Palantir™

Palantir Developer's Guide

Version 3.3.1

Legal Statements

Document Name: Palantir Developer's Guide
Document Version: 3.3.1
Release Date: 2011-03

Palantir Customer Support

Contact Palantir Customer Support with questions about our product or training opportunities:

Phone: 877-247-2513
E-mail: support@palantir.com
Website: www.palantir.com

If you have questions, comments, or suggestions about this or any other Palantir document, contact us at support@palantir.com. Your feedback is always welcomed.

Contents

1 Development on the Palantir Platform

This chapter provides basic information you need to understand and procedures you need to perform before you can program on the Palantir Platform. It covers the following topics.

- Introducing the Possibilities
- Understanding the Hardware/Software Required
- Preparing the Development Environment

Introducing the Possibilities

This section introduces the concept of a palantir, provides a list of the available project types, and describes the characteristics you need to program in the platform.

What is a Palantir?

In J.R.R. Tolkien's fantasy world a palantir (little p) is a magical artifact similar to a crystal ball. A person looking into a palantir can communicate with others who also have a palantir. People who owned palantirs had great power because a properly manipulated palantir allowed a person to see any part of the world.

The Palantir Platform, just like Tolkien's magical palantir artifact, can give analysts powerful insight into the world around them. Moreover, you can extend the platform and customize it for your particular business. Effectively, you can create your own custom palantir that sees into your world in the way that is best for your users or analysts.

For example, if your business involves doing security analysis of video footage, you might want to write a custom plugin that brings video into Palantir. Your custom plugin might:

- run the video in the context of the Palantir Workspace
- allow a user to pause the video and tag a person or event
- construct a video time line with markers for each person or event
- excerpt key portions of the video and store them

Your code need only understand and handle video. The Palantir Platform provides the tools for storing the tagged data and for constructing a time line — just for a start. Once in Palantir, your users can publish this new data for use by other analysts.

What Kind of Experience is Required?

To build a plugin for the Palantir Platform, you should have experience programming with Java. The Palantir Workspace relies on Swing and the Abstract Window Toolkit for creating user interfaces. If you are not familiar with the specifics of programming Java user interfaces, a good place to start is a review of the Swing Architecture Overview.

What Can Be Built?

Palantir allows you to customize many aspects of the platform. This section provides a brief overview of some of the possible customizations and extensions.

- Custom Applications
- Application Helpers
- Ontology Customizations
- Custom Highlighting Rules
- Custom Jobs
- Importing and Extracting Data
- Authentication and Authorization
- Custom Tile Sources

Custom Applications

You can write a custom application for the Palantir Workspace. For information on writing a custom application, see the Palantir DevZone for more information.

Application Helpers

A common customization is to add a ***helper*** to one or more of the Workspace applications. A helper is a utility such as the **Timeline**. helpers can access and update information in Palantir's repository. Additionally, a custom helper can store and maintain its own private application state. See Chapter 7, Helper Programming for more information.

Ontology Customizations

You can create customizations for how Palantir interacts with the different aspects of an ontology. During a data import, Palantir takes raw String data and creates properties. You can customize how Palantir validates this property data and how it is formatted. You can also customize how the system interprets this data when attempting to match it against other data in the system.

You can write customization code that validates user-created ***data source***. Your code can also customize the type of document object Palantir creates from a source. You can also write customizations that control how dates are formatted.

See the following chapters in this guide for more information:

- Chapter 11, Extending the IProperty APIs
- Chapter 12, IDataSourceMaker API
- Chapter 13, DatePrimitiveMaker: Extending the Ontology's Date Parsing

Custom Highlighting Rules

You can write custom code that highlights text in the Document Viewer. You would use this type of customization to define the presentation of ontology elements that appear in text. For example, you might want to display an event in a red color and with a solid line underneath. See the Palantir DevZone for more information.

Custom Jobs

You can write custom code that executes a job on the Job Server. Custom jobs appear within the Palantir Enterprise Manager just as do the standard jobs such as import/resync that are standard with the platform. See Chapter 10, Working with the Job APIs for more information.

Importing and Extracting Data

You can write custom crawlers and extractors that automatically pull data into and output data from the Palantir Platform. This is a specialized form of custom code and it is covered in detail in the *Palantir Data Integration Guide*.

Authentication and Authorization

There are also APIs you can extend that support the custom validation of password policies. You can also write a custom authentication plugin. For more information on password policy extensions, see Chapter 14, IPasswordPolicy API.

Custom Tile Sources

The Map application can use custom tile sources. You can also create and plugin custom tile sources. See the Palantir DevZone for more information.

Understanding the Hardware/Software Required

Palantir suggests that you develop your custom applications in a development system rather than your production environment. Once your application development is complete, you should move your code into a staging environment for extensive testing and quality assurance. Only when you are sure that your code is performing correctly and securely in the staging environment should you move it into production. This section describes the requirements for both development and staging environments.

Requirements for Developer Workstations

A developer workstation is at minimum a single workstation. Typically, you should install the Palantir QuickStart installation on your developer workstations. The Palantir QuickStart is a demonstration pared-down version of Palantir that runs on Windows. It contains the core servers but does not use the Palantir Enterprise Manager. The QuickStart installation includes a Workspace client and a non-production PostgresSQL database.

The QuickStart is suitable for most of the custom code development you will do. For this reason, this guide assumes you are using the QuickStart for your development environment. The requirements for a developer workstation running the QuickStart installation are:

Table 1 Hardware Requirements for Developer Workstations

Component	Minimum Requirement	Recommended Configuration
CPU	One of: • Intel Pentium 4 @ 2GHz+ • AMD 2000xp+	One of the latest generation x86 CPUs with dual cores: • Intel Pentium 4 @ 3.0GHz+ • AMD 2400xp+
Graphics card	nVidia or ATI branded with 64 MB of VRAM, PCIe (8x)	nVidia or ATI branded with 128 MB of VRAM, PCIe (16x)
Network interface card (NIC)	Ethernet	Fast Ethernet
Monitor	One, with 1280x1024, 32-bit True Color resolution	Multiple, with 1280x1024, 32-bit True Color resolution
Memory (RAM)	2 GB	4 GB or more
Hard disk	100 GB	200 GB or more

Note: To run PostgreSQL, which is required for Palantir Quickstart, Microsoft patch KB951847 [Microsoft .NET Framework 3.5 Service Pack 1 and .NET Framework 3.5 Family Update] is also required.

The software requirements are as follows:

Table 2 Software Requirements for Developer Workstations

Software	Minimum Requirement	Recommended Configuration
Java Standard Edition Development Kit (Java SE JDK)	Version 5 (Java 1.5)	Version 6 (Java 1.6)
	Note: Compile your custom code with the same Java version with which users access Palantir.	

Requirements for Staging System Servers

You use the staging system to test your custom code before moving it to a production system. Ideally, your staging system mirrors exactly the configuration of your production environment. In practice this is often not possible. You should install at least the core servers — Dispatch Server, Search Server, Job Server, and the optional Web Services, and any necessary Raptor modules in a staging area.

If you have access to three machines, a recommended configuration might include:

- Dispatch, Job, Search, and Authentication servers on one machine

- Raptor Search Engine and Raptor Search Node servers on a second machine
- Extraction Integration Server on a third machine.

Software requirements include details about software for database instances accessed by the Palantir Dispatch and Raptor Search Engine servers. Please review the software specifications and use those details to derive hardware requirements for the database servers. Hardware specifications for database servers are not provided.

The following hardware requirements are for a full production installation of Palantir. You should use your best judgement when creating your staging environment.

Table 3 Hardware Requirements for Palantir Servers

Component	Server	Minimum Requirement	Recommended Configuration
CPU	All	2 x dual-core latest generation AMD or Intel Server CPUs	2 x quad-core latest generation AMD or Intel Server CPUs
NIC	All	Gigabit Ethernet	Gigabit Ethernet with redundant connectivity
RAM	All	16 GB	32 GB
Hard disks	• Dispatch • Job • Web • Authentication • Extraction Integration • Raptor Search Engine • Monitoring Server and Client	• 512 MB or more of disk space for server installations Be sure there is enough room in /tmp to expand the packges during the installation. • 73 GB SAS or SATA drives in a hardware RAID 1 (mirrored) configuration • Additional disk space as needed to scale servers and to retain log files	
	• Search Node • Raptor Search Node • Horizon • Raptor Horizon	• 512 MB of disk space for Palantir server installation • Four or more 73 GB SAS drives in a hardware RAID 10 (mirrored and striped) configuration • Additional disk space as needed to scale servers and to provide enough space for `/indexes.bak` backups	
	• Enterprise Manager	• 25 GB of disk space for installation package repository, configuration repository, and log archives • 73 GB SAS or SATA drives in a hardware RAID 1 (mirrored) configuration • Additional disk space as needed to provide enough space for *retentionPolicy* backups	

Note: Palantir has successfully deployed, and recommends, HP DL385/585 G5 and Dell 2970 servers. Palantir Customer Support can help you determine specific hardware needs for your deployment.

Software requirements for staging systems:

Table 4 Software requirements for Palantir and Database Servers

Software	Server	Requirement
Linux Kernel 2.6 (32-bit or 64-bit)	All Palantir servers (32-bit or 64-bit)	CentOS 5.4, or Red Hat Enterprise Linux (RHEL) Advanced Platform 5 update 4 with `sysstat` package installed.* **Note:** Linux software on all servers must be the same version and must have the same updates installed.
Oracle	Database server(s)	Oracle 11g R2, Standard or Standard One Edition, with Oracle Patch Set 11.2.0.1.0 64-bit, with separate database user accounts for: ● Dispatch Servers, in all Palantir deployments ● Raptor Search Engines, if Raptor is installed

*The `sysstat` package is required to run the Palantir Monitoring Client on host machines.

Preparing the Development Environment

This section explains how to prepare and configure your Palantir development environment. The following processes are documented.

- Installing Palantir QuickStart
- Setting up the Palantir Eclipse Plugin
- Verifying Your Environment for JDK 1.5 or Higher

Installing Palantir QuickStart

To download the QuickStart installation, visit Palantir Support at:

https://support.palantir.com/

Double click the icon and follow the instructions in the installation wizard. If you are unfamiliar with Palantir, you might want to review the *Palantir Getting Started Guide*.

