

BlackBerry Java SDK

Network Communication

Version: 6.0 Beta

Development Guide

Contents

1	Network communication overview.....	3
2	Web services.....	4
	Communicating with web services.....	5
	Request data using a BlockingSenderDestination object.....	6
	Request data using a NonBlockingSenderDestination object.....	7
	Send data using a FireAndForgetDestination object.....	10
	Parsing a common Internet data format.....	11
	Subscribing to a push initiator.....	11
	Subscribe to a push initiator.....	12
	Sending login information to a web service.....	13
3	Radios and transports.....	14
	Working with radios.....	14
	Query a radio's availability and change its status.....	14
	Architecture: Network transports.....	15
	Using the BlackBerry Enterprise Server as an intranet gateway.....	18
	Using the wireless service provider's Internet gateway.....	18
4	Network connections.....	20
	The Network API.....	21
	Choosing network transports using the Network API.....	21
	Opening a network connection using the Network API.....	22
	Generic Connection Framework.....	24
	Send and receive data using a network connection.....	26
	Network transport options.....	29
5	Code samples.....	31
	Code sample: Requesting data using a BlockingSenderDestination object.....	31
	Code sample: Requesting data using a NonBlockingSenderDestination object.....	33
	Code sample: Parsing a JSON data structure.....	36
	Code sample: Retrieving a web page using the Network API.....	41
	Code sample: Retrieving a web page using the GCF.....	45
	Code sample: Determining network transports with sufficient coverage using the Network API.....	48
	Code sample: Controlling radios.....	50

Code sample: Determining the status of a network transport using the Network API.....	52
Code sample: Retrieving the service sook record for WAP 2.0.....	54
6 Glossary.....	56
7 Provide feedback.....	59
8 Legal notice.....	60

Network communication overview

A BlackBerry® device uses radio communication technologies such as Wi-Fi® technology, CDMA or GPRS to communicate over wireless connections. BlackBerry devices work with different network transports over those radio technologies to connect to the Internet or a private network. Each transport offers different advantages. As a developer, you should choose the transport that is most appropriate for your requirements.

When you choose a transport for your application, you should consider a number of different factors including:

- Type of user
- Importance of security and reliability
- Networks that the application must use, and support for roaming users
- Amount of data you expect to send and receive
- Need to proactively push data for updates and alerts
- Location of the destination server (Internet or intranet)

The BlackBerry® Java® SDK 6.0 includes three approaches to open a network connection over a transport.

API	Description
Communication API	The Communication API encapsulates the all the operations required to connect to a web service that is located on the Internet or behind an organization's firewall.
Network API	The Network API provides a simple interface to many connection-related options and network transports.
Generic Connection Framework	The Generic Connection Framework (GCF) provides a flexible way to create network connections using the transport of your choice.

In addition to the functionality each API provides, you should consider whether the API exists on the version of the BlackBerry® Device Software that your target devices are likely to run. For example, the Communication API requires BlackBerry Device Software 6.0. You would need to write your own code to perform similar tasks on devices that run BlackBerry Device Software 5.0 and earlier.

The BlackBerry solution also enables server-side applications to proactively push data to BlackBerry devices in a highly secure and reliable manner.

Web services

The Communication API simplifies the process of interacting with web services and other applications. The objects created by this API automate the process of finding an available network transport, creating a thread-safe connection, and negotiating an exchange of data with the URI or URL end point that you specify. The API is implemented in the `net.rim.device.api.io.messaging` package.

Within the Communication API, URI and URL end points are called destinations. You can use objects that implement the `SenderDestination` interface send data to a web service and access the response. Similarly, objects that implement the `ReceiverDestination` interface subscribe to a push service and provide access to new messages from that service. You must use the `DestinationFactory` class to create an object that implements one of the `Destination` sub-interfaces that suits your requirements.

Interface	Description
<code>BlockingSenderDestination</code>	You can use this interface to send a message to a web service, but this object will block thread execution while it waits for a response from the web service. Alternatively, <code>BlockingSenderDestination</code> can return a <code>MessageFuture</code> object. This allows thread execution to continue until you invoke one of the <code>get()</code> methods from the <code>MessageFuture</code>. When you invoke <code>sendReceive()</code> in <code>BlockingSenderDestination</code> to send a message and wait for a response, you should not invoke it from the main event thread.
<code>NonBlockingSenderDestination</code>	You can use this interface to send a message to a web service. In the parameter list, pass your implementation of the <code>MessageListener</code> interface to receive the response from a web service.
<code>BlockingReceiverDestination</code>	You can use this interface to subscribe to a push service and block thread execution until you receive confirmation of your subscription request.
<code>NonBlockingReceiverDestination</code>	You can use this interface to subscribe to a push service. In the parameter list, pass your implementation of <code>MessageListener</code> to receive push messages.
<code>FireAndForgetDestination</code>	You can use this interface to send data to a web service when you don't expect a response.

Data sent to and received from a web service is wrapped in a `Message` object. You can set the headers and content of the data sent to a web service in the `Message` object. Alternatively, you can allow the Communicatino API to choose reasonable default values for your request. For example, you can request a web page by invoking `SenderDestination.send()` without specifying an HTTP GET command.

You can also instruct the API to automatically parse the response data from a web service. If you implement the `MessageProcessor` interface with your parser class, you can supply your parser object to the method in `DestinationFactory` that creates your `Destination` object. You can then retrieve the parsed response using `Message.getObjectPayload()`.

The `net.rim.device.api.io.parser` packages contain message processors for a variety of standard data formats used on the internet. The packages include parsers for:

- JSON
- RSS (RDF, Atom)
- SOAP
- XML

To improve efficiency, the Communication API provides large responses from web services in an `InputStream` object. An application can request additional data as required rather than downloading a large amount of data at once and storing it in memory on the device. This enables you to make better use of both network bandwidth and memory.

Communicating with web services

To send a command to a web service, you need a `Destination` object to send a message. The `DestinationFactory` class creates and maintains a list of `Destination` objects that you can use to communicate with an end point. Before you create a `Destination`, you should check whether one exists by invoking `getSenderDestination()`. You can access an existing destination by providing the name of the `Context` object that you supplied when you invoked any of the `DestinationFactory.create...()` methods. When you finish exchanging data with an end point, you should invoke `Destination.release()`. The `DestinationFactory` removes any resources associated with a destination.

A message contains the details of your command, including the HTTP verb and any additional data you require. Some `Destination` objects choose reasonable default values for your message if you do not specify one.

After you send your message, you may need to listen for a response. For `BlockingSenderDestination` objects, you need to create a `Thread` object when you invoke one of the `sendReceive()` methods. For `NonBlockingSenderDestination` objects, you must create a `MessageListener` object to receive the response.

In either case, you need to process a `Message` object that contains the response. By default, the body of the `Message` contains the raw response data. You can choose to specify a message processor from the `net.rim.device.api.io.parser` package, or create your own using the `MessageProcessor` interface.

You can connect multiple message processors together if necessary. The `MessageProcessorChain` class shares memory between the `MessageProcessors` objects to improve efficiency. For example, if you receive video data that uses custom encoding and compression, you can separate the decoding and decompression logic into separate message processors, and then use `MessageProcessorChain` to group them together.

Request data using a BlockingSenderDestination object

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.io.URI;
```

2. Create a thread from which to call `BlockingSenderDestination.sendReceive()`.

```
Thread t = new Thread( new Runnable()
{
    public void run()
    {
```

3. Create a `Message` object to hold the response from the URL.

```
Message response = null;
```

4. Create a `URI` object to pass to the `DestinationFactory` class.

```
URI uri = new URI("http://www.blackberry.com");
```

5. Create a `BlockingSenderDestination` object.

```
BlockingSenderDestination bsd = null;
```

6. Retrieve the `BlockingSenderDestination` object for your context, if one exists.

```
try
{
    bsd = (BlockingSenderDestination)
        DestinationFactory.getSenderDestination
            ("MyContext", uri);
```

7. If no `BlockingSenderDestination` exists, create one.

```
if(bsd == null)
{
    bsd = DestinationFactory.createBlockingSenderDestination
        (new Context("MyContext"), uri);
}
```

8. Send a message and wait for the response.

```
response = bsd.sendReceive();
```

9. If the web service sent a response, process the response.

```
if(response != null)
{
    processResponse(response);
}
```

10. Catch any errors that occur if the message could not be sent for some reason.

```
catch (Exception e)
{
    // Process the error
}
```

11. Release the `BlockingSenderDestination`.

```
finally
{
    if(bsd != null)
    {
        bsd.release();
    }
}

});
```

12. Start the thread.

```
t.start();
```

After you finish: If your `processResponse()` updates the UI, you must do so on the main event thread, not on the thread you created in this task. For more information, read the knowledge base article at <http://supportforums.blackberry.com/t5/Java-Development/Manage-UI-interactions/ta-p/502378>.

Request data using a `NonBlockingSenderDestination` object

Before you begin: To request data using the `NonBlockingSenderDestination` class, you must create an object that implements the `MessageListener` interface. For more information, see "Implement the `MessageListener` interface".

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.io.URI;
```

2. Create a `Message` object to hold the response from the URL.

```
Message response = null;
```

3. Create a `URI` object to pass to the `DestinationFactory` class.

```
URI uri = new URI("http://www.blackberry.com");
```

4. Create a `NonBlockingSenderDestination` object.

```
NonBlockingSenderDestination nbsd = null;
```

5. Create a reference to your `MessageListener`.

```
MyMessageListener msgListener = new MyMessageListener();
```

6. Retrieve the `NonBlockingSenderDestination` object for your context, if one exists.

```
try
{
    nbsd = (BlockingSenderDestination) DestinationFactory.getSenderDestination
        ("MyContext", uri);
```

7. If no `NonBlockingSenderDestination` exists, create one.

```
    if(nbsd == null)
    {
        nbsd = DestinationFactory.createNonBlockingSenderDestination
            (new Context("MyContext"), uri, msgListener);
    }
```

8. Send a message to the web service.

```
nbsd.send();
```

9. Catch any errors that occur if the message could not be sent for some reason.

```
catch (Exception e)
{
    // Process the error
}
```

After you finish:

Release the `NonBlockingSenderDestination` when you are done processing the response.

