

## I

## image

default encoding configuration . . . . .	34
defining encoding format . . . . .	33
defining quality . . . . .	33
encoder configuration . . . . .	33
interactive mode . . . . .	33
mime type . . . . .	33
non-interactive mode . . . . .	33
setting encoder configuration . . . . .	34
interactive image transmission mode . . . . .	33

## M

## mime type

JPEG . . . . .	34
PNG . . . . .	34
tiles . . . . .	34

## N

non-interactive image transmission mode . . . . .	33
---	----

## P

postKeyEvent function . . . . .	26, 29
postMouseEvent function . . . . .	26, 29
profiling . . . . .	61
PureWeb	
application enablement process . . . . .	16
PureWeb STK	
content . . . . .	8

## Q

queueCommand function . . . . .	38
---------------------------------	----

## R

RenderedView interface	
getActualSize function . . . . .	26
postKeyEvent function . . . . .	26
postMouseEvent function . . . . .	26
renderView function . . . . .	26
setClientSize function . . . . .	26
where to implement . . . . .	26
renderView function . . . . .	26
resource manager	
definition . . . . .	50
GUID key . . . . .	51
retrieving data from a client . . . . .	53
screen capture example . . . . .	51

storing data in the service . . . . .	52
workflow . . . . .	51

## S

## server

communication workflow . . . . .	20
description . . . . .	10
purpose . . . . .	10
setting up the connection with the client . . . . .	22
setting up the connection with the service . . . . .	21

## service

description . . . . .	10
graceful disconnect . . . . .	22
supported languages . . . . .	10
setClientSize function . . . . .	26

## T

## touch-screen devices

converting user input to keyboard events . . . . .	32
tracing information . . . . .	60

## V

value changed handlers in application state . . . . .	45
---	----

## view

defining encoding format . . . . .	33
defining image quality . . . . .	33
description . . . . .	11
displaying in a client . . . . .	29
generic handling class . . . . .	26
handling user input . . . . .	29
keeping up-to-date . . . . .	14
registering in the service . . . . .	27
remoting a service view . . . . .	26
RenderedView interface . . . . .	26
sample implementation in a service . . . . .	27