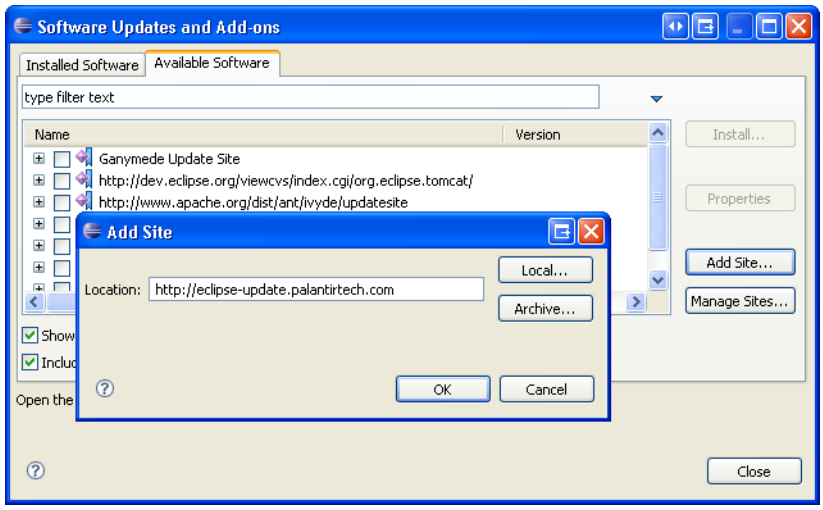
Setting up the Palantir Eclipse Plugin

Palantir recommends that your develop your applications in the Eclipse IDE. Toward this end, Palantir has developed an Eclipse plugin. The plugin sets up your Palantir installation as a build path variable. The plugin also has a wizard you can use to easily create new Palantir projects.

Installing the Plugin

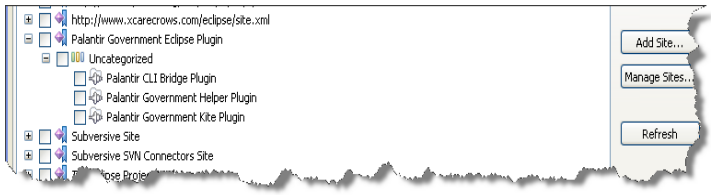
To install the Eclipse plugin, start Eclipse and do the following:

1. Choose **Help > Software Updates...** from the main Eclipse toolbar.
   The **Software Updates and Add-ons** dialog appears.
2. Choose the **Available Software** tab.
3. Click the **Add Site...** button.
4. Enter `http://eclipse-update.palantir.com` in the **Add Site** dialog,



Note: If you do not have access to the Internet, obtain the installation files from the Palantir Support Portal. Then, use the **Local...** button to load the appropriate source. If you do not have access to the support portal, contact `support@palantir.com`.

5. Click **OK** to accept your changes and return to the **Available Software** tab.
6. Check the **Palantir Government Helper** on the dialog:



   Deselect Group items by category to see the Palantir plugin components.

7. Click **Install**.
   The **Install** wizard appears.
8. Ensure your selection appears and press **Finish** to install.
   After the installation completes, Eclipse will prompt you to restart.
9. Click the **Yes** button to restart Eclipse.

Configuring the Plugin

Start Eclipse if you have not. Then, do the following to configure your workspace to automatically include the Palantir installation in the build path:

1. Create a new workspace for your Palantir projects.

   This will preserve any existing workspace that you might have.

2. Select **Window > Preferences** from the main menu.

   The system displays the **Preferences** dialog.

3. Navigate to the **Palantir** panel by selecting **Palantir** in the left-hand side category column of the preferences dialog or type `Palantir` in the filter box.

   Note: If you do not see the **Palantir** option, restart Eclipse with the `-clean` flag.

4. Click the **Browse** button on the **Palantir** panel.

5. Browse to your Palantir QuickStart installation directory and click **OK**.

   By default, the QuickStart installs into a `C:\Palantir\`*`version`* directory. Eclipse fills in the field with your selection:

6. Click **OK** to exit the **Preferences** dialog.

Note: If you ever need to reset this preference, for example, if you install a new version of the QuickStart, restart Eclipse after making the change.

Verifying Your Environment for JDK 1.5 or Higher

Palantir relies on you having version 1.5 or later of Sun's Java SE Developer Kit (JDK). You should download and install an appropriate version. You should make sure your Eclipse workspace is using the correct JDK.

Ensuring You Have the Correct JDK Installed

To do this, start Eclipse making sure to open the workspace you will use for Palantir. Then:

1. Select **Window > Preferences** from the main menu.

   The **Preferences** dialog appears.

2. Navigate to the Installed JRE panel by choosing **Java > Installed JREs** or by typing `JRE` in the filter box and choosing **Installed JREs** from the result.

   Note: The JDK includes a JRE directory.

3. Ensure that JDK 1.5 or later is selected as the default.

Adding a JDK

If you do not have the proper JDK installed, you can add it by doing the following:

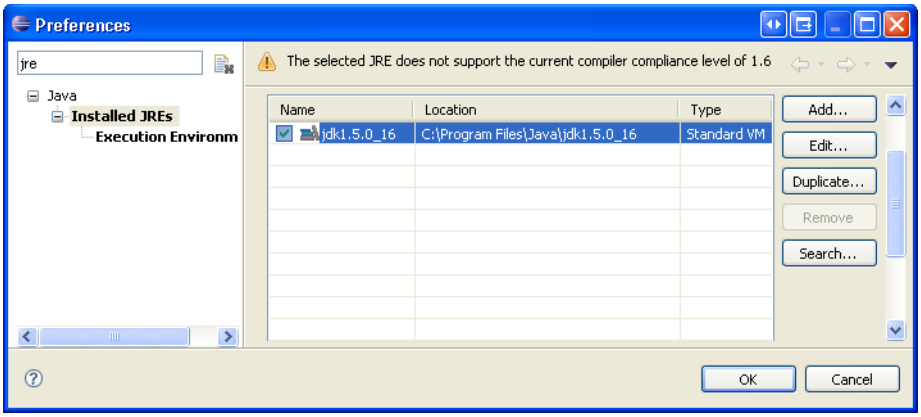
1. Navigate to the **Installed JREs** dialog in Eclipse.
2. Choose **Add...**

   The **Add JRE** wizard appears.
3. Choose **Standard VM** and **Press Next**.

   The **JRE Definition** step of the wizard appears.
4. Click the **Directory** button and navigate to the directory that contains your JDK.

   You should see a jre subdirectory in your JDK home directory.
5. Click **Finish** to complete the Wizard.

   The **Installed JREs** dialog should include the JDK you just added:



6. Press **OK** to accept your changes.

If You are Using JDK 1.6 or Higher

Only do this step if you are using JDK 1.6 higher. If you are using 1.6 or higher you must ensure that your Eclipse compliance levels are set as follows.

1. Start Eclipse and select **Window > Preferences** from the main menu.

   The the **Preferences** dialog appears.
2. Navigate to the **Java > Compiler** panel.
3. Set the **Compiler compliance level** to `1.5`.

Caution: You must also ensure that the `WEBSTART_JRE_VERSIONS` value in the `dispatch.prefs` file is set appropriately to support your JRE. This value contains a comma-separated list of the JRE versions that are supported for Web Start users. Failure to set this value can result in problems with your client freezing.

Enabling Assertions

While you are developing, you should enable assertions with your Installed JRE.

1. Start Eclipse and select **Window > Preferences** from the main menu.
2. Navigate to the **Installed JREs** panel.
3. Select your JDK and click **Edit**.
   The the **Edit JRE** dialog appears.
4. Enter `-ea` in the **Default VM Arguments** field and click **OK** to save the change.

2 Making Your First Custom Code Project

This section walks you through the creation of an sample client plugin project using the Eclipse plugin. This plugin adds a helper to the Palantir Workspace client. Make sure you have already completed Preparing the Development Environment in Chapter 1 before working through these steps. The following topics are covered:

- Starting a New Client Plugin Project
- Understanding the Project Components
- Walking through the Sample Plugin Code
- Running a Palantir Workspace Plugin from Eclipse
- Debugging the Sample Plugin
- Developing Other Plugin Types with QuickStart
- Trouble Shooting Eclipse Plugin Problems

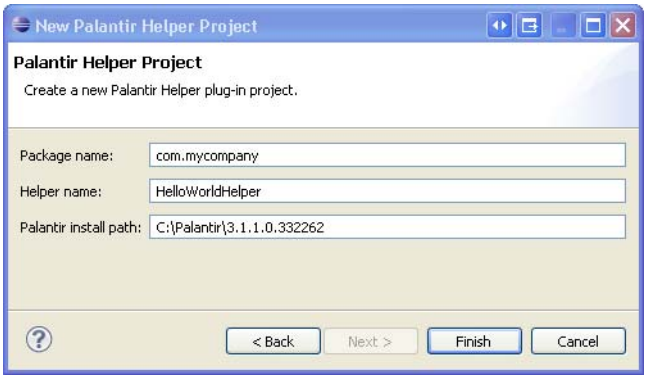
Starting a New Client Plugin Project

Note: The procedures in this section are specific to deploying a Workspace client-plugin project. See Developing Other Plugin Types with QuickStart on page 27 for information about deploying other types of plugins in a QuickStart environment.

If you have not already done so, start Eclipse, opening it to the workspace configured for Palantir. Then, do the following:

1. Choose **File > New > Project** from the main menu.
2. Type `Palantir` in the **Wizards** field.
3. Select **Palantir Helper Plugin Project**.
4. Press **Next**.
5. Enter `HelloWorld` for a project name, check **Use default location**, and then click **Next**.
6. Change the defaults for **Package name** to reflect your company name.

7. Change the helper name to `HelloWorldHelper` and click **Finish**.



You can customize the **Palantir install path** on a per-project basis.

Eclipse creates a new `HelloWord` project in your workspace.

Understanding the Project Components

Palantir's Eclipse plugin produces a sample plugin that you can build and deploy without making any further changes at all. This section discusses the structure of a simple Palantir helper project which is as follows:

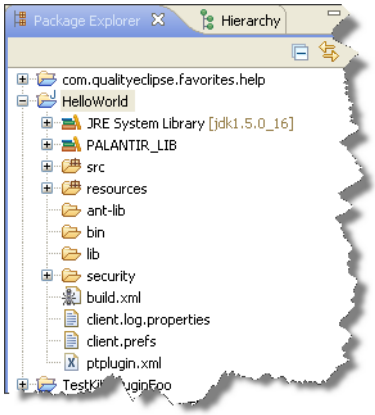


Table 5 Default Helper Project Layout

Component	Description
JRE System Library	Configured via Eclipse (see Verifying Your Environment for JDK 1.5 or Higher on page 14)
`PALANTIR_LIB`	Created when you configured the Palantir Eclipse plugin (Setting up the Palantir Eclipse Plugin on page 12). This build variable points to the location of your QuickStart client libraries.
`src`	Contains the source for your project.