Implement the `MessageListener` interface

You can use the `MessageListener` to receive messages sent by a `NonBlockingSenderDestination` object, or push messages from a push initiator.

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.messaging.*;

import java.io.IOException;
import java.io.InputStream;
```

2. Define a class that implements the `MessageListener` interface.

```
public class MyMessageListener implements MessageListener {
```

3. Implement `onMessage()`.

```
    public void onMessage(Destination dest, Message incomingMessage)
    {
```

4. Initialize a `String` variable to hold the response data.

```
String payload = null;
```

5. If the response is a `ByteMessage` object, retrieve the response as a `String` and assign it to `payload`.

```
if (incomingMessage instanceof ByteMessage)
{
    ByteMessage reply = (ByteMessage) incomingMessage;
    payload = (String) reply.getStringPayload();
}
```

6. If the response is a `StreamMessage`, retrieve the response as an `InputStream` object.

```
else if (incomingMessage instanceof StreamMessage)
{
    StreamMessage reply = (StreamMessage) incomingMessage;
    InputStream is = reply.getStreamPayload();
```

- a. If the response is small, convert the contents of the stream into a byte array.

```
byte[] data = null;
try
{
    data = net.rim.device.api.io.IOUtilities.streamToBytes(is);
} catch (IOException e)
{
    // process the error
}
```

- b. If the conversion was successful, convert the byte array to a `String` and assign it to `payload`.

```
if (data != null)
{
    payload = new String(data);
}
}
```

7. If `payload` contains data, display it.

```
if (payload != null)
{
    synchronized (Application.getEventLock())
    {
        UiApplication.getUiApplication().pushScreen
            (new HTTPOutputScreen(payload));
    }
}
```

After you finish:

Implement `onMessageCancelled()` and `onMessageFailed()` to process other notifications.

Send data using a FireAndForgetDestination object

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.io.URI;
```

2. Create a URI object to pass to the DestinationFactory class.

```
URI uri = new URI("http://www.example.com");
```

3. Create a FireandForgetDestination object.

```
FireandForgetDestination ffd = null;
```

4. Retrieve the FireandForgetDestination object for your context, if one exists.

```
try
{
    ffd = (FireandForgetDestination) DestinationFactory.getSenderDestination
        ("MyContext", uri);
}
```

5. If no FireandForgetDestination exists, create one.

```
if(ffd == null)
{
    ffd = DestinationFactory.createFireandForgetDestination
        (new Context("MyContext"), uri);
}
```

6. Create a ByteMessage object and populate it with information to send to a web service.

```
ByteMessage myMsg = ffd.createByteMessage();
myMsg.setStringPayload("I love my BlackBerry device!");
```

7. Cast your message as an HTTPMessage to set the HTTP method you want to use.

```
((HttpMessage) myMsg).setMethod(HttpMessage.POST);
```

8. Send the message to the web service.

```
ffd.sendNoResponse(myMsg);
```

9. Catch any errors that occur if the message could not be sent for some reason.

```
catch (Exception e)
{
    // Process the error
}
```

Parsing a common Internet data format

You can configure a `Destination` object to return a Java® object that contains data parsed from a standard format, such as JSON. To retrieve the data structure in a Java object, you should:

1. Specify a `MessageProcessor` when you invoke `DestinationFactory.create...()`.
2. Retrieve the response from your `Destination`.
3. Extract the data from the `Message` object that is returned in the response.

Code sample

The following code sample demonstrates the three actions required to retrieve a Java object that contains a JSON data structure.

```
import net.rim.device.api.io.messaging.*
import net.rim.device.api.io.parser.json.*

// Specify a MessageProcessor when you create your Destination
BlockingSenderDestination bsd = DestinationFactory(myContext, myURI,
                                                    new JSONMessageProcessor());

// Retrieve the response
Message response = bsd.sendReceive();

// Extract the Java object that contains the JSON data structure
Object payload = response.getObjectPayload();
```

Subscribing to a push initiator

You can use the Communication API to send a subscription request to a content provider's URI. In addition to the `SenderDestination` object that sends the subscription request, you need to create a `ReceiverDestination` object that defines the `MessageListener` object to which the BlackBerry® device should deliver incoming push messages from a given URI.

The following table describes the components involved in a subscription request.

Component	Description
SenderDestination	This component sends the subscription message to the content provider. This component is optional. You can send a subscription message using any network connection.

Component	Description
Subscription Message	The Communication API can create a properly formed subscription message for you. Invoke <code>BpsSubscriptionMessageBuilder.createByteSubscriptionMessage()</code> to retrieve a <code>ByteMessage</code> object configured for your <code>SenderDestination</code> , <code>NonBlockingReceiverDestination</code> , and the <code>MessageListener</code> where you want to receive incoming push messages.
<code>NonBlockingReceiverDestination</code>	This component associates a particular URI with a <code>MessageListener</code> where you want to receive incoming push messages, and an <code>InboundDestinationConfiguration</code> object.
<code>InboundDestinationConfiguration</code>	This component specifies a variety of push service parameters, such as an indicator that specifies whether your application should be started when a new push message arrives. You can create an <code>InboundDestinationConfiguration</code> using the <code>InboundDestinationConfigurationFactory</code> class.
<code>MessageListener</code>	This component processes incoming push messages.

Subscribe to a push initiator

This task focuses on creating a subscription message and the `NonBlockingReceiverDestination` object for a BlackBerry® Push Service subscription. You must send your subscription message using a `SenderDestination` object, or by creating a network connection manually.

Before you begin: You must obtain an application ID, the port on which you should listen for incoming messages. These details are provided to the content provider that creates a server-side push application.

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.messaging.*;
```

2. Initialize a `NonBlockingReceiverDestination` variable to manage the subscription.

```
NonBlockingReceiverDestination nbrd = null;
```

3. Create a `MessageListener` object to process incoming push messages.

```
try
{
    MyMessageListener pushListener = new MyMessageListener();
```

4. Create a `URI` object with the port information you were assigned by the BlackBerry Push Service content provider.

```
pushURI = URI.create("http://:101");
```

5. Create an `InboundDestinationConfiguration` to set parameters for the push subscription.

```
InboundDestinationConfiguration config =
    InboundDestinationConfigurationFactory.createBPSConfiguration
        (true, // start this application when a push message arrives
         false, // allow other applications to receive these push messages
         false, // do not store these messages (persistence not supported)
         "95-Ry313fBlrM46m761", // application ID,
         BPSuri); // BlackBerry Push Service URI
```

6. Create a `NonBlockingReceiverDestination`.

```
nbrd = DestinationFactory.createNonBlockingReceiverDestination
    (config, pushURI, pushListener);
```

7. Create a `SenderDestination` to send a subscription message.

```
bpsDestination = DestinationFactory.createNonBlockingSenderDestination
    (myContext, uri, responseListener);
```

8. Configure a subscription message.

```
ByteMessage subMsg =
    BpsSubscriptionMessageBuilder.createByteSubscriptionMessage
        (bpsDestination, nbrd, "user", "pwd");
```

9. Send the subscription message.

```
bpsDestination.send();
```

Sending login information to a web service

A `Destination` object can supply login information, such as a user name and password, when a web service requests it. Login information is stored in the `Context` object you supply when you create a `Destination` using the `DestinationFactory` class.

Code sample

```
context = new Context("DemoContext", new CredentialsCollector() {
    public UsernamePasswordCredentials getBasicAuthenticationCredentials
        (String aeID, Hashtable properties) {
        // Provide or retrieve authentication credentials here.
        // For example, you could display a dialogue box to ask your user
        // to enter their username and password, then return it from
        // this method.
        return new UsernamePasswordCredentials("username", "password");
    }
});
```

Radios and transports

Working with radios

There may be cases where you need to activate a radio on a BlackBerry® device to use a particular transport. For example, you may want your BlackBerry device application to connect explicitly over Wi-Fi® to make efficient use of bandwidth. Or, you may want to compare your application's performance when you use different wireless technologies.

You can use the `net.rim.device.api.system.RadioInfo` class to gather information about the wireless technologies available on a device. Using the `rim.device.api.system.Radio` class, you can manage the active radios on the device. You should also implement the `net.rim.device.api.system.RadioStatusListener` interface to receive notifications about changes in radio status.

The wireless technologies that are available on BlackBerry devices are grouped into three wireless access families. The following table describes those groups:

Wireless access family	Description
3GPP®	includes GPRS, EDGE, UMTS® GERAN, UTRAN, and GAN
CDMA	includes CDMA1x and EVDO
WLAN	includes 802.11™, 802.11a, 802.11b, 802.11g

Query a radio's availability and change its status

You can use the `RadioInfo` and `Radio` classes to determine whether a particular radio technology is supported on a BlackBerry® device, and to turn on or turn off the radio. For more information, see the API reference for the BlackBerry Java® SDK.

1. Import the required classes.

```
import net.rim.device.api.system.Radio;
import net.rim.device.api.system.RadioInfo;
```

2. Retrieve a bit field of supported wireless access families.

```
int intSupportedWAFs = RadioInfo.getSupportedWAFs();
```

3. Check if the wireless access family you want to use is available on the device. Perform a logical AND operation that combines the `intSupportedWAFs` bit field and the constant for the wireless access family you want to use. If the family is available the result will be non-zero.