Table 5 Default Helper Project Layout (continued)

Component	Description
`resources`	Contains resources your project uses such as custom icons, image files, or preferences files. Put third-party JAR files in the `lib` directory.
`ant-lib`	Contains Palantir-specific ant tasks.
`bin`	Contains executables your project generates.
`lib`	Contains third-party libraries (dependencies) your project might need.
`security`	Contains files for accessing the Palantir installation. By default, the Eclipse plugin creates a `security` directory that works with the QuickStart installation. If you are not using QuickStart, you will need to copy the `security` folder from your Palantir Dispatch Server over to this folder in your Eclipse project.
`build.xml`	Defines the ant build for the project. Run this when you are done writing your project. This build produces a PAR file for your project.
`ptplugin.xml`	Configures a plugin project for loading into the Palantir Platform. See Chapter 3, Working with ptplugin.xml Files.

Walking through the Sample Plugin Code

The following section is an overview of the code in the single factory class created by the Palantir project wizard. If you followed the steps in previous sections, you should already have the code for this walk through. If you do not have the code, you can also download a ZIP package that contains the entire set of sample code.

For the majority of user interface widgets, the Palantir Workspace makes use of standard Java AWT and Swing classes. At runtime, Palantir takes care of applying any special decorations on the components to ensure that they match the standard Palantir look and feel.

```
import java.awt.BorderLayout;
import java.awt.Color;
...
import javax.swing.SwingConstants;
```

Palantir *does* supply the `com.palantir.ui` package that contains utility classes and special user interface elements.

```
import com.palantir.ui.TableLayouts;
import com.palantir.ui.component.ScaledImagePanel;
```

The `com.palantir.api.workspace` package contains key packages for constructing helpers and working with the Workspace client:

```
import com.palantir.api.workspace.AbstractHelperFactory;
import com.palantir.api.workspace.ApplicationContext;
import com.palantir.api.workspace.ApplicationInterface;
import com.palantir.api.workspace.HelperFactory;
import com.palantir.api.workspace.HelperInterface;
import com.palantir.api.workspace.PalantirContext;
import com.palantir.api.workspace.PalantirFrame;
import com.palantir.api.workspace.applications.GraphApplicationInterface;
```

All helpers extend the `AbstractHelperFactory`. Below, the `HelloWorldHelperFactory` is a `HelperFactory` which generates the `HelloWorldHelper` interface.

```
public class HelloWorldHelperFactory extends AbstractHelperFactory {
   public HelloWorldHelperFactory() {
     super("HelloWorldHelper",
         new String[] { GraphApplicationInterface.APPLICATION_URI },
         new Integer [] { SwingConstants.VERTICAL },
         new Dimension(330,500),
         null,
         "com.mycompany.HelloWorldHelper");
   }
```

The constructor below creates a helper instance. Every helper operates within a specific `PalantirContext`. The context gives your helper access to the Workspace and the applications within it such as the Browser, Map, and so forth.

```
   public HelperInterface createHelper(PalantirContext palantirContext,
                                       ApplicationInterface application) {
     return new HelloWorldHelper(this, palantirContext, application);
   }
```

Each helper must implement the `HelperInterface`.

```
   protected static class HelloWorldHelper implements HelperInterface{
     private HelperFactory factory;
     private JTextField nameField;
     private JPanel panel;
     private PalantirContext palantirContext;
     private PalantirFrame palantirFrame;

@SuppressWarnings("serial")
     public HelloWorld(HelperFactory factory, PalantirContext
palantirContext, ApplicationInterface application) {
       this.factory = factory;
       this.palantirContext = palantirContext;
       panel = new JPanel(new BorderLayout());
       JPanel topPanel = new JPanel(TableLayouts.create(
                                    "1,p,p","p,p,p,p", 8, 8));

topPanel.setBackground(palantirContext.getColors().getSecondaryBackgroundCol
or());
       topPanel.setBorder(BorderFactory.createMatteBorder(0, 0, 1, 0,
                                                          Color.GRAY));
       JLabel label = new JLabel("Request for Information");
       palantirContext.registerComponentForFontManagement(+4, label);
       topPanel.add(label, "1,0,2,0");
```

The `PalantirContext` also provides utilities for registering components for font management, adding listeners, and access to the interface colors.

```
JLabel l4 = new JLabel("Name:");
   palantirContext.registerComponentForFontManagement(0, l4);
   topPanel.add(l4, "1,1");
   nameField = new JTextField("Janet Jones");
   palantirContext.registerComponentForFontManagement(0, nameField);
   topPanel.add(nameField, "2,1");
   JLabel l2 = new JLabel("Forward Operating Base:");
   palantirContext.registerComponentForFontManagement(0, l2);
   topPanel.add(l2, "1,2");
   JComboBox box = new JComboBox(new Object[] {"Papa Tango", "Alpha
                                 Charlie Delta Mike", "Bravo", "Zulu"});
   palantirContext.registerComponentForFontManagement(0, box);
   topPanel.add(box, "2,2");
   JButton searchButton = new JButton("Search");
   searchButton.setActionCommand("SEARCH");
   searchButton.addActionListener(this);
   topPanel.add( searchButton, "1,3");
   JButton createButton = new JButton("Create");
   createButton.setActionCommand("CREATE");
   createButton.addActionListener(this);
   topPanel.add( createButton, "2,3");
```

If your helper contains custom images or icons, you need to place these in the `resources` folder of your project.

```
panel.setBackground(palantirContext.getColors().getPrimaryBackgroundColor())
;
   ScaledImagePanel image = new ScaledImagePanel(retrieveImage("/
                                                  baghdad_tall.jpg")) {
       @Override
       protected void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.setColor(new Color(255, 0, 0, 64));
         g.fillRect(30, 150, 260, 140);
         g.setColor(new Color(255, 0, 0, 255));
         g.drawRect(30, 150, 260, 140);
       }
      };
      image.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
      image.setLineBorderColor(Color.BLACK);
      panel.add(image, BorderLayout.CENTER);
      panel.add(topPanel, BorderLayout.NORTH);
    }
    public String getDefaultPosition() {
      return BorderLayout.EAST;
    }
    public JComponent getDisplayComponent() {
      return panel;
    }
    public Image getFrameIcon() {
      return null;
    }
    public void setOwners(PalantirFrame pFrame, ApplicationContext
appContext) {
      this.palantirFrame = pFrame;
    }
    public Icon getIcon() {
```

```
    Icon myicon = new ImageIcon(retrieveImage("/small_world.gif"));
    return myicon;
  }
  public HelperFactory getFactory() {
    return factory;
  }
  public String getTitle() {
    return "HelloWorld Helper";
  }
  public void initialize(ApplicationInterface app) {
    // do nothing
  }
  public void dispose(ApplicationInterface arg0) {
    // do nothing
  }
  public void setConstraint(String constraint) {
    // do nothing
  }
  private Image retrieveImage(String name) {
    Image theImage=null;
    try {
      theImage = new ImageIcon(ImageIO.read(HelloWorldFactory.class.
                    getResource(name))).getImage();
      return theImage;
    } catch (Exception e) {
      e.printStackTrace();
    }
    return theImage;
  }
  public void actionPerformed(ActionEvent e) {
    if( e.getActionCommand().equals("SEARCH") ) {
      doSearch();
    }
    else if( e.getActionCommand().equals("CREATE")) {
      doCreate();
    }
  }
```

The following code defines the search routine called from the helper's **Search** button.

```
private void doSearch() {
    final String name = this.nameField.getText();
    if( StringUtils.isEmpty(name)) {
      JOptionPane.showMessageDialog( palantirFrame.getFrame(),
                                     "No name specified");
      return;
    }

    final PalantirConnection conn = this.palantirContext.
                              getPalantirConnection();

    SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
      @Override
      protected Void doInBackground() throws Exception {
        //new search
        ISearchQuery sq = palantirContext.getSearchFactory().
                          getNewSearchQuery(OperatorType.INTERSECT );

        // by person
        sq.addObjectTypeTerm(PTObjectType.PERSON);
```

```
            PropertyType nameType = PropertyType.NAME;
            sq.addPropertyTerm(nameType, name, SearchOperator.EQUALS);

            // run the query (get pages of results)
            SearchResultsPager pager = conn.search(sq);
            ResultsPage<PTObjectContainer,PalantirSearchException>
                        currentPage = pager;
            Collection<PTObjectContainer> ptocs = new
                                              ArrayList<PTObjectContainer>();
            while (currentPage.moreResultsAvailable()) {
              currentPage = currentPage.getNextPage();
                    ptocs.addAll(currentPage.getResults());
            }
            // Load the data
            Collection<Locator> locs =
                                 BasicPTObjectContainerUtils.toLocators(ptocs);
            ptocs = conn.objectLoad(locs,
                    LoadLevelFactory.getFullyExceptDSRLoadedInstance());
            // if no results, notify user
            if( ptocs.isEmpty() ) {
              JOptionPane.showMessageDialog(palantirFrame.getFrame(),"No
                                           objects found with name:" + name );
              return null;
            }

            //add the results to the graph
            palantirContext.getGraph().addObjectsToGraph(ptocs);

            return null;
          }

        };

        palantirContext.getMonitoredExecutorService().execute(worker);
      }
```

Finally, the following section creates data sources and uses them as a basis to source new Palantir objects:

```
private void doCreate() {
       final String name = this.nameField.getText();
       if( StringUtils.isEmpty(name)) {
         JOptionPane.showMessageDialog(palantirFrame.getFrame(),
                                       "No name specified");
         return;
       }
       final PalantirConnection conn =
                                 this.palantirContext.getPalantirConnection();
       final PalantirInvestigation inv =
                                    this.palantirContext.getInvestigation();

       SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
         @Override
         protected Void doInBackground() throws Exception {

           //create the data source records
           DataSourceRecord dsr =
                          conn.getDsrFactory().createManuallyEnteredDsr();
```

```
            Collection<DataSourceRecord> dsrs = Collections.singleton(dsr);

            //create object
            PTObjectType ptocType = PTObjectType.PERSON;
            PTObjectContainer ptoc = PTObjectContainerFactory.
                                     createBlankObject(conn, ptocType );

            //set the title/label
            String label = name;
            ptoc.setTitle(label, conn, dsrs, SETTER_STYLE.KEEP_OTHERS, 1L);
            //set the name property
            PropertyType nameProp = PropertyType.NAME;
            Property p = Property.attemptToCreate(conn, nameProp, name,
Role.NONE);
            p.addDataSourceRecord(dsr.deepCopy(true));
            ptoc.addProperty(p);
            ArrayList<PTObjectContainer> ptocs = new
ArrayList<PTObjectContainer>();
            ptocs.add( ptoc );

            // Store the object to the current investigative realm
            conn.objectStore(ptocs,PalantirDataEventType.DATA);

            // add to the graph
            palantirContext.getGraph().addObjectsToGraph(ptocs);
            return null;
          }

        };

        palantirContext.getMonitoredExecutorService().execute(worker);
      }
    }
}
```

Updating the dispatch.prefs File for State (Optional)

If you are developing plugins with application state or preferences, you need to anticipate this when setting up your plugin project. A Quickstart environment does not automatically support application state or plugin preferences. You must modify the QuickStart's `dispatch.prefs` file and add a `DEVELOPER_APP_STATE_URIS` preference.

The `dispatch.prefs` file is in the Quickstart `INSTALL_DIR` folder. To use this preference, specify a comma separated list of URIs for your plugins that contain application state or plugin preferences. For example:

```
DEVELOPER_APP_STATE_URIS=com.mycompany.plugin1, com.mycompany.plugin2
```

This registers the necessary URIs with the server so that it can support the requisite states or preferences for your plugin.

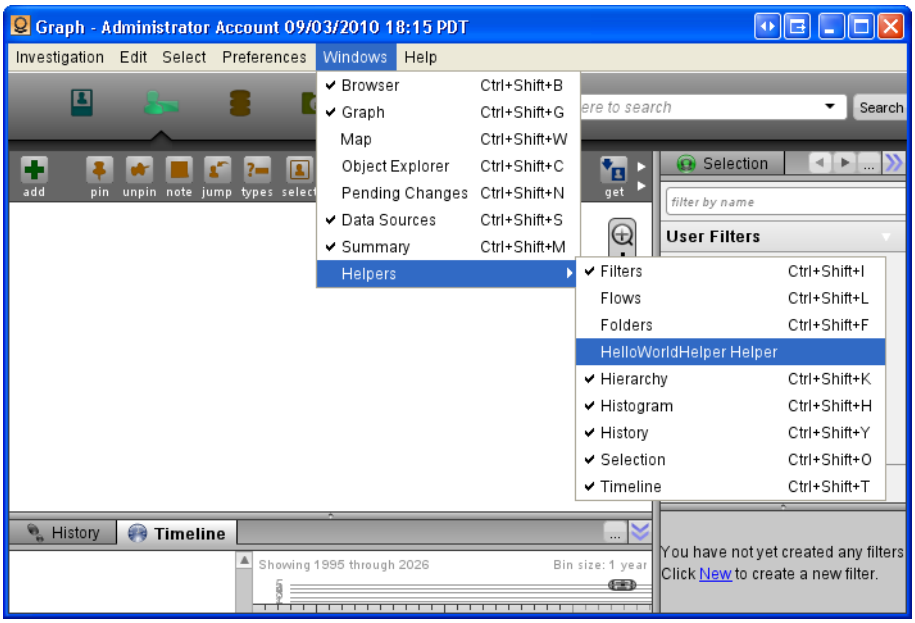
If you do not specify this preference in `dispatch.prefs`, your plugin might not load properly using the **Run As > Palantir Workspace Plugin** feature of the Eclipse plugin. If you do load the plugin either directly through the **Run As** feature the state or preferences will not behave correctly.

Running a Palantir Workspace Plugin from Eclipse

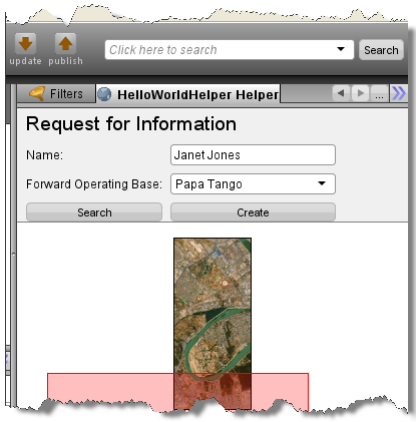
Developing a Palantir plugin in Eclipse gives you the ability to make code changes and then load them without having to recompile. To launch your helper in the Workspace, do the following:

1. Start the Palantir Dispatch Server.
2. If you have not yet done so, start Eclipse and open your project.
3. Open the Package Explorer view.
4. Right click the project folder and select **Run As > Palantir Workspace Plugin** from the pop-up menu.

   The system launches the Workspace and prompts you to login.
5. Log into the Workspace.
6. Open the Graph application.
7. From the Workspace menu bar, choose **Windows > Helper > *HelperName*** to display your helper.



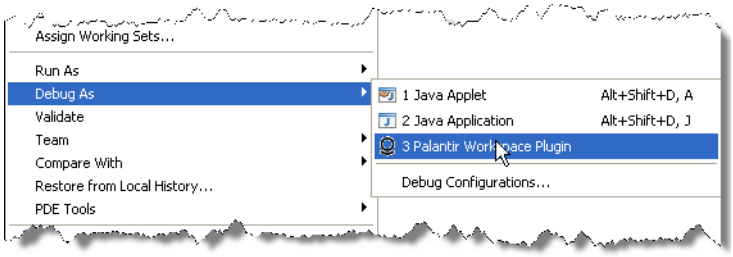
8. Verify that your helper appears and that it contains what you expect.



At this point, you can continue your development. When you are ready to see your changes in the Workspace, close any open Workspace clients and choose **Run As > Palantir Workspace Plugin** again.

Debugging the Sample Plugin

You can use the standard Eclipse debugging features to debug your plugin. To do this from the Package Explorer view, right click the project folder and select **Debug As > Palantir Workspace Plugin** from the pop-up menu.



The system starts the Eclipse debugger and launches the Workspace. When you launch your plugin from Eclipse, monitoring via JConsole is enabled by default. Eclipse also supports hot code replacement as a standard technique in the Java VM. See your Eclipse documentation for more information about using this feature.

Note: The **Debug As** option only operates with Workspace or client plugins. For a list of the plugin types see component Element on page 36.

Developing Other Plugin Types with QuickStart

A client plugin is one type of plugin you can create. You can also create the following types of plugins:

Plugin Type	Description
client	Runs as a headless client, a Workspace helper, or a Workspace application. Includes plugins that run as a client to the Dispatch Server(s) such as clients that crawl, extract, and transform data.
ontology	Adds functionality to customize property components such as validators, formatters, parsers or approxes.
Dispatch Server	Extends functionality on the Dispatch Server.
Horizon Server	Extends the functionality of the Horizon Server.
Spring XML	Makes use of Spring functionality to customize geographic features and highlight rules.

The procedure for deploying a plugin in the QuickStart environment depends on the type of plugin you are deploying. The following sections detail how to deploy in the QuickStart development environment. If you want to deploy in a staging or production environment, see Deploying Custom Code in a Staging or Production Environment on page 43.

Dispatch Server, Horizon, or Spring Plugin Development with QuickStart

This section explains how to create and deploy a non-ontology plugin in the QuickStart development environment.

1. Create a Palantir helper project in Eclipse.

   You can use the project wizard provided by Palantir. See Starting a New Client Plugin Project on page 17 for an explanation on how to do this.

2. Write your plugin code implementing one of the Dispatch Server, Horizon, or Spring plugin interfaces.

   For a list of interfaces that you can implement see component Element on page 36.

3. Right click the `build.xml` file and choose **Run As > Ant Build**.

   After the build completes, there is a `.par` file in the distribution directory.

4. Open a DOS command line.

5. Navigate to the `QUICKSTART_INSTALLDIR\bin` folder.

6. Ensure the Dispatch Server is running.

7. Run the `dist_manage_plugins.bat` command to install your plugin.

   ```
   dist_manage_plugins.bat -i -f par_jar_filename
   ```

Ontology Plugin Development in QuickStart

This section explains how to create and deploy an ontology plugin in the QuickStart development environment.

1. Create a Palantir helper project in Eclipse.

   You can use the project wizard provided by Palantir. See Starting a New Client Plugin Project on page 17 for an explanation on how to do this.

2. Write your project's code.

   This includes updating the ptplugin.xml file for your project. For details on that file format, see The ptplugin.xml File Reference in Chapter 3.

3. Right click the `build.xml` file and choose **Run As > Ant Build**.

4. Start the Dynamic Ontology Manager installed with the QuickStart.

   You can double clicking the **Palantir Dynamic Ontology Manager** desktop icon or locate the application through the Windows Start menu. If you did not install the desktop icons, you can also use the `QUICKSTART_INSTALLDIR\bin\dist_admin_editor.bat` script.

5. Select **File->Import from DB**.

   The **File** dialog appears.

6. On the **File** dialog, enter a file name, choose a location for the ontology file, and click **Ok** to save the file.

7. Open the **Plugin Editor** page and click **Add Plugin**.

   An **Open** dialog appears.

8. In the **Open** dialog, navigate to the PAR file containing the plugin you created in Step 3, select it, and click **Open**.

   The Dynamic Ontology Manager adds the PAR file to your ontology and saves the change to the ontology.

9. Make use of your plugin in your ontology.

   For example, if you plugin is a validator, add it to a property.

10. Save your ontology changes.

11. Select **File->Export to DB** to deploy the changed ontology to the Dispatch Server's database.

12. Restart the Dispatch Server installed with the QuickStart installation by double clicking the **Start Palantir Server** desktop icon.

    If you did not install the desktop icons, you can invoke the `QUICKSTART_INSTALLDIR\bin\dist_palantir_server.bat` script.

13. Wait for the server to finish starting and start the Workspace.

    You can double click the **Start Palantir Workspace** desktop icon or use the options on the Window Start menu. If you did not install the desktop icons, you can invoke the `QUICKSTART_INSTALLDIR\PalantirWorkspace.exe` command.

14. Test your plugin in the context of the Workspace.

Trouble Shooting Eclipse Plugin Problems

If you are running Palantir Workspace Plugin out of Eclipse, and you get a message similar to the following:

```
Exception in thread "Name" java.lang.OutOfMemoryError: PermGen space
```

Then you need to increase your PermGen space. To do this, choose:

1. Right -click your project and choose **Run As > Run Configurations**.
2. Select your project under **PalantirWorkspacePlugins** folder.
3. On the **Extra VM Arguments** tab, edit the `-XX:MaxPermSize=512m` argument and increase the value.
4. Press `Apply` to save your changes and choose **Run**.