```
if ((intSupportedWAFs & RadioInfo.WAF_WLAN) != 0 )
{
```

4. Retrieve a bit field of active radios,

```
int intActiveWAFs = RadioInfo.getActiveWAFs();
```

5. Check if the radio you want to use is turned on. If it is off, a logical AND operation that combines `intActiveWAFs` and the constant for the radio you want to use will return 0.

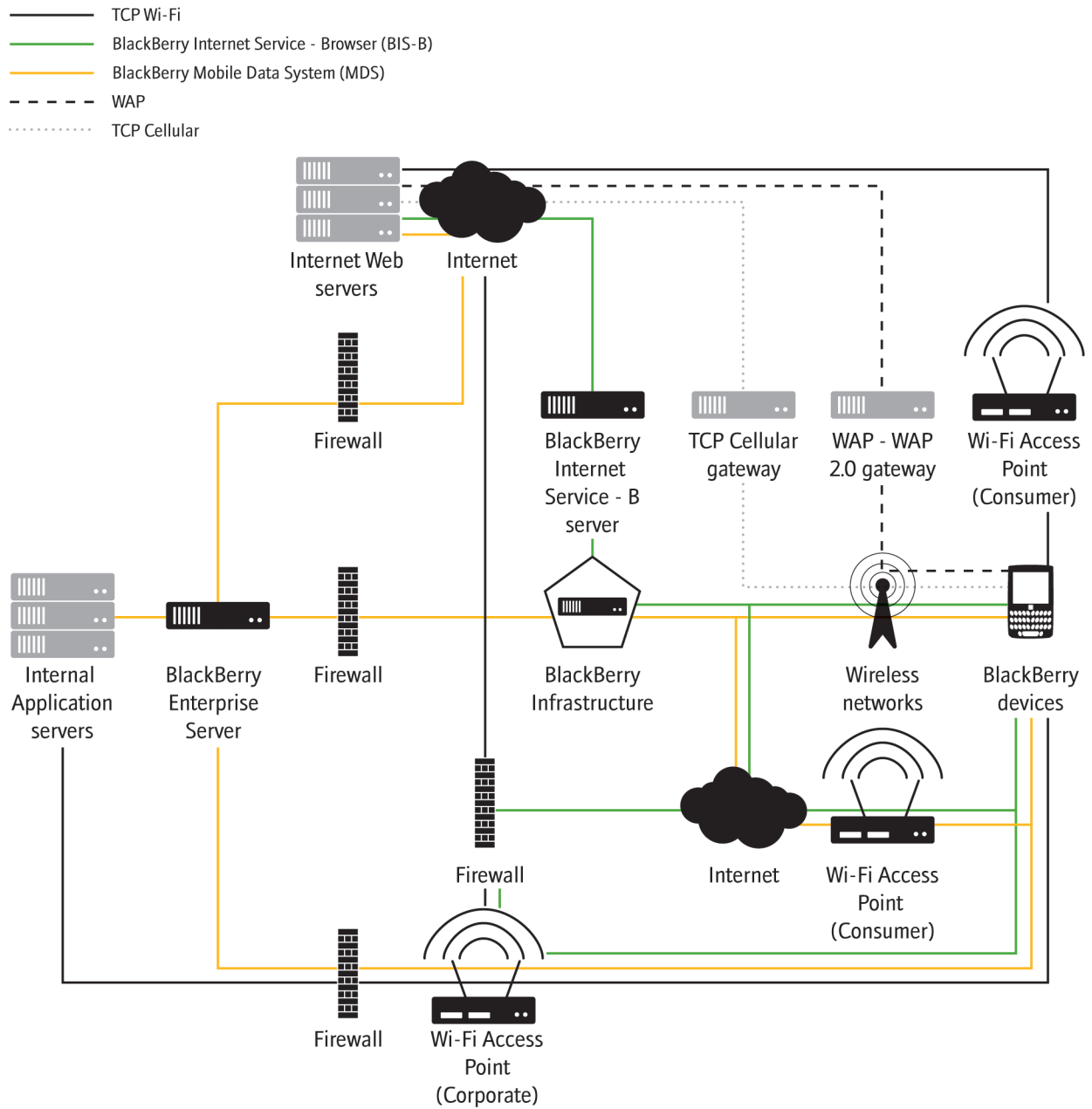
```
if ((intActiveWAFs & RadioInfo.WAF_WLAN) == 0)
{
```

6. If the radio is turned off, then turn it on.

```
    Radio.activateWAFs(RadioInfo.WAF_WLAN);
}
```

Architecture: Network transports

A BlackBerry® device can connect to a wireless network using different transports. Not all devices have access to all transports. For example, the BlackBerry® Mobile Data System is only available to devices associated with a BlackBerry® Enterprise Server.



Network transport	Description
Wi-Fi®	The Wi-Fi transport creates a connection to the Internet, or to a private network such as those in an organization or home. When the Wi-Fi radio is turned on, a BlackBerry device user or an application can configure the device to connect to the Internet using this transport. If appropriate, the device may automatically send data for the BlackBerry Enterprise Server or BlackBerry® Internet Service over a Wi-Fi connection. For example, if the device is connected to a Wi-Fi network and a VPN profile is configured for the user's organization, the device will attempt to connect to the BlackBerry Enterprise Server over the VPN using the Wi-Fi connection. This behavior is known as least cost routing.
TCP Cellular	The TCP Cellular transport creates a connection to the Internet through a wireless service provider's Internet gateway. This method offers the most direct type of connection over the cellular radio. Most wireless service providers configure a user's BlackBerry device to use the provider's Internet gateway. However, when the user roams on a different network they need to manually configure the device to use the Internet gateway on the host network. If your application uses this transport, you should test it for each wireless service provider that your application might use.
BlackBerry Internet Service	The BlackBerry Internet Service transport creates a connection to the Internet through the BlackBerry Infrastructure. Data that you send and receive using this transport is compressed and optimized for transmission over wireless connections. Applications that use the BlackBerry Internet Service transport automatically benefit from least cost routing. The device automatically connects over a Wi-Fi network, if available. Note: To use the BlackBerry Internet Service, you must sign up for the BlackBerry® Push Service. For more information, visit http://na.blackberry.com/eng/partners/.
BlackBerry MDS	The BlackBerry MDS transport enables a BlackBerry device to connect to its associated BlackBerry Enterprise Server through the BlackBerry Infrastructure or a Wi-Fi connection.

Network transport	Description
WAP	For more information, see "Using the BlackBerry Enterprise Server as an intranet gateway". The WAP transport creates a connection through a wireless service provider's WAP gateway. WAP 1.0, 1.2, and 2.0 are supported. To support this transport, a user or wireless service provider must configure the WAP connection parameters on the device. As a result, this transport might not be supported on all wireless networks or with all data plans. If your application uses this transport, you should test it for each wireless service provider that your application might use. For more information, see "Using the wireless service provider's Internet gateway".

Using the BlackBerry Enterprise Server as an intranet gateway

Organizations host the BlackBerry® Enterprise Server behind their firewall to allow BlackBerry device users to access the organization's intranet. The BlackBerry® Mobile Data System component of the BlackBerry Enterprise Server includes the BlackBerry® MDS Services, which provides an HTTP and TCP/IP proxy service to let third-party Java® applications use it as a secure gateway for managing HTTP and TCP/IP connections to the intranet. When you use the BlackBerry Enterprise Server as an intranet gateway, all data traffic between your application and the BlackBerry Enterprise Server is automatically encrypted using AES encryption or Triple DES encryption. Because the BlackBerry Enterprise Server resides behind the organization's firewall and provides inherent data encryption, applications can communicate with application servers and web servers that reside on the organization's intranet.

If your application connects to the Internet rather than to the corporate intranet, you might be able to use the BlackBerry Enterprise Server that the device is associated with as a gateway. You might be able to use the one that belongs to the customer as a gateway. In this case, network requests travel behind the corporate firewall to the BlackBerry Enterprise Server, which makes the network request to the Internet through the corporate firewall. However, organizations can set an IT policy to enforce that the BlackBerry Enterprise Server is the gateway for all wireless network traffic, including traffic destined for the Internet.

Using the wireless service provider's Internet gateway

Most wireless service providers provide an Internet gateway that offers direct TCP/IP connectivity to the Internet. Some wireless service providers also provide a WAP gateway that allows HTTP connections to use the WAP protocol. A BlackBerry® Java® Application can use either of these gateways to connect to the Internet. If your application is for BlackBerry® device users who are on a specific wireless network, using the wireless service provider's Internet gateway can often yield fast, and reliable

connections. If your application is for BlackBerry device users on a variety of wireless networks, testing your application against the different Internet gateways can be challenging. In this case, you should use the BlackBerry® Internet Service, and use the wireless service provider's Internet gateway as a backup connection type if the BlackBerry Internet Service is not available.

Network connections

The BlackBerry® Application Platform offers two APIs to create network connections manually: the Network API and the GCF. Applications that target BlackBerry® devices running BlackBerry® Device Software 5.0 and later can use the Network API. The Network API provides a simple interface for working with network transports, and setting a wide variety of parameters for a connection. You can use the GCF to create network connections on devices running older BlackBerry Device Software. However, using the GCF requires knowledge of more BlackBerry® APIs to discover what transports are available and how to configure them.

Regardless of which API you choose, the process of creating a network connection is similar. First, you open a network connection, then read and write data over that connection, and finally close the connection. Before you open a connection, you usually specify a transport, a protocol, and an end point. You can connect to network resources using a wide range of protocols. However, you should be aware of the limitations of particular transports to support each protocol. For example, WAP 1.0 and 1.1 only support HTTP over WAP and HTTPS over WAP/WTLS. The BlackBerry Application Platform supports the following protocols:

- HTTP
- HTTPS
- Socket
- TLS
- SSL
- UDP (Datagram)

The protocol and the end point are determined by your application, but the transport is determined by your users' operating environment. You may need to try more than one transport before you can make a successful connection. The following table suggests preference orders for enterprise and consumer applications.

Enterprise	Consumer
BlackBerry® Mobile Data System	TCP Wi-Fi
TCP Wi-Fi®	WAP 2.0
WAP 2.0	BlackBerry Internet Service
BlackBerry® Internet Service	BlackBerry MDS
TCP Cellular	TCP Cellular
WAP 1.0 or 1.1	WAP 1.0 or 1.1

After you open a connection, you receive a `Connection` object that represents your connection. The `javax.microedition.io` package contains multiple interfaces that you can use to manipulate a `Connection` object.

Interface	Description
<code>HttpConnection</code>	This interface provides methods to set the HTTP request method (GET, POST, and so on) and headers, and send and receive data over the connection.
<code>HttpsConnection</code>	This interface includes all of the methods in the <code>HttpConnection</code> interface, and adds <code>getSecurityInfo()</code> that returns the certificate supplied by the web server.
<code>SocketConnection</code>	This interface exposes methods to send data to and receive data from a network host over a socket connection.
<code>SecureConnection</code>	This interface provides methods to create TLS and SSL socket connections. This interface includes all of the methods in the <code>SocketConnection</code> interface, and adds a <code>getSecurityInfo()</code> method that returns security information about the connection.
<code>UDPDatagramConnection</code>	This interface provides methods to retrieve information about the connection, and create new datagram objects.
<code>InputConnection</code> , <code>OutputConnection</code>	These interfaces provide access to a connection's input and output data streams.

When you are ready to use your connection, cast the `Connection` object according to the protocol you used to open the connection. You will need the `InputConnection` and `OutputConnection` interfaces from the `javax.microedition.io` package to access the `Connection` object's send and receive functions. Those interfaces expose the `Connection` object's input and output streams. For more information about sending data over the connection, see the `InputStream` and `OutputStream` classes in the BlackBerry® Java® SDK.

The Network API

The Network API is implemented in the `net.rim.device.api.io.transport` and `net.rim.device.api.io.transport.options` packages. Most of the functionality is provided by the `ConnectionFactory` and `TransportInfo` classes. A `ConnectionFactory` object returns a `ConnectionDescriptor` object that contains a `Connection` object and information about the transport over which the connection was opened.

Choosing network transports using the Network API

In the Network API, the `TransportInfo` class provides information about the network transports available on a device. You can retrieve a list of transports that currently have coverage, or use the API to determine whether a particular transport is available and has coverage.

If you don't provide an ordered list of preferred transports when you open a connection, the Network API chooses a transport for you. However, you should prioritize the transports your application uses based on the type of BlackBerry® device user your application targets. The API steps through an array of transport types that you set using `ConnectionFactory.setPreferredTransportTypes()`.

After you list your preferred transports, you should set options for any transports that require them. For example, the WAP 1.0 and 1.1 transports require information about an APN, the gateway authentication, and so on. For more information, see "Network transport options".

You should also create a `CoverageStatusListener` object (in the `net.rim.device.api.io.transport` package), and register it using `TransportInfo.addListener()`. The BlackBerry device notifies your application about changes in coverage status so that it can respond appropriately.

Opening a network connection using the Network API

After you decide which network transports to use, you can open a connection and use it to send and receive data. The `ConnectionFactory` object is responsible for opening connections. In addition to the preferred transport types, you can use the `ConnectionFactory` to configure connection options such as:

- maximum number of connection attempts.
- maximum time to spend making connection attempts.
- delay between connection attempts.
- whether or not end-to-end encryption is required, or even desired.

When you are ready to open a connection, invoke one of the `ConnectionFactory.getConnection()` methods. Be sure that you call `getConnection()` on a thread separate from the main event thread. For more information about creating a thread, see the `Thread` class in the API reference for the BlackBerry® Java® SDK.

When you open a connection successfully, the `ConnectionFactory` returns a `ConnectionDescriptor` object. The `ConnectionDescriptor` contains a `Connection` object, and information about the transport used to open the connection.

For more information, see "Open a network connection using the Network API" and "Send and receive data using a network connection".

Open a network connection using the Network API

CAUTION: The `ConnectionFactory` class is not thread-safe. The `getConnection()` method blocks thread execution. You should create a separate thread to call `getConnection()`.

1. Import the required classes and interfaces.