3 Working with ptplugin.xml Files

This chapter provides detailed information you need to know when working with `ptplugin.xml` files. You should read this chapter if you are deploying a helper, an application, or other custom code. The following topics are discussed:

- Understanding Plugins and the Plugin Manager
- The ptplugin.xml File Reference
- Example ptplugin.xml Files
-

Understanding Plugins and the Plugin Manager

This section describes concepts and structures important for your understanding of working with plugins. It contains the following topics:

- The Plugin Manager and Plugin Management
- The Role of the ptplugin.xml File
- Packing Files into a PAR or JAR
- Understanding Plugin State and the Plugin APIs

The Plugin Manager and Plugin Management

The Palantir Platform contains a plugin manager framework that allows you to load plugins in an isolated manner. This isolation reduces namespace conflicts and supplies the means for Palantir to apply additional class-loading security.

Palantir supports several different types of plugins. You can create these types of plugins:

Plugin Type	Description
client	Runs as a headless client, a Workspace helper, or a Workspace application. Includes plugins that run as a client to the Dispatch Server(s) such as clients that crawl, extract, and transform data.
ontology	Adds functionality to customize property components such as validators, formatters, parsers or approxes.
Dispatch Server	Extends functionality on the Dispatch Server.

Plugin Type	Description
Horizon Server	Extends the functionality of the Horizon Server.
Spring XML	Makes use of Spring functionality to customize geographic features and highlight rules.

All plugins use of the plugin manager framework for loading. This framework relies on the `ptplugin.xml` file to describe the plugin to the plugin manager. You can package your code for delivery as a a Palantir ARchive or PAR file or a Java ARchive or JAR file. If you use Palantir's Eclipse Helper Plugin to develop your plugin, the default `build.xml` packages your helper as a PAR file.

How you manage and deploy a plugin depends on what area of the Palantir Platform you are customizing. With the exception of ontology plugins, you manage Palantir plugins from the Palantir Enterprise Manager (PEM) interface. Ontology plugins, because the rely on and modify the dynamic ontology, are loaded and managed from Palantir's Dynamic Ontology Manager.

For a discussion of deploying the different types of plugins, see Deploying Custom Code in a Staging or Production Environment on page 43.

The Role of the ptplugin.xml File

The `ptplugin.xml` supports the Plugin Manager in securing the Palantir Platform from inadvertent use by plugin code. The `ptplugin.xml` file defines your custom code components and what interfaces they implement. Even if your plugin implements multiple components you need to write only one `ptplugin.xml` file. The `ptplugin.xml` file allows custom code to:

- contain different versions of the same third-party libraries used by the Palantir Platform or other plugins
- start via Palantir Web Start without extra coding
- specify plugin preferences that you can query at runtime or change without modifying the ontology

A `ptplugin.xml` file also removes the requirement to sign plugin JARs. For a full discussion of the elements in a `ptplugin.xml` file see The ptplugin.xml File Reference on page 34.

Packing Files into a PAR or JAR

After you are satisfied with your custom code, you will package the compiled classes, libraries, and resources along with the `ptplugin.xml` file into a Palantir Archive (`.par` file) or a Java archive (`.jar` file).

You create your PAR file with a ZIP or JAR archive command and rename the resulting file with a `.par` extension. The Plugin Manager expects the contents of every PAR file to have the same structure. The following table describes the expected structure:

Table 6 PAR File Structure

Folder	Description
`/`	No files at this level; it contains the subdirectory folders.
`classes`	Contains the following: • the `ptplugin.xml` file • a source code hierarchy starting with `com/companyname/...` and so forth. • material from your project's `resources` directory such as custom images or icons
`lib`	Third party source code in JAR format.

When zipping together your PAR file start from within the root, that is at the same level as `classes` and `lib`.

The following table illustrates the structure expected in a `.jar` file format:

Table 7 JAR File Structure

Folder	Description
`/`	The root folder. Should contain the `ptplugin.xml` file and any other resources your project requires.
`com`	Source file tree.
`META-INF`	The JAR manifest.

If you have multiple plugins, you can put each plugin in a separate archive (PAR or JAR) or all of them into single archive. Palantir recommends that all custom code be stored in one single archive package. The advantage of the single file method is that you only need to write one `ptplugin.xml` file. For example, if you have multiple helpers that use the same third party library, you would package all the helpers and the library together in a single PAR file.

Note: If you are deploying your custom code for use with the `kiteXmlCrawl.sh` command, do not package it in a PAR file. That format is not supported in this release. Package your code in a regular JAR file for use with the `kiteXmlCrawl.sh` command.

Understanding Plugin State and the Plugin APIs

After you load your plugin and export the ontology to the Dispatch Server, the system maintains and manages information about the plugin's state. If your plugin needs to create and manage state information, you should use the `PalantirClientContext` API. This API provides simple utilities for loading and working with application state.

Preferences represent a special case of application state. Preference state information is separate from any state your plugin's code might create and manage for its own purposes. When a client connects to the Dispatch Server, the system passes the client the current plugin preferences.

For detailed information about programming a plugin with state and preferences, see Chapter 8, Managing Application State and Plugin Preferences.

The ptplugin.xml File Reference

This section discusses the fields and attributes found in the `ptplugin.xml` file. Use the topics in this section to learn how to write your `ptplugin.xml`:

- uri and displayName Attributes
- version Element
- targetVersion Element
- component Element
- preferences Element

A Simple ptplugin.xml File

The following example illustrates a `ptplugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantir.com/schemas/ptplugin/v2.1.0">
    <uri>com.acme.plugin.helpers.Samples</uri>
    <displayName>MySamples</displayName>
    <version>
      <major>1</major>
      <minor>0</minor>
    </version>
    <targetversion>3.0.3</targetversion>
    <component>
      <interface>com.palantir.api.workspace.HelperFactory</interface>
      <implementation>com.acme.sample1.Plugin</implementation>
    </component>
    <component>
      <interface>com.palantir.api.workspace.HelperFactory</interface>
      <implementation>com.acme.sample2.Plugin</implementation>
    </component>
    <preferences>
       <preference uri="location">
       <name>Location</name>
       <type>string</type>
       <default><string>Sample Plugin</string></default>
     </preference>
     <preference uri="initialcount">
       <name>initial count</name>
       <type>integer</type>
       <default><integer>42</integer></default>
```

```
    </preference>
    <preference uri="config">
      <name>ConfigFile</name>
      <type>file</type>
      <default><filename>/path/to/default/config.txt</filename></default>
    </preference>
   </preferences>
</ptplugin>
```

uri and displayName Attributes

Each `ptplugin.xml` file contains a `uri` and `displayName` attribute. These are both required. The `uri` is the package's unique resource name. The `uri` can contain the characters a-z, A-Z, 0-9, and a . (dot). This value is required for Palantir Platform APIs. For example, the Palantir Platform uses the `uri` to register your plugin for application state persistence. A plugin `uri` cannot exceed 300 characters.

The `displayName` is any valid string value. String values cannot exceed 500 characters.

version Element

Each plugin has a required `version` element. You determine the version of your plugin. The version can contain `major` and `minor` integer values. It can also contain optional `maintenance` and `build` values.

```
<version>
   <major>3</major>
   <minor>1</minor>
   <maintenance>12</maintenance>
   <build>652</build>
</version>
```

You should change the version value each time you modify and then release your plugin. You can support two versions of the same ontology plugin in the ontology. The system determines which plugin to used based on the `version` numbers. It uses the newest compatible version and ignores older versions. Non-ontology plugins cannot support two versions of the same plugin.

You cannot disable a plugin, you must remove it. If it is convenient, you can keep a previous (older) version of a plugin in an ontology. For example, you might want to do this if you are testing a new version. To revert to the old plugin, simply remove the new version.

Keep in mind, keeping multiple versions of an ontology plugin increases an ontology's size. You should limit this practice to development or staging environments. An ontology is downloaded using Web Start so larger ontology sizes can reduce performance. Best practice is only one version of an ontology plugin in a production environment.

targetVersion Element

The `targetVersion` element is required. This element defines the Palantir version in which the plugin is intended to run. The minimum acceptable version is 3.0.3.0 and the maximum is 3.1.1.0 inclusive. Palantir version numbers are four digits long. If you do not specify a digit, the system assumes a 0 (zero).

Palantir checks the version number against the minimum and maximum acceptable version numbers when you load it. If the value you specify does not fall into this range, your plugin will load but cannot be enabled. If you do not specify a `targetVersion`, Palantir assumes the default target version.

Caution: You should always specify a `targetVersion` and not rely on the default. Currently, the default version number is 3.1.1.0.

component Element

Most plugins require a `component` element for Palantir to recognize them. The exceptions are custom plugins that rely on the Spring dependency injection paradigm; these do not require a `component` element. Currently, only the custom highlighting rules or tile source plugins use this technology. All other plugins require a `component` element.

You can include any number of component elements in your `ptplugin.xml` file. If your plugin has multiple components, you will have one `component` section for each. If your plugin contains only configuration files rather than executable code, no `component` tags are required.

The `component` element has two child elements `interface` and `implementation`. The `interface` is the fully qualified class name (package and class name) of the interface the plugin implements. The `implementation` is the fully-qualified class name of the implementing class. This is a required element.

The following tables lists the `component` elements you can specify and what type of plugin

Ontology Plugin Interfaces
`com.palantir.commons.security.ClassificationRulesEnginePlugin`
`com.palantir.commons.security.ClassificationStringHandlerPlugin`
`com.palantir.services.impl.property.formatter.modifier.ITokenModifier`
`com.palantir.services.impl.property.IPropertyApproxGenerator`
`com.palantir.services.impl.property.IPropertyFormatter`
`com.palantir.services.impl.property.IPropertyMaker`
`com.palantir.services.impl.property.IPropertyValidator`

Client Plugin Interfaces
com.palantir.api.auth.InvestigationDefaultsProvider
com.palantir.api.workspace.ApplicationFactory
com.palantir.api.workspace.datasource.IDataSourceMaker
com.palantir.api.workspace.HelperFactory
com.palantir.api.workspace.MenuInterface (uses Spring)
com.palantir.commons.security.ClassificationChooserFactoryPlugin
com.palantir.gis.gazetteer.Gazetteer
com.palantir.gis.gazetteer.SimpleGazetteer
com.palantir.gis.integration.GISExportPlugin
com.palantir.api.dataintegration.crawl.Crawler
com.palantir.api.dataintegration.detect.Detector
com.palantir.api.dataintegration.extract.Extractor
com.palantir.api.dataintegration.transform.Transformer

Dispatch Plugin Interface
com.palantir.services.pam.PamMetricRequestHandler

Horizon Plugin Interfaces
com.palantir
com.palantir.api.horizon.v1.formula.OperationArgument
com.palantir.api.horizon.v1.formula.Operator
com.palantir.api.horizon.v1.formula.OperatorFactory
com.palantir.api.horizon.v1.view.View
com.palantir.api.horizon.v1.view.ViewFactory
com.palantir.api.objectexplorer.v1.menu.OperationsMenuItem
com.palantir.api.objectexplorer.v1.menu.OperationsMenuItemFactory
com.palantir.api.objectexplorer.v1.vis.Visualization
com.palantir.api.objectexplorer.v1.vis.VisualizationFactory

preferences Element

The `preferences` element is an optional element that stores global parameters for use by your plugin. Parameters are variables that your plugin can access at runtime. When a user opens the Workspace, the system downloads the current plugin preferences into the user's runtime environment. Only plugins that implement the following interfaces can support parameters:

- `com.palantir.api.workspace.HelperFactory`
- `com.palantir.api.workspace.ApplicationFactory`
- `com.palantir.api.auth.InvestigationDefaultsProvider`
- `com.palantir.api.workspace.map.Gazetteer`

The `preferences` element contains one or more `preference` child elements. The following example illustrates a `preferences` element:

```
...
<preferences>
  <preference uri="com.salmon.preference">
    <name>human readable name</name>
    <type>string</type>
    <default><string>Default value</string></default>
  </preference>
  <preference uri="com.salmon.preference2">
    <name>human readable name</name>
    <type>integer</type>
    <default><integer>42</integer></default>
  </preference>
  <preference uri="com.salmon.preference3">
    <name>human readable name</name>
    <type>file</type>
    <default><filename>/path/to/default/file.txt</filename></default>
  </preference>
</preferences>
...
```

Note: You cannot exceed 10,000 preferences in your `ptplugin.xml` file.

Each `preference` element has a single, required `uri` attribute that is the `preference` identifier. Every plugin has its own preference namespace so the `uri` need only be a simple name. The plugin's own URI is implicitly appended to the preference URI.

A `preference` contains the following child elements:

Table 8 The preference Child Elements

Element	Description
`name`	Specifies a human readable name for the preference. This is required.

Table 8 The preference Child Elements (continued)

Element	Description
`type`	Specifies a valid type for the plugin. The type can be: • `integer` • `string` • `filename` Filename preferences are downloaded to each client machine by the system when a user logs on. A `type` is required.

`default`	Specifies the initial setting for a `preference`. You specify a default's data type by enclosing the value with the proper `type` tag. The `type` format is as follows:
	`<default><type>value</type></default>`
	For example:
	`<default><integer>489</integer></default>`
	These `default` values are optional. If you do not specify a `default` value, the system uses a system-defined default.
	The system default for an integer is 0. A `string` value cannot exceed 500 characters; the default is an empty string. The file identified by the `filename` value cannot exceed 60 KB in size; the default is an empty file (byte[0] length).

The `ptplugin.xml` file specifies the initial default set of plugin preferences. Once a plugin is loaded, preference values are maintained in the backend database. Changes made with the command line replace the initial value(s) specified in the plugin's `ptplugin.xml` file.

Once you have loaded a plugin and made a change with the CLI, the only way to change a preference using the Palantir Platform is with the CLI or an API call. You cannot delete a preference with the CLI, through an API call, or by reloading a `ptplugin.xml` file. As global parameters, preferences apply to all users; changing a preference value changes it for all users.

Note: Defaults that are unchanged by the CLI are updated by new values in the `preferences` section of the `ptplugin.xml` file.

Keep in mind, the association between the plugin URI and the preference URI persists in the database. This means if you remove a preference from the `ptplugin.xml` file and add a preference with the same URI back through a later version of your plugin, the platform will reuse the last value stored in the database regardless of any changes in the `ptplugin.xml` file.

After you have installed your plugin, you can change its preferences, using the `pluginPreferencesAdmin` CLI. If you use the `pluginPreferencesAdmin` CLI to change the default settings, the system will store these into the repository for you. Users who have the Workspace open when you change preferences will not receive the new values until they relog. See the *Palantir Administrative CLI Reference* for detailed information about this CLI.

Example ptplugin.xml Files

This section contains examples of `ptplugin.xml` files.

The Typical Case

The following example illustrates a plugin that implements the `IPropertyValidator` interface and is targeted to the 2.3.3 version of Palantir:

```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantir.com/schemas/ptplugin/v2.1.0">
    <uri>com.palantir.custom.MyCustomAddressValidator</uri>
    <displayName>Custom Validator</displayName>
    <version>
      <major>1</major>
      <minor>0</minor>
    </version>
    <targetversion>3.0.3</targetversion>
    <component>
      <interface>
          com.palantir.services.impl.property.IPropertyValidator
      </interface>
      <implementation>
          com.palantir.custom.MyCustomAddressValidator
      </implementation>
    </component>
</ptplugin>
```

No Component Required

The following example illustrates a plugin that would accompany a custom syntax highlighting plugin. This plugin uses Spring injection dependency paradigm, so it does not require a `component` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantir.com/schemas/ptplugin/v2.1.0">
    <uri>com.palantir.custom.SyntaxHighlights</uri>
    <displayName>Custom Validator</displayName>
    <version>
      <major>2</major>
      <minor>1</minor>
    </version>
</ptplugin>
```

Notice that this example does not include a `targetVersion` element. The version of this plugin will default to 2.1.3.7, which is too low for this release. This `ptplugin.xml` is disabled by the system on load. Increase the `targetVersion` to at least a 3.0.3.X version number.

Icon Library Plugin

The following example illustrates a plugin that would accompany a set of graph custom icons and extend the ***icon library*** available to Workspace graphs in a Palantir Platform deployment. An ***icon library plugin*** is a special type of plugin that contains images for Workspace graph node icons. This type of plugin does not require a `component` element, must be packaged in a JAR (not a PAR), must reference a file called `IconLibrary.xml`, and has other specialized requirements for file structure and `ptplugin.xml` format.

```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantirtech.com/schemas/ptplugin/v2.1.0">
    <uri>com.cool.graph.SampleIconLibrary.2525b</uri>
    <displayName>2525b icons</displayName>
    <version>
      <major>1</major>
      <minor>0</minor>
    </version>
    <targetVersion>3.2.1</targetVersion>
</ptplugin>
```

To learn more about setting up and deploying graph icon library plugins, see Managing Icon Libraries for Graph Presentations in *Palantir Administrator's Guide*. The subtopic, Defining an Icon Library Plugin includes a sample walkthrough of all components needed in the JAR.

To learn more about using custom icons in Workspace graphs, see Customizing Node Icons.

4 Deploying Custom Code in a Staging or Production Environment

This section details the steps you must follow to deploy your custom code from your staging system to a production system. The steps for deploying are dependent on the type of plugin you are deploying.For information about deploying in a development environment see Chapter 2, Making Your First Custom Code Project.

Building Your Code and Generating a PAR File

Palantir assumes you are using Eclipse to build and debug your code. If you have installed the Eclipse plugin (see Setting up the Palantir Eclipse Plugin on page 12) you can use its functionality to help you build your project. You can use **File > New > Project** from the main Eclipse menu to run the Palantir project creation wizard.

If you have an existing project, you can choose **Run As > Ant Build** to build your code. The build.xml file that ships with Palantir's Eclipse plugin, builds a PAR file. When the build successfully completes, there should be a new `dist` folder in your project. The folder contains the PAR file for your plugin.

Deploying Non-Ontology Plugins

If you are working with a Spring, client, Dispatch Server, or Horizon Server plugin, you deploy these plugins using the Palantir Enterprise Manager. This section describes the processes you can use for deploying non-ontology plugins:

- Installing a Non-Ontology Plugin
- Updating a Non-Ontology Plugin
- Exporting One or All Non-Ontology Plugins
- Removing a Non-Ontology Plugin

You can also use a CLI to perform the procedures detailed in this section, see the managePlugins CLI in the *Palantir Administrative CLI Reference*.

Installing a Non-Ontology Plugin

You can use the Enterprise Manager GUI **Plugins** application to install *non-ontology* plugins in your deployments. Before you install a plugin or plugins, make sure you have the appropriate source code package as described in the *Palantir Developer's Guide*. Plugin packages can be in PAR or JAR file formats.

You can install all non-ontology plugins without restarting the Dispatch Server or other servers. If plugin operations are performed while the Workspace or other clients are running, Workspace users must restart the clients to get the new client plugins (for example, for Workspace helpers).

Log into the Enterprise Manager GUI, select a deployment, and follow this procedure to install plugins:

1. In the Enterprise Manager GUI, select the **Plugins** application from the left side of the page under **Administration**.

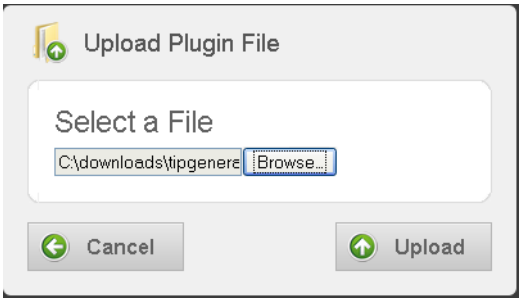
   The Plugins page lists the currently installed Dispatch Server, and Raptor Server plugins.

2. Click the **Install** button.

   The **Install Plugin** dialog opens and provides the option to upload a file.

3. Click the **Upload File** button.

   The **Upload Plugin File** dialog is displayed.



4. Click **Choose File** to get a file browser, and use it to navigate to and select a plugin package file to upload.

   Valid plugin package files to choose from are a JAR or PAR file containing a single plugin, or a zip archive that contains multiple plugins.