```
import net.rim.device.api.io.transport.*;
import net.rim.device.api.io.transport.options.*;
import net.rim.device.api.io.transport.TransportInfo;
import net.rim.device.api.ui.UiApplication;
```

2. Create an ordered list of preferred transports that you want to use.

```
int[] intTransports =
{
    TransportInfo.TRANSPORT_TCP_WIFI,
    TransportInfo.TRANSPORT_WAP2,
    TransportInfo.TRANSPORT_TCP_CELLULAR
}
```

3. Configure the options for the TCP Cellular transport, if applicable.

```
TcpCellularOptions tcpOptions = new TcpCellularOptions();
if(!TcpCellularOptions.isDefaultAPNSet())
{
    tcpOptions.setApn("My APN");
    tcpOptions.setTunnelAuthUsername("user");
    tcpOptions.setTunnelAuthPassword("password");
}
```

4. Create a `ConnectionFactory` object.

```
ConnectionFactory factory = new ConnectionFactory();
```

5. Set any other `ConnectionFactory` options that are applicable to your application. In this case, TCP Cellular is one of the preferred transports, so set the TCP Cellular transport options.

```
factory.setTransportTypeOptions(TransportInfo.TRANSPORT_TCP_CELLULAR, tcpOptions);
factory.setAttemptsLimit(5);
```

6. Create a thread to retrieve the connection. If the connection was successful then `ConnectionFactory` returns a `ConnectionDescriptor` object that you can use. In this step, pass the `ConnectionDescriptor` object to another method (`displayContent()`) that is responsible for displaying the content.

```
Thread t = new Thread(new Runnable()
{
    public void run()
    {
        ConnectionDescriptor cd = _factory.getConnection("http://
www.blackberry.com");
        if(cd != null) {
            Connection c = cd.getConnection();
            displayContent(c);
        }
    }
});
t.start();
```

7. Create `displayContent()`. In this case, push a screen that uses a `Connection` parameter to retrieve and display the content, after the connection retrieval thread completes.

```
private void displayContent(final Connection conn)
{
    UiApplication.getUiApplication().invokeLater(new Runnable()
```

```
{
    public void run() {
        UiApplication.getUiApplication().pushScreen(new HTTPOutputScreen(conn));
    }
};
}
```

After you finish: For more information about using a connection, and to implement the `HTTPOutputScreen` class, see "Send and receive data using a network connection".

Generic Connection Framework

The GCF is implemented in the `javax.microedition.io.Connector` class. You can call `Connector.open()` to create any type of supported connection. The `open()` method accepts a connection string that specifies the type of connection to make, the end point and optional configuration details for the connection.

The connection string used by the GCF comprises three parts:

<scheme>: <hierarchical location information>; <additional information>

The *<scheme>* indicates the protocol to use to establish the connection. The *<hierarchical location information>* describes the end point for the connection. Finally, the *<additional information>* parameter controls the transport used, and provides authentication details for a proxy server if they are required.

For example, in the following code sample, the connection string specifies the HTTP protocol and uses an host name to indicate the connection end point. The `interface=wifi` parameter is included to specify that the Wi-Fi® radio should be used to establish the connection.

```
HTTPConnection c = (HTTPConnection)Connector.open("http://
example.com;interface=wifi");
```

Open a network connection using the GCF

The following task shows you how to open an HTTP connection using the BlackBerry® Mobile Data System transport. You can use a similar process to open a connection using other protocols and transports.

CAUTION: The `Connector.open()` method is not thread-safe. Ensure that you invoke `open()` on a thread separate from the main event thread.

Before you begin: Make sure that the transport you want to use is available and has coverage.

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.UiApplication;
import java.io.IOException;
import javax.microedition.io.*;
```

2. Create a new thread to open the connection.

```
Thread t = new Thread(new Runnable()
{
    public void run()
    {
```

3. Create a local variable for your `Connection` object.

```
Connection conn = null;
```

4. Invoke `open()`. Specify the connection string.

```
try
{
    conn = Connector.open("http://www.blackberry.com;deviceside=false");
} catch (IOException e)
{
    // Process your error condition
}
```

5. If the connection attempt was successful, `open()` returns a `Connection` object that you can use. Pass the connection object to another method (`displayContent()`) that is responsible for displaying the content.

```
if (conn != null)
{
    displayContent(conn);
}
});
```

6. Start your thread.

```
t.start();
```

7. Create `displayContent()`. In this case, push a screen that uses a `Connection` parameter to retrieve and display the content, after the connection retrieval thread completes.

```
private void displayContent(final Connection conn)
{
    UiApplication.getUiApplication().invokeLater(new Runnable()
    {
        public void run() {
            UiApplication.getUiApplication().pushScreen(new HTTPOutputScreen(conn));
        }
    });
}
```

After you finish: For more information about using a connection, and to implement the `HTTPOutputScreen` class, see "Send and receive data using a network connection".

Send and receive data using a network connection

The following task shows you how to send and receive data using an HTTP connection to a web site. The data returned from the website is displayed on the screen. You can use a similar process to send and receive data using other network protocols. To make this task as general as possible, the following code sends an HTTP GET command manually to a server. Normally, you would use an `HttpConnection` interface, which constructs the HTTP command string according to the options you configure.

CAUTION: Network input and output operations are not thread-safe. Make sure you create a separate thread when you use a `Connection` object.

Before you begin: Open a network connection and pass a `Connection` object to the constructor of the class developed in this task. For more information about opening network connections, see "Open a network connection using the Network API" or "Open a network connection using the GCF".

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.component.RichTextField;
import net.rim.device.api.ui.UiApplication;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import javax.microedition.io.*;
```

2. Create an instance variable for a `RichTextField` object to display the results.

```
RichTextField _rtfOutput = new RichTextField();
```

3. Create a constructor that accepts a `Connection` object.

```
public HTTPOutputScreen(Connection conn)
{
```

4. Add the `RichTextField` to your screen, and start a thread to access the network connection.

```
_rtfOutput.setText("Retrieving data. Please wait...");
add(_rtfOutput);
ContentReaderThread t = new ContentReaderThread(conn);
t.start();
```

5. Create a method to populate your `RichTextField` with the data returned from the website, after `ContentReaderThread` completes. You invoke this method in step 19.

```
public void showContents(final String result)
{
    UiApplication.getUiApplication().invokeLater(new Runnable() {
        public void run() {
            _rtfOutput.setText(result);
        }
    });
}
```

```

    }
    });
}

```

6. In an inner class, create a thread to communicate with the web server. Accept a `ConnectionDescriptor` object as an argument.