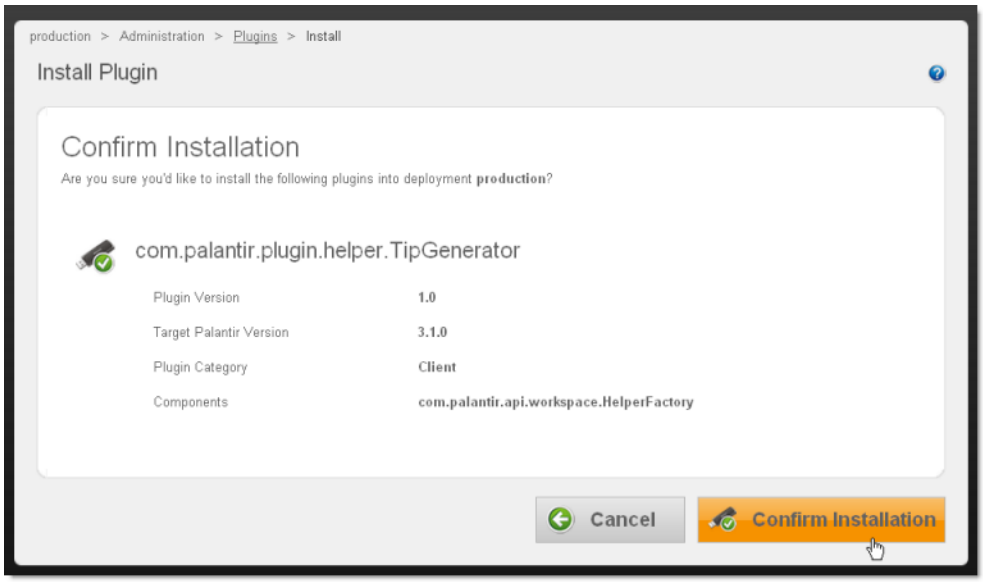
Tip: For details on valid contents of JAR, PAR, or ZIP files see the *Palantir Developer's Guide*.

5. Click the **Upload** button to install the plugin.

   The system uploads the plugin, and lists details such as plugin URI, plugin version, target Palantir version, plugin category, and components. For zip files with multiple plugins included, details for each plugin are provided.

   This step includes validation checking to verify that the plugin implements an allowed non-ontology plugin interface and specifies a target platform version supported by the current deployment. For a plugin install, the system also checks to make sure the plugin is not already installed. (You can *update* currently installed plugins.)

6. Review the summary of uploaded plugin(s).



7. Click the **Confirm Installation** button to install the plugin(s).

   The system installs the plugin(s), and indicates successful completion of the task when done.

8. Click the **Go Back** button to get back to the Plugins page.

   This page lists the currently installed plugins. If your new plugin(s) installed successfully, they are shown in this list.

   By default, when you load a plugin, all the users in the Everyone group have access to the plugin. You change a plugin's access from its **Details and Access** page.

   You can click any plugin URI in this list to get more information about the plugin, including plugin and target platform version, plugin category, the interface its implements, and created/modified dates. Also, you can export, update, or remove the selected plugin directly from this dialog.

Updating a Non-Ontology Plugin

You can use the Enterprise Manager GUI **Plugins** application to update non-ontology plugins. For an update, the new plugin must have an identical version number and a matching URI. Updating a plugin causes the system to update the plugin's code in the Palantir Platform's application state. For Dispatch Server and Horizon Server plugins, the system automatically updates any dependant architecture components. Client applications, such as Palantir Workspace, automatically receive updates.

You can install all non-ontology plugins without restarting the Dispatch Server or other servers. If plugin operations are performed while the Workspace or other clients are running, users must restart the clients to get the client plugin updates (for example, for Workspace helpers). Update does not change the access list for a plugin. The users and groups that were allowed access before the update continue to have access to the newly updated plugin.

Note: If you are updating or installing newer versions of Horizon Server plugins while clients using that plugin are online, the update might cause the server to get out of sync with the version on the client. If this happens, the client might receive a serialization exception; restart the client to resolve the issue.

Log into the Enterprise Manager GUI, select a deployment, and follow this procedure to update plugins:

1. In the Enterprise Manager GUI, select the **Plugins** application from the left side of the page under **Administration**.
2. Select the **Plugins** application from the left side of the page.

   The Plugins page lists the installed client, Dispatch Server, and Horizon Server plugins.
3. Identify an individual plugin on the list and click its **Update** button.

   The system displays the Update Plugin dialog.

   From this point on, the update procedure is very similar to installing a new plugin.

Note: If a plugin is installed on more than one server (for example, on Dispatch and Raptor Servers), clicking update results in a different dialog than is shown here. In this case, you can choose to:

- update the plugin on just the selected server
- update the plugin on all servers containing a plugin with this version and URI.

Each plugin is identified by the plugin URI and version shown in its `ptplugin.xml` file.

---

4. Click the **Upload Plugin** button.

   The Upload Plugin File dialog is displayed.
5. Click **Choose File** to get a file browser, and use it to navigate to and select a plugin package file to upload.

   Valid plugin package files to choose from are a JAR or PAR file containing a single plugin, or a zip archive that contains multiple plugins.
6. Click the **Upload** button to install the new plugin.

   The system uploads the plugin, and lists details such as plugin URI, plugin version, target Palantir version, plugin category, and components. For zip files with multiple plugins included, the system provides details for each plugin.
7. Review the summary of uploaded plugin(s).
8. Click the **Update** button to install the plugin updates.

   The system installs the plugin(s), and indicates successful completion of the task when done.

9. Click the **Go Back** button to get back to the Plugins page.

   This page lists the currently installed plugins. If your new plugin(s) updated successfully, they are shown in this list.

Exporting One or All Non-Ontology Plugins

You can use the Enterprise Manager GUI **Plugins** application to export an individual plugin or all installed plugins. Exporting a plugin does not remove it from a deployment. The system places exported plugins to the default downloads directory specified for your web browser (this might be on the system that hosts the Palantir Enterprise Manager, or on another system).

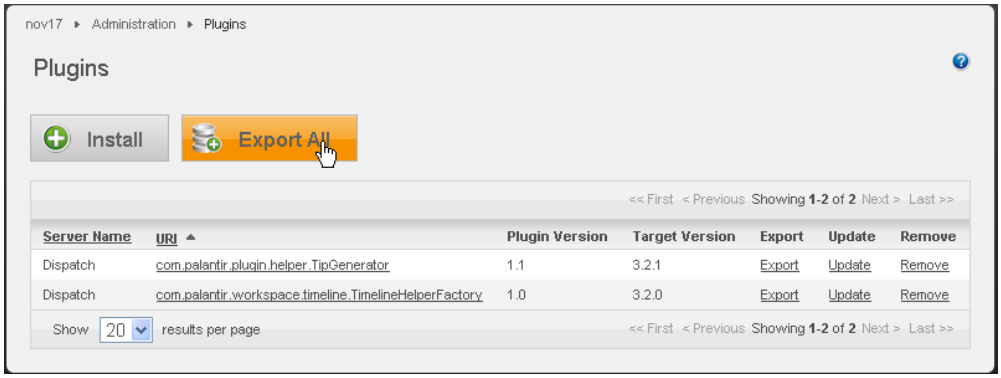
If you export an individual plugin, the system saves it as a PAR file or JAR file, according to its original install format. If you export all the plugins together, the system exports a single `.zip` file that can include both PAR and JAR files.

Log into the Enterprise Manager GUI, select a deployment, and follow this procedure to export plugins:

1. In the Enterprise Manager GUI, select the **Plugins** application from the left side of the page under **Administration**.

   The **Plugins** page lists the installed Dispatch Server, client, and Horizon Server plugins.

2. Do one of the following:
   - Identify an individual plugin on the list and click its **Export** button.
   - Click the **Export All** button.



Note: If no plugins are installed, the **Export All** option is unavailable.

Removing a Non-Ontology Plugin

You can use the Enterprise Manager GUI **Plugins** application to remove plugins at any point.If you remove a plugin, the access for that plugin is also removed. If you install the plugin again later, the list of users and groups with plugin access are not restored.

Log into the Enterprise Manager GUI, select a deployment, and follow this procedure to remove plugins:

Note: You can remove client plugins like Workspace helpers without restarting the servers, but running Workspace clients will need a restart to properly reflect the currently installed plugins.

1. In the Enterprise Manager GUI, select the **Plugins** application from the left side of the page under **Administration**.

   The **Plugins** page lists the installed client, Dispatch Server, and Horizon Server plugins.

2. Identify an individual plugin on the list and click its **Remove** button.

   The system displays the **Remove Plugin** dialog.

   Note: If a plugin is installed on more than one server (for example, on Dispatch and Raptor Servers), you can choose to:

   - remove the plugin on just the selected server
   - remove the plugin on all servers containing a plugin with this URI.

   Each plugin is identified by the plugin URI and version shown in its `ptplugin.xml` file.

3. Click the **Remove** button to remove the plugin.

   The system removes the plugin.

4. Click the **Go Back** button to get back to the **Plugins** page.

   This page lists the currently installed plugins. If the plugin was removed, it will no longer show in this list.

Deploying Ontology Plugins

Deploying an ontology plugin is a process that requires you to perform multiple procedures. This section details the procedures you perform when deploying ontology plugins:

- Getting an Ontology Using the Palantir Enterprise Manager
- Adding Plugins to an Ontology
- Updating the Palantir Servers with Your Changes
- Deleting Ontology Plugins
- Replacing an Ontology Plugin

Getting an Ontology Using the Palantir Enterprise Manager

You add a plugin to an existing ontology. Before you can do this you must save a copy of the current ontology. To save an ontology file by using the Enterprise Manager task:

1. Log into the Enterprise Manager as the **admin** user**.**
2. From the **Tasks** selection in the navigation pane, click **Export Ontology**.

3. Respond to the prompts to finish this task.

   You must select a Dispatch Server or Raptor Server and provide a file name ending in `.ont`. The ontology is exported from the selected server's database.

4. Click **Submit**.

   You have the opportunity to review your selections. If you need to change any option, click **Back** and make your changes.

5. Click **Start** to begin task processing.

   The task writes your ontology to the `.ont` file, located in the `/ontology` directory of the Enterprise Manager installation.

   Note: If you rename the ontology file, do not use special characters in the file name for the exported ontology. ~ ! @ # $ % ^ & * ( ) + = [ ] | : " < > ? ' \ / are not allowed.

6. Note the file's name and location.

   You might need to copy the file to your local machine. You will need the name and location of the file to edit the ontology using the Dynamic Ontology Manager. You should make a backup of this file before making changes.

Adding Plugins to an Ontology

You must add your custom code to the Dynamic Ontology and then make it available to a running Dispatch Server instance using the Palantir Enterprise Manager. This procedure assumes the Enterprise Manager is running. If it is not running, you will not be able to complete this procedure. Refer to the *Enterprise Manager: Administration Applications* for instructions about starting the Enterprise Manager.