```

private final class ContentReaderThread extends Thread
{
    private Connection _connection;

    ContentReaderThread(Connection conn)
    {
        _connection = conn;
    }
    public void run()
    {

```

7. Initialize an `OutputStream` object and an `InputStream` object to exchange data with the website.

```

        OutputStream os = null;
        InputStream is = null;

```

8. Initialize a `String` object to store the response from the website.

```

        String result = "";

```

9. Create an `OutputConnection` object to send data over your connection.

```

        try
        {
            OutputConnection outputConn = (OutputConnection) connection;

```

10. Retrieve an `OutputStream` from your `OutputConnection`.

```

            os = outputConn.openOutputStream();
            String getCommand = "GET " + "/" + " HTTP/1.0\r\n\r\n";
            os.write(getCommand.getBytes());
            os.flush();

```

11. Send an HTTP GET command to the web server over your `OutputStream`. Convert the GET command into a byte array.

```

            String getCommand = "GET " + "/" + " HTTP/1.0\r\n\r\n";
            os.write(getCommand.getBytes());
            os.flush();

```

12. Create an `InputConnection` object to receive data from your connection.

```

            InputConnection inputConn = (InputConnection) connection;

```

13. Retrieve an `InputStream` from your `InputConnection`.

```

            is = inputConn.openInputStream();

```

14. Retrieve the stream data and store it in a byte array.

```
byte[] data = net.rim.device.api.io.IOUtilities.streamToBytes(is);  
result = new String(data);
```

15. Catch any errors generated by your procedure.

```
catch(Exception e)  
{  
    result = "ERROR fetching content: " + e.toString();  
}
```

16. Close your OutputStream.

```
if(os != null)  
{  
    try  
    {  
        os.close();  
    }  
    catch(IOException e)  
    {  
        // process the error condition  
    }  
}
```

17. Close your InputStream.

```
if(is != null)  
{  
    try  
    {  
        is.close();  
    }  
    catch(IOException e)  
    {  
    }  
}
```

18. Close your Connection.

```
{  
    connection.close();  
}  
catch(IOException ioe)  
{  
}  
}
```

19. Call showContents() you created in step 5.

```
showContents(result);  
}  
}
```

Network transport options

Transport	Description
TCP Cellular	When you create a connection using the TCP Cellular transport, you may need to specify the following parameters: • APN • Tunnel authorization user name • Tunnel authorization password If you are using the Network API, you can use the <code>TCPCellularOptions</code> class to configure these options. For devices running BlackBerry® Device Software 6.0 and later, you can discover whether these options are already set by invoking <code>TCPCellularOptions.isDefaultAPNSet()</code>. If you are using the GCF, a connection string could be similar to: <pre>"http://www.blackberry.com;deviceside=true; apn=carriergateway;TunnelAuthusername=GW@CARRIERGPRS.COM; TunnelAuthPassword=CARRIER"</pre>
TCP Wi-Fi®	If you are using the GCF, you can use the Wi-Fi® transport with a connection string similar to: <pre>"http://www.blackberry.com;interface=wifi"</pre>
WAP 2.0	If WAP 2.0 is configured on a device, you will need to retrieve a connection ID from the service book. For more information, see "Code sample: Retrieving the service book record for WAP 2.0".
WAP 1.0/1.1	Like TCP Cellular, you may need to specify additional parameters to use WAP 1.0 or 1.1 gateway: • Gateway APN • Gateway IP address • Gateway port • Source IP address • Source port • Tunnel authorization user name • Tunnel authorization password

Transport	Description
	• Enable WTLS If you are using the Network API, you can use the <code>WapOptions</code> class to configure these options. If you are using the GCF, a connection string could be similar to: <pre>"http://www.blackberry.com;deviceside=true; WapGatewayIP=127.0.0.1;WapGatewayPort:9201; WapGatewayAPN=carriergateway"</pre>
BlackBerry® Mobile Data System	If you are using the GCF, you can use the BlackBerry MDS transport with a connection string similar to: <pre>"http://www.blackberry.com;deviceside=false"</pre>
BlackBerry® Internet Service	To use the BlackBerry Internet Service, you must sign up for the BlackBerry® Push Service. For more information, visit http://na.blackberry.com/eng/partners/.

Code samples

5

Code sample: Requesting data using a BlockingSenderDestination object

```
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.ui.*
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

import java.io.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new BlockingSenderSample());
    }
}

public class BlockingSenderSample extends MainScreen implements FieldChangeListener
{
    ButtonField _btnBlock = new ButtonField(Field.FIELD_HCENTER);

    private static UiApplication _app = UiApplication.getUiApplication();

    public BlockingSenderSample()
    {
        _btnBlock.setChangeListener(this);
        _btnBlock.setLabel("Fetch page");

        add(_btnBlock);
    }

    public void fieldChanged(Field button, int unused)
    {
        if(button == _btnBlock)
        {
            Thread t = new Thread(new Runnable() {
                public void run()
                {
                    Message response = null;
                    String uriStr = "http://www.blackberry.com";
                    BlockingSenderDestination bsd = null;
```

```

        try
        {
            bsd = (BlockingSenderDestination)
                DestinationFactory.getSenderDestination
                    ("CommAPISample", URI.create(uriStr));
            if(bsd == null)
            {
                bsd =
                    DestinationFactory.createBlockingSenderDestination
                        (new Context("CommAPISample"),
                            URI.create(uriStr)
                        );
            }

            // Send message and wait for response
            response = bsd.sendReceive();

            if(response != null)
            {
                BSDResponse(response);
            }
        }
        catch(Exception e)
        {
            // process the error
        }
        finally
        {
            if(bsd != null)
            {
                bsd.release();
            }
        }
    }

    });
    t.start();

}

private void BSDResponse(Message msg)
{
    InputStream is = ((StreamMessage)msg).getStreamPayload();
    try
    {
        byte[] data = net.rim.device.api.io.IOUtilities.streamToBytes(is);
        _result = new String(data);
    } catch (IOException e)
    {
        // process the error
    }
}

```

```

        _app.invokeLater(new Runnable() {
            public void run() {
                _app.pushScreen(new HTTPOutputScreen(_result));
            }
        });
    }
}

public class HTTPOutputScreen extends MainScreen
{
    RichTextField _rtfOutput = new RichTextField();

    public HTTPOutputScreen(String message)
    {
        _rtfOutput.setText("Retrieving data. Please wait...");
        add(_rtfOutput);
        showContents(message);
    }

    // After the data has been retrieved, display it
    public void showContents(final String result) {
        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run() {
                _rtfOutput.setText(result);
            }
        });
    }
}

```

Code sample: Requesting data using a NonBlockingSenderDestination object

```

import net.rim.device.api.io.URI;
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.ui.*
import net.rim.device.api.ui.component.ButtonField;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.system.Application;

import java.io.*;

```

```

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new NonBlockingSenderSample());
    }
}

public class NonBlockingSenderSample extends MainScreen
                                   implements FieldChangeListener
{
    ButtonField _btnNonBlock = new ButtonField(Field.FIELD_HCENTER);

    private static UiApplication _app = UiApplication.getUiApplication();

    public NonBlockingSenderSample()
    {
        _btnNonBlock.setChangeListener(this);
        _btnNonBlock.setLabel("Fetch page");

        add(_btnNonBlock);
    }

    public void fieldChanged(Field button, int unused)
    {
        if(button == _btnNonBlock)
        {
            NonBlockingSenderDestination destination = null;
            try
            {
                URI uri = URI.create("http://www.blackberry.com");

                NBSDMsgListener responseListener = new NBSDMsgListener();
                destination = (NonBlockingSenderDestination)
                    DestinationFactory.getSenderDestination
                        ("CommAPISample", uri);

                if (destination == null)
                {
                    destination =
                        DestinationFactory.createNonBlockingSenderDestination
                            (new Context("CommAPISample"), uri, responseListener);
                }

                // Send message to retrieve the response
                destination.send();
            }
        }
    }
}

```

```

        catch(Exception e)
        {
            // process the error
        }
    }
}

public class NBSMsgListener implements MessageListener {
    public void onMessage(Destination dest, Message msg) {
        String payload = null;

        if (msg instanceof ByteMessage)
        {
            ByteMessage reply = (ByteMessage) msg;
            payload = (String) reply.getStringPayload();
        } else if(msg instanceof StreamMessage)
        {
            StreamMessage reply = (StreamMessage) msg;
            InputStream is = reply.getStreamPayload();
            byte[] data = null;
            try {
                data = net.rim.device.api.io.IOUtilities.streamToBytes(is);
            } catch (IOException e) {
            }
            if(data != null)
            {
                payload = new String(data);
            }
        }
        if(payload!=null)
        {
            synchronized(Application.getEventLock())
            {
                UiApplication.getUiApplication().pushScreen(new HTTPOutputScreen
(payload));
            }
        }

        }

    public void onMessageCancelled(Destination arg0, int arg1) {
        // process message cancelled notification
    }
}

```

```

    public void onMessageFailed(Destination arg0, MessageFailureException arg1) {
        // process message failed notification
    }
}

public class HTTPOutputScreen extends MainScreen
{
    RichTextField _rtfOutput = new RichTextField();

    public HTTPOutputScreen(String message)
    {
        _rtfOutput.setText("Retrieving data. Please wait...");
        add(_rtfOutput);
        showContents(message);
    }

    // After the data has been retrieved, display it
    public void showContents(final String result) {
        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run() {
                _rtfOutput.setText(result);
            }
        });
    }
}

```

Code sample: Parsing a JSON data structure

The following code sample demonstrates one way to parse a JSON data structure. You can use a similar process to parse JSON or other data formats provided by the Message Processing API.

```

import net.rim.device.api.io.URI;
import net.rim.device.api.io.messaging.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

import java.io.*;

import org.json.me.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
}

```

```

    }
    public NetworkSample() {
        pushScreen(new ParseJSONSample());
    }
}

public class ParseJSONSample extends MainScreen implements FieldChangeListener
{
    ButtonField _btnJSON = new ButtonField(Field.FIELD_HCENTER);

    private static UiApplication _app = UiApplication.getUiApplication();

    public ParseJSONSample()
    {
        _btnJSON.setChangeListener(this);
        _btnJSON.setLabel("Fetch page");

        add(_btnJSON);
    }

    public void fieldChanged(Field button, int unused)
    {
        if(button == _btnJSON)
        {
            Thread t = new Thread(new Runnable() {
                public void run()
                {
                    Message response = null;
                    String uriStr = "http://docs.blackberry.com/sampleddata.json";
                    BlockingSenderDestination bsd = null;
                    try
                    {
                        bsd = (BlockingSenderDestination)
                            DestinationFactory.getSenderDestination
                                ("CommAPISample", URI.create(uriStr));
                        if(bsd == null)
                        {
                            bsd =
                                DestinationFactory.createBlockingSenderDestination
                                    (new Context("CommAPISample"),
                                        URI.create(uriStr), new JSONMessageProcessor()
                                    );
                        }

                        // Send message and wait for response
                        response = bsd.sendReceive();
                        _json = response.getObjectPayload();

                        if(_json != null)

```

```

        {
            _app.invokeLater(new Runnable() {
                public void run() {
                    _app.pushScreen(new JSONOutputScreen(_json));
                }
            });
        }
    }
    catch(Exception e)
    {
        System.out.println(e.toString());
    }
    finally
    {
        if(bsd != null)
        {
            bsd.release();
        }
    }
}

});
t.start();
}

}

}

public class JSONOutputScreen extends MainScreen implements TreeFieldCallback {
    private TreeField _treeField;

    public JSONOutputScreen(Object JSONData)
    {
        _treeField = new TreeField(this, Field.FOCUSABLE);
        add(_treeField);
        setTree(JSONData);
    }

    void setTree(Object obj)
    {
        int parentNode = 0;

        _treeField.deleteAll();

        try
        {
            if(obj instanceof JSONArray)
            {

```

```

        parentNode = populateTreeArray
                        (_treeField, (JSONArray) obj, parentNode);
    }
    else if(obj instanceof JSONObject)
    {
        parentNode = populateTreeObject
                        (_treeField, (JSONObject) obj, parentNode);
    }
}
catch(JSONException e)
{
    System.out.println(e.toString());
}

_treeField.setCurrentNode(parentNode);
}

// Populate the trees with JSON arrays
int populateTreeArray(TreeField tree, JSONArray o, int p) throws JSONException
{
    Object temp;
    int newParent;

    newParent = tree.addChildNode(p, "Array " + p);

    for(int i = 0; i < o.length(); ++i)
    {
        temp = o.get(i);

        if(temp == null || temp.toString().equalsIgnoreCase("null"))
        {
            continue;
        }

        if(temp instanceof JSONArray)
        {
            // Array of arrays
            populateTreeArray(tree, (JSONArray) temp, newParent);
        }
        else if(temp instanceof JSONObject)
        {
            // Array of objects
            populateTreeObject(tree, (JSONObject) temp, newParent);
        }
        else
        { // other values
            newParent = tree.addSiblingNode(newParent, temp.toString());
        }
    }
}

return newParent;

```

```

    }

    // Populate the tree with JSON objects
    int populateTreeObject(TreeField tree, JSONObject o, int p) throws JSONException
    {
        Object temp;

        int newParent = tree.addChildNode(p, "Object" + p);

        JSONArray a = o.names();

        for(int i = 0; i < a.length(); ++i)
        {
            temp = o.get(a.getString(i));

            if(temp == null || temp.toString().equalsIgnoreCase("null"))
            {
                continue;
            }
            if(temp instanceof JSONArray)
            {
                populateTreeArray(tree, (JSONArray) temp, newParent);
            }
            else if(temp instanceof JSONObject)
            {
                populateTreeObject(tree, (JSONObject) temp, newParent);
            }
            else
            {
                tree.addSiblingNode
                    (newParent, a.getString(i) + ": " + temp.toString());
            }
        }

        return newParent;
    }

    public void drawTreeItem(TreeField treeField, Graphics graphics, int node,
        int y, int width, int indent) {
        if(treeField == _treeField)
        {
            Object cookie = _treeField.getCookie(node);
            if(cookie instanceof String)
            {
                String text = (String) cookie;
                graphics.drawText(text, indent, y, Graphics.ELLIPSIS, width);
            }
        }
    }
}

```

```

    public boolean onSavePrompt()
    {
        // Suppress the save dialog
        return true;
    }
}

```

Code sample: Retrieving a web page using the Network API

This code sample demonstrates the process of retrieving a web page using the Network API. To make this code sample as generic as possible, the `InputConnection` and `OutputConnection` interfaces have been used in place of the `HttpConnection` interface.

```

import net.rim.device.api.io.transport.*;
import net.rim.device.api.io.transport.options.*;
import net.rim.device.api.ui.component.RichTextField;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.util.Arrays;

import java.io.*;
import javax.microedition.io.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        new HTTPConnectionSetup();
    }
}

public class HTTPConnectionSetup {

    ConnectionFactory _factory = new ConnectionFactory();

    public HTTPConnectionSetup() {

        // Create preference ordered list of transports
        int[] _intTransports =
        { TransportInfo.TRANSPORT_TCP_WIFI,
          TransportInfo.TRANSPORT_WAP2,
          TransportInfo.TRANSPORT_TCP_CELLULAR
        };
    }
}

```

```

// Remove any transports that are not (currently) available
for(int i = 0; i < _intTransports.length ; i++)
{
    int transport = _intTransports[i];
    if(!TransportInfo.isTransportTypeAvailable(transport)
        || !TransportInfo.hasSufficientCoverage(transport))
    {
        Arrays.removeAt(_intTransports, i);
    }
}

// Set options for TCP Cellular transport
TcpCellularOptions tcpOptions = new TcpCellularOptions();
if(!TcpCellularOptions.isDefaultAPNSet())
{
    tcpOptions.setApn("My APN");
    tcpOptions.setTunnelAuthUsername("user");
    tcpOptions.setTunnelAuthPassword("password");
}

// Set ConnectionFactory options
if(_intTransports.length > 0)
{
    _factory.setPreferredTransportTypes(_intTransports);
}
_factory.setTransportTypeOptions(TransportInfo.TRANSPORT_TCP_CELLULAR,
tcpOptions);
_factory.setAttemptsLimit(5);

// Open a connection on a new thread
Thread t = new Thread(new Runnable()
{
    public void run()
    {
        ConnectionDescriptor cd = _factory.getConnection
            ("http://www.blackberry.com");
        // If connection was successful, fetch and show the content from
        // the web server
        if(cd != null) {
            Connection c = cd.getConnection();
            displayContent(c);
        }
    }
});
t.start();
}

private void displayContent(final Connection conn) {
    // When the connection thread completes, show the data from the web server
    UiApplication.getUiApplication().invokeLater(new Runnable()
    {

```

```

        public void run() {
            UiApplication.getUiApplication().pushScreen(new HTTPOutputScreen(conn));
        }
    });
}

}

public class HTTPOutputScreen extends MainScreen {
    RichTextField _rtfOutput = new RichTextField();

    public HTTPOutputScreen(Connection conn)
    {
        // Create a container for the data, and put it on the screen
        _rtfOutput.setText("Retrieving data. Please wait...");
        add(_rtfOutput);
        // Retrieve the data from the web server, using the connection, on a
        // separate thread
        ContentReaderThread t = new ContentReaderThread(conn);
        t.start();
    }

    // After the data has been retrieved, display it
    public void showContents(final String result) {
        UiApplication.getUiApplication().invokeLater(new Runnable() {

            public void run() {
                _rtfOutput.setText(result);
            }
        });
    }
}

private final class ContentReaderThread extends Thread
{
    private Connection _connection;

    ContentReaderThread(Connection conn)
    {
        _connection = conn;
    }

    public void run()
    {
        String result = "";
        OutputStream os = null;
        InputStream is = null;

        try

```

```

{
    // Send HTTP GET to the server
    OutputConnection outputConn = (OutputConnection) _connection;
    os = outputConn.openOutputStream();
    String getCommand = "GET " + "/" + " HTTP/1.0\r\n\r\n";
    os.write(getCommand.getBytes());
    os.flush();

    // Get InputConnection and read the server's response
    InputConnection inputConn = (InputConnection) _connection;
    is = inputConn.openInputStream();
    byte[] data = net.rim.device.api.io.IOUtilities.streamToBytes(is);
    result = new String(data);
    // is.close();

}
catch(Exception e)
{
    result = "ERROR fetching content: " + e.toString();
}
finally
{
    // Close OutputStream
    if(os != null)
    {
        try
        {
            os.close();
        }
        catch(IOException e)
        {
        }
    }

    // Close InputStream
    if(is != null)
    {
        try
        {
            is.close();
        }
        catch(IOException e)
        {
        }
    }

    // Close Connection
    try
    {
        _connection.close();
    }
    catch(IOException ioe)

```

```

        {
        }
    }

    // Show the response received from the web server, or an error message
    showContents(result);
}
}
}
}

```

Code sample: Retrieving a web page using the GCF

This code sample demonstrates how to retrieve a web page using the GCF. This code sample uses the BlackBerry® Mobile Data System transport. To make this sample as generic as possible, the `InputConnection` and `OutputConnection` interfaces are used in place of the `HttpConnection` interface.

```

import net.rim.device.api.ui.component.RichTextField;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.UiApplication;

import java.io.*;
import javax.microedition.io.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        new HTTPConnectionSetup();
    }
}

public class HTTPConnectionSetup {
    public HTTPConnectionSetup()
    {
        Thread t = new Thread(new Runnable()
        {
            public void run() {
                Connection c = null;
                try {
                    c = Connector.open("http://www.blackberry.com;deviceside=false");
                } catch (IOException e) {

```

```

        e.printStackTrace();
    }
    if(c != null)
    {
        displayContent(c);
    }
}

});
t.start();
}

private void displayContent(final Connection c) {
    // When the connection thread completes, show the data from the web server
    UiApplication.getUiApplication().invokeLater(new Runnable()
    {
        public void run() {
            UiApplication.getUiApplication().pushScreen(new HTTPOutputScreen(c));
        }
    });
}

}

public class HTTPOutputScreen extends MainScreen {

    RichTextField _rtfOutput = new RichTextField();

    public HTTPOutputScreen(Connection conn)
    {
        // Create a container for the data, and put it on the screen
        _rtfOutput.setText("Retrieving data. Please wait...");
        add(_rtfOutput);
        // Retrieve the data from the web server, using the connection, on a
        // separate thread
        ContentReaderThread t = new ContentReaderThread(conn);
        t.start();
    }

    // After the data has been retrieved, display it
    public void showContents(final String result) {
        UiApplication.getUiApplication().invokeLater(new Runnable() {

            public void run() {
                _rtfOutput.setText(result);
            }
        });
    }

    private final class ContentReaderThread extends Thread
    {
        private Connection _connection;
    }
}

```

```

ContentReaderThread(Connection conn)
{
    _connection = conn;
}

public void run()
{
    String result = "";
    OutputStream os = null;
    InputStream is = null;

    try
    {
        // Send HTTP GET to the server
        OutputConnection outputConn = (OutputConnection) _connection;
        os = outputConn.openOutputStream();
        String getCommand = "GET " + "/" + " HTTP/1.0\r\n\r\n";
        os.write(getCommand.getBytes());
        os.flush();

        // Get InputConnection and read the server's response
        InputConnection inputConn = (InputConnection) _connection;
        is = inputConn.openInputStream();
        byte[] data = net.rim.device.api.io.IOUtilities.streamToBytes(is);
        result = new String(data);
        // is.close();

    }
    catch(Exception e)
    {
        result = "ERROR fetching content: " + e.toString();
    }
    finally
    {
        // Close OutputStream
        if(os != null)
        {
            try
            {
                os.close();
            }
            catch(IOException e)
            {
            }
        }

        // Close InputStream
        if(is != null)
        {
            try
            {

```

```

        is.close();
    }
    catch(IOException e)
    {
    }
}

// Close Connection
try
{
    _connection.close();
}
catch(IOException ioe)
{
}
}

// Show the response received from the web server, or an error message
showContents(result);
}
}

}

```

Code sample: Determining network transports with sufficient coverage using the Network API

```

import net.rim.device.api.io.transport.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new ListTransportsWithCoverageScreen());
    }
}

public class ListTransportsWithCoverageScreen extends MainScreen
{
    private int[] _transportsWithCoverage;

```

```

private TransportDescriptor[] _transports;
private RichTextField _rtfDisplay;
private ButtonField _btnShowTransports;

public ListTransportsWithCoverageScreen()
{
    VerticalFieldManager vfm = new VerticalFieldManager();
    setTitle("Network Sample");
    _rtfDisplay = new RichTextField
        ("Click button below to display available transports.");
    _btnShowTransports = new ButtonField
        ("Show Transports", Field.FIELD_HCENTER);
    _btnShowTransports.setChangeListener(new FieldChangeListener()
    {
        public void fieldChanged(Field field, int context)
        {
            getTransports();
        }
    });
    vfm.add(_rtfDisplay);
    vfm.add(_btnShowTransports);
    add(vfm);
}

public void getTransports()
{
    StringBuffer sb = new StringBuffer();
    sb.append("The transports currently available are: \n");
    _transportsWithCoverage = TransportInfo.getCoverageStatus();
    _transports = TransportInfo.getTransportDescriptors
        (_transportsWithCoverage);
    for(int i = _transports.length - 1; i >=0; --i)
    {
        switch(_transports[i].getTransportType())
        {
            case TransportInfo.TRANSPORT_BIS_B:
                sb.append("BlackBerry Internet Service\n");
                break;
            case TransportInfo.TRANSPORT_MDS:
                sb.append("BlackBerry Mobile Data Service\n");
                break;
            case TransportInfo.TRANSPORT_TCP_CELLULAR:
                sb.append("TCP Cellular\n");
                break;
            case TransportInfo.TRANSPORT_TCP_WIFI:
                sb.append("TCP WiFi\n");
                break;
            case TransportInfo.TRANSPORT_WAP:
                sb.append("WAP 1.0 or 1.1\n");
                break;
            case TransportInfo.TRANSPORT_WAP2:

```

```

        sb.append("WAP 2.0\n");
        break;
    }
    _rtfDisplay.setText(sb.toString());
}
}

```

Code sample: Controlling radios

```

import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new RadioControlScreen());
    }
}

public class RadioControlScreen extends MainScreen implements FieldChangeListener {
    boolean[] _intRadioOn = new boolean[3];
    ButtonField _btn3G, _btnCDMA, _btniDEN, _btnWiFi;

    public RadioControlScreen()
    {
        VerticalFieldManager vfm = new VerticalFieldManager(Field.FIELD_HCENTER);

        int wafs = RadioInfo.getSupportedWAFs();
        if ((wafs & RadioInfo.WAF_3GPP) != 0 )
        {
            _btn3G = new ButtonField("3G Radio");
            _btn3G.setChangeListener(this);
            vfm.add(_btn3G);
        }
        if ((wafs & RadioInfo.WAF_CDMA) != 0 )
        {
            _btnCDMA = new ButtonField("CDMA Radio");
            _btnCDMA.setChangeListener(this);
            vfm.add(_btnCDMA);
        }
    }
}

```

```

        if ((wafs & RadioInfo.WAF_WLAN) != 0 )
        {
            _btnWiFi = new ButtonField("WiFi Radio");
            _btnWiFi.setChangeListener(this);
            vfm.add(_btnWiFi);
        }

        wafs = RadioInfo.getActiveWAFs();
        if ((wafs & RadioInfo.WAF_3GPP) != 0 )
        {
            _intRadioOn[0] = true;
            _btn3G.setLabel("3G Radio is on");
        }
        if ((wafs & RadioInfo.WAF_CDMA) != 0 )
        {
            _intRadioOn[1] = true;
            _btnCDMA.setLabel("CDMA Radio is on");
        }
        if ((wafs & RadioInfo.WAF_WLAN) != 0 )
        {
            _intRadioOn[2] = true;
            _btnWiFi.setLabel("WiFi Radio is on");
        }

        add(vfm);
        StandardTitleBar tb = new StandardTitleBar();
        tb.addTitle("Radio Control Demo");
        tb.addSignalIndicator();
        setTitleBar(tb);
    }

    public void fieldChanged(Field field, int context) {
        if (field instanceof ButtonField)
        {
            if(field == _btn3G)
            {
                if (_intRadioOn[0])
                {
                    Radio.deactivateWAFs(RadioInfo.WAF_3GPP);
                    _btn3G.setLabel("3G Radio is off");
                    _intRadioOn[0] = false;
                } else
                {
                    Radio.activateWAFs(RadioInfo.WAF_3GPP);
                    _btn3G.setLabel("3G Radio is on");
                    _intRadioOn[0] = true;
                }
            } else if(field == _btnCDMA)
            {
                if (_intRadioOn[1])

```

```

        {
            Radio.deactivateWAFs(RadioInfo.WAF_CDMA);
            _btnCDMA.setLabel("CDMA Radio is off");
            _intRadioOn[1] = false;
        } else
        {
            Radio.activateWAFs(RadioInfo.WAF_CDMA);
            _btnCDMA.setLabel("CDMA Radio is on");
            _intRadioOn[1] = true;
        }
    } else if(field == _btnWiFi)
    {
        if (_intRadioOn[2])
        {
            Radio.deactivateWAFs(RadioInfo.WAF_WLAN);
            _btnWiFi.setLabel("WiFi Radio is off");
            _intRadioOn[2] = false;
        } else
        {
            Radio.activateWAFs(RadioInfo.WAF_WLAN);
            _btnWiFi.setLabel("WiFi Radio is on");
            _intRadioOn[2] = true;
        }
    }
}

}

}

```

Code sample: Determining the status of a network transport using the Network API

```

import net.rim.device.api.io.transport.TransportInfo;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new ProbeSpecificTransportScreen());
    }
}

```

```

}

public class ProbeSpecificTransportScreen extends MainScreen implements
FieldChangeListener {

    private TextField _tfTransportStatus;
    private ObjectChoiceField _ocfTransports;

    public ProbeSpecificTransportScreen()
    {
        String[] strTransportNames =
        {"none", "TCP Cellular", "WAP 1.0/1.1",
        "WAP 2.0", "MDS", "BIS", "TCP WiFi"
        };

        VerticalFieldManager vfm = new VerticalFieldManager();

        _tfTransportStatus = new TextField(Field.FIELD_HCENTER);
        _tfTransportStatus.setText
        ("Select a transport from the list above, then click 'Probe Transport'");

        _ocfTransports = new ObjectChoiceField
        ("Select Transport to Probe: ", strTransportNames,
        0, Field.FIELD_HCENTER);
        _ocfTransports.setEditable(true);

        ButtonField btnProbe = new ButtonField("Probe Transport", Field.FIELD_HCENTER);
        btnProbe.setChangeListener(this);

        vfm.add(_ocfTransports);
        vfm.add(btnProbe);
        vfm.add(_tfTransportStatus);
        add(vfm);
    }

    public void fieldChanged(Field field, int context)
    {
        int intTransportType = _ocfTransports.getSelectedIndex();
        if(intTransportType > 0)
        {
            if(TransportInfo.isTransportTypeAvailable(intTransportType))
            {
                if(TransportInfo.hasSufficientCoverage(intTransportType))
                {
                    _tfTransportStatus.setText
                    ((String)_ocfTransports.getChoice(intTransportType)
                    + " is available.");
                } else
                {
                    _tfTransportStatus.setText
                    ((String)_ocfTransports.getChoice(intTransportType)
                    + " is available but has insufficient coverage.");
                }
            }
        }
    }
}

```

```

        }
    } else
    {
        _tfTransportStatus.setText("Sorry, " +
            (String)_ocfTransports.getChoice(intTransportType)
            + " is not available.");
    }
} else
{
    _tfTransportStatus.setText
        ("Please select a transport first, then click 'Probe Transport'");
}
}
}
}

```

Code sample: Retrieving the service sook record for WAP 2.0

If you choose to use the WAP 2.0 network transport, you need to provide the CID.

```

import net.rim.device.api.servicebook.*;
import net.rim.device.api.ui.component.TextField;
import net.rim.device.api.ui.container.MainScreen;

public class NetworkSample extends UiApplication {
    public static void main(String[] args) {
        NetworkSample app = new NetworkSample();
        app.enterEventDispatcher();
    }
    public NetworkSample() {
        pushScreen(new WAP2SBRScreen());
    }
}

public class WAP2SBRScreen extends MainScreen {
    public WAP2SBRScreen()
    {
        ServiceBook sb = ServiceBook.getSB();
        ServiceRecord[] records = sb.findRecordsByCid("WPTCP");
        String uid = null;
        TextField tf = new TextField();

        for(int i=0; i < records.length; i++)

```

```

{
    //Search through all service records to find the
    //valid non-Wi-Fi and non-MMS
    //WAP 2.0 Gateway Service Record.
    if (records[i].isValid() && !records[i].isDisabled())
    {
        if (records[i].getUid() != null && records[i].getUid().length() != 0)
        {
            if ((records[i].getUid().toLowerCase().indexOf("wifi") == -1) &&
                (records[i].getUid().toLowerCase().indexOf("mms") == -1))
            {
                uid = records[i].getUid();
                break;
            }
        }
    }
}

if (uid != null)
{
    tf.setText("The CID for WAP2 is " + uid);
}
else
{
    tf.setText("The WAP2 transport is not available on this device.");
}
add(tf);
}
}

```

Glossary

6

3GPP

Third Generation Partnership Project

AES

Advanced Encryption Standard

API

application programming interface

APN

access point name

BlackBerry MDS

BlackBerry® Mobile Data System

CDMA

Code Division Multiple Access

EDGE

Enhanced Data Rates for Global Evolution

EVDO

Evolution Data Optimized

GAN

generic access network

GAN

generic access network

GERAN

GSM-EDGE Radio Access Network

GPRS

General Packet Radio Service

GSM

Global System for Mobile Communications®

HTTP

Hypertext Transfer Protocol

HTTPS

Hypertext Transfer Protocol over Secure Sockets Layer

IPPP

Internet Protocol Proxy Protocol

JSR

Java® Specification Request

MIDP

Mobile Information Device Profile

PAP

Push Access Protocol

PIN

personal identification number

RAPC

RIM Application Program Compiler

service books

Service books determine which services are available on BlackBerry devices or BlackBerry enabled devices.

SSID

service set identifier

TCP

Transmission Control Protocol

TLS

Transport Layer Security

Triple DES

Triple Data Encryption Standard

UDP

User Datagram Protocol

UMTS

Universal Mobile Telecommunications System

UTRAN

UMTS Terrestrial Radio Access Network

WAP

Wireless Application Protocol

WLAN

wireless local area network

Provide feedback

7

To provide feedback on this deliverable, visit www.blackberry.com/docsfeedback.

Legal notice

8

©2010 Research In Motion Limited. All rights reserved. BlackBerry®, RIM®, Research In Motion®, SureType®, SurePress™ and related trademarks, names, and logos are the property of Research In Motion Limited and are registered and/or used in the U.S. and countries around the world.

3GPP, UMTS are trademarks of the European Telecommunications Standards Institute. Wi-Fi is a trademark of the Wi-Fi Alliance. Java, Java ME, Java SE are trademarks of Oracle America, Inc. All other trademarks are the property of their respective owners.

This documentation including all documentation incorporated by reference herein such as documentation provided or made available at www.blackberry.com/go/docs is provided or made accessible "AS IS" and "AS AVAILABLE" and without condition, endorsement, guarantee, representation, or warranty of any kind by Research In Motion Limited and its affiliated companies ("RIM") and RIM assumes no responsibility for any typographical, technical, or other inaccuracies, errors, or omissions in this documentation. In order to protect RIM proprietary and confidential information and/or trade secrets, this documentation may describe some aspects of RIM technology in generalized terms. RIM reserves the right to periodically change information that is contained in this documentation; however, RIM makes no commitment to provide any such changes, updates, enhancements, or other additions to this documentation to you in a timely manner or at all.

This documentation might contain references to third-party sources of information, hardware or software, products or services including components and content such as content protected by copyright and/or third-party web sites (collectively the "Third Party Products and Services"). RIM does not control, and is not responsible for, any Third Party Products and Services including, without limitation the content, accuracy, copyright compliance, compatibility, performance, trustworthiness, legality, decency, links, or any other aspect of Third Party Products and Services. The inclusion of a reference to Third Party Products and Services in this documentation does not imply endorsement by RIM of the Third Party Products and Services or the third party in any way.

EXCEPT TO THE EXTENT SPECIFICALLY PROHIBITED BY APPLICABLE LAW IN YOUR JURISDICTION, ALL CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS, OR WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS OR WARRANTIES OF DURABILITY, FITNESS FOR A PARTICULAR PURPOSE OR USE, MERCHANTABILITY, MERCHANTABLE QUALITY, NON-INFRINGEMENT, SATISFACTORY QUALITY, OR TITLE, OR ARISING FROM A STATUTE OR CUSTOM OR A COURSE OF DEALING OR USAGE OF TRADE, OR RELATED TO THE DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN, ARE HEREBY EXCLUDED. YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY STATE OR PROVINCE. SOME JURISDICTIONS MAY NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES AND CONDITIONS. TO THE EXTENT PERMITTED BY LAW, ANY IMPLIED WARRANTIES OR CONDITIONS RELATING TO THE DOCUMENTATION TO THE EXTENT THEY CANNOT BE EXCLUDED AS SET OUT ABOVE, BUT CAN BE LIMITED, ARE HEREBY LIMITED TO NINETY (90) DAYS FROM THE DATE YOU FIRST ACQUIRED THE DOCUMENTATION OR THE ITEM THAT IS THE SUBJECT OF THE CLAIM.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, IN NO EVENT SHALL RIM BE LIABLE FOR ANY TYPE OF DAMAGES RELATED TO THIS DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN INCLUDING WITHOUT LIMITATION ANY OF THE FOLLOWING DAMAGES: DIRECT, CONSEQUENTIAL, EXEMPLARY, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR AGGRAVATED DAMAGES, DAMAGES FOR LOSS OF PROFITS OR REVENUES, FAILURE TO REALIZE ANY EXPECTED SAVINGS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF

BUSINESS OPPORTUNITY, OR CORRUPTION OR LOSS OF DATA, FAILURES TO TRANSMIT OR RECEIVE ANY DATA, PROBLEMS ASSOCIATED WITH ANY APPLICATIONS USED IN CONJUNCTION WITH RIM PRODUCTS OR SERVICES, DOWNTIME COSTS, LOSS OF THE USE OF RIM PRODUCTS OR SERVICES OR ANY PORTION THEREOF OR OF ANY AIRTIME SERVICES, COST OF SUBSTITUTE GOODS, COSTS OF COVER, FACILITIES OR SERVICES, COST OF CAPITAL, OR OTHER SIMILAR PECUNIARY LOSSES, WHETHER OR NOT SUCH DAMAGES WERE FORESEEN OR UNFORESEEN, AND EVEN IF RIM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, RIM SHALL HAVE NO OTHER OBLIGATION, DUTY, OR LIABILITY WHATSOEVER IN CONTRACT, TORT, OR OTHERWISE TO YOU INCLUDING ANY LIABILITY FOR NEGLIGENCE OR STRICT LIABILITY.

THE LIMITATIONS, EXCLUSIONS, AND DISCLAIMERS HEREIN SHALL APPLY: (A) IRRESPECTIVE OF THE NATURE OF THE CAUSE OF ACTION, DEMAND, OR ACTION BY YOU INCLUDING BUT NOT LIMITED TO BREACH OF CONTRACT, NEGLIGENCE, TORT, STRICT LIABILITY OR ANY OTHER LEGAL THEORY AND SHALL SURVIVE A FUNDAMENTAL BREACH OR BREACHES OR THE FAILURE OF THE ESSENTIAL PURPOSE OF THIS AGREEMENT OR OF ANY REMEDY CONTAINED HEREIN; AND (B) TO RIM AND ITS AFFILIATED COMPANIES, THEIR SUCCESSORS, ASSIGNS, AGENTS, SUPPLIERS (INCLUDING AIRTIME SERVICE PROVIDERS), AUTHORIZED RIM DISTRIBUTORS (ALSO INCLUDING AIRTIME SERVICE PROVIDERS) AND THEIR RESPECTIVE DIRECTORS, EMPLOYEES, AND INDEPENDENT CONTRACTORS.

IN ADDITION TO THE LIMITATIONS AND EXCLUSIONS SET OUT ABOVE, IN NO EVENT SHALL ANY DIRECTOR, EMPLOYEE, AGENT, DISTRIBUTOR, SUPPLIER, INDEPENDENT CONTRACTOR OF RIM OR ANY AFFILIATES OF RIM HAVE ANY LIABILITY ARISING FROM OR RELATED TO THE DOCUMENTATION.

Prior to subscribing for, installing, or using any Third Party Products and Services, it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers might not offer Internet browsing functionality with a subscription to the BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with RIM's products and services may require one or more patent, trademark, copyright, or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use Third Party Products and Services and if any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with RIM's products and services are provided as a convenience to you and are provided "AS IS" with no express or implied conditions, endorsements, guarantees, representations, or warranties of any kind by RIM and RIM assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with RIM.

Certain features outlined in this documentation require a minimum version of BlackBerry® Enterprise Server, BlackBerry® Desktop Software, and/or BlackBerry® Device Software.

The terms of use of any RIM product or service are set out in a separate license or other agreement with RIM applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY RIM FOR PORTIONS OF ANY RIM PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.

Research In Motion Limited
295 Phillip Street
Waterloo, ON N2L 3W8

Canada

Research In Motion UK Limited
Centrum House
36 Station Road
Egham, Surrey TW20 9LF
United Kingdom

Published in Canada