You add the packaged plugin code via the Dynamic Ontology Manager and export the ontology to the Dispatch Server. To add a custom PAR file to your ontology:

1. Open the Plugin Editor.
2. Click **Add Plugin**.

   An **Open** dialog appears.

3. In the **Open** dialog, choose the PAR file containing the plugin you want to add.
4. Click **Open**.

   The Dynamic Ontology Manager adds the PAR file to your ontology and saves the change to the ontology.

At this point, you can make use of your ontology plugin in your ontology. For example, if you have a custom validator, you can add it to a property. Then, when you are ready, save your ontology and exit the Dynamic Ontology Manager.

Updating the Palantir Servers with Your Changes

To update the ontology on the Dispatch Server(s):

1. Open your bookmark to the login page for the Enterprise Manager GUI.

   If you do not have a bookmark, copy a URL from the *PEM_INSTALLDIR*/log/pem-console.log file on the Enterprise Manager, paste it into your favorite browser, and log in as the Enterprise Manager **admin** user**.**

2. Stop the Palantir servers.

   From the **Tasks** selection in the navigation pane, click **Stop Servers** and follow the prompts to complete the task.

3. Verify the database servers are up and running.

4. From the **Tasks** selection, click **Update Ontology**.

5. Select your Dispatch Server.

   If you have installed clustered Dispatch Servers, you only need to deploy the ontology to one of the servers. The servers share the ontology in their shared database.

6. Specify your .ont file.

7. Choose whether to reindex the databases underlying the Search Server. You need to reindex if you have made any of the following changes:

   - changed a display formatter
   - added a validator
   - added an approx

   To learn more about reindexing, see the *Palantir Administrative CLI Reference*.

8. Click **Submit**.

   You have the opportunity to review your selections. If you need to change any options, click **Back** and make your changes.

9. Click **Start** to begin task processing.

   The Enterprise Manager updates the ontology in the Dispatch Server's database and reindexes the search databases if you chose that option.

10. If the Raptor module is part of your deployment, you must update the ontology for the Raptor Search Engine. Repeat Step 4 through 9, choosing your Raptor Search Engine server instead of the Dispatch Server.

    The Enterprise Manager updates the ontology in the Raptor Server's database and reindexes the Raptor Search Nodes' database if you chose that option.

11. Start the Palantir servers.

    From the **Tasks** selection, click **Start Servers** and follow the prompts to complete the task.

12. Notify your analysts to restart their client application.

    The Workspace loads custom plugins on startup, so analysts must restart it to get access to the new plugin.

Deleting Ontology Plugins

Deleting a plugin requires you to complete two procedures, removing the plugin through the Dynamic Ontology Manager and updating the ontology on the servers.

To delete an ontology plugin:

1. Remove the instances in the ontology that make use of your ontology plugin.

   For example, if you are using a custom number formatter for a property, you need to remove that formatter using the Property Editor.
2. In the Plugin Editor, select the PAR file containing the plugin you want to delete.
3. Click **Delete Plugin**.

   A confirmation dialog appears.
4. Click **OK** to delete the PAR file.
5. Update your server ontologies according to Updating the Palantir Servers with Your Changes on page 51.

Replacing an Ontology Plugin

Typically, you replace a plugin when its code has changed. Your new plugin will have a new version and the same URI. You do not need to remove the old versions of the ontology plugins from your staging instance. The system automatically runs the latest version of a plugin. You can also replace a plugin with one that has the same version number and URI. This procedure assumes you delete and then add the plugin in a single operation.

Caution: If you choose to replace a plugin in a single operation, be certain that you have the code for the original (deployed) version backed up somewhere. If you do not, you run the risk of losing your original plugin.

To use the Plugin Editor to replace a custom plugin that you have deployed:

1. Click **Add Plugin**.

   The system displays the **Open** dialog.
2. Choose the PAR file containing the plugin you want to deploy as a replacement.
3. Click **Open**.

   The Dynamic Ontology Manager removes the old PAR file and adds the replacement PAR to your ontology
4. Select the PAR file containing the plugin you want to replace.
5. Click **Delete Plugin**.

   A confirmation dialog appears.
6. Click **OK** to delete the PAR file.

7. Save the ontology.

   You can exit the Dynamic Ontology Manager if you wish. Before your change can take effect, you must update the ontology on the Dispatch Server and the Raptor Search Engine (if you are using it), and then restart the Palantir servers.

8. Update your server ontologies according to Updating the Palantir Servers with Your Changes on page 51.

5 Programming Infrastructure

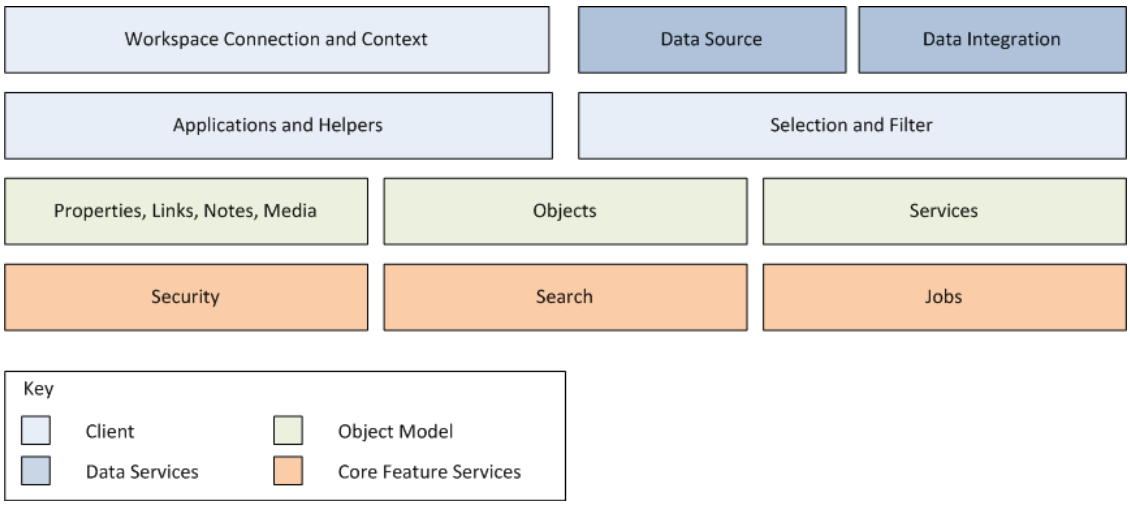
This chapter describes Palantir's programming infrastructure. You will find in this a structural overview of the infrastructure, summaries of the major API categories, and descriptions of the key APIs you will find in each. The following topics appear in this chapter:

- Structural Overview of APIs
- The Client APIs
- Data Services APIs
- Object Model APIs
- Core Feature Services
- Geographic Information
- Web REST APIs

Structural Overview of APIs

The following figure depicts the general components that make up the core APIs that can be found in the Palantir Platform:

Figure 1 The Palantir Public API



There are also some additional APIs specific to working with geographic systems, the Palantir Enterprise Manager, and of course with the Web application.

The Client APIs

The client APIs supply interfaces for integrators to program to the Palantir Workspace and command-line clients.

```
com.palantir.api.client
com.palantir.api.exception
com.palantir.api.workspace
com.palantir.api.workspace.applications
com.palantir.api.auth
com.palantir.api.workspace.dataitem
com.palantir.api.workspace.documentviewer
com.palantir.api.workspace.documentviewer.highlightrules
com.palantir.api.workspace.event
com.palantir.api.workspace.filters
com.palantir.api.workspace.filters.model
com.palantir.api.workspace.graph
com.palantir.api.workspace.helpers
com.palantir.api.workspace.map
com.palantir.api.workspace.progress
com.palantir.api.workspace.selection
com.palantir.api.objectexplorer.v1
com.palantir.api.objectexplorer.v1.dnd
com.palantir.api.objectexplorer.v1.image
com.palantir.api.objectexplorer.v1.menu
com.palantir.api.objectexplorer.v1.model
com.palantir.api.objectexplorer.v1.vis
```

Connection, Clients, and Context

The connection and context APIs reside in t he `com.palantir.api.workspace` package. You use these methods to create a client-level connection to the Workspace. From a connection, you can access the Workspace's applications and their associated helpers. The key APIs this package include:

Table 9 Key Connection and Context APIs

API	Description
`AclCreationConnection`	Defines the methods for classes that provide a connection to the Palantir Platform servers for the purpose of creating ACLs.
`PalantirClient`	Represents the client itself, which allows for logging in, accessing the context, and logging out. Only needed for headless applications that run outside of the Workspace.
`PalantirClientContext`	Provides accessor methods for common client tasks, including getting a connection (`PalantirConnection`), search (`SearchFactory`), investigation creation and loading, object publishing, and object resolution.

Table 9 Key Connection and Context APIs (continued)

API	Description
`PalantirConnection`	Represents the connection to the Palantir Dispatch Server and the database. Contains methods to search for, load, and store data. Also contains methods for managing job status and application state.
`PalantirContext`	Starting point for applications and helpers accessing Palantir internals. This interface represents a Palantir client that includes the graphical user interface. Use this class to access specific investigations, add listeners, and get user preferences. Inherits from the `PalantirClientContext`.
`PalantirInvestigation`	Represents an investigation in the Workspace. Use this API to obtain an investigation's `Locator`.

Applications and Helpers

The `com.palantir.api.workspace` packages contain APIs that you can use to work with Workspace applications and their helpers. The key interfaces include the following:

Table 10 Key Application and Helper APIs

API	Description
`ApplicationFactory`	Provides a factory for instantiating an application within the Palantir Platform.
`ApplicationContext`	Represents the starting context for an application within the Palantir Platform. All helpers are created relative to a specific `ApplicationContext`.
`ApplicationInterface`	Defines the basic interface for Palantir applications. All applications in Palantir implement this interface. Use this interface to obtain an application's interaction handlers and listener.
`HelperFactory`	Provides a factory for instantiating a helper within the Palantir Platform (Histogram, Timeline, etc.). Implement this class to plugin a custom helper.
`HelperInterface`	Defines the basic functions that all helpers implement. Registers and unregisters listeners. The helper might be recycled if the user keeps the workspace open but closes the helper.
`Launcher`	Provides a simple interface for applications and helpers to invoke tasks, jobs, wizards, and dialogs in the Palantir Platform.

You will also find the following `com.palantir.api.workspace` subpackages helpful:

Table 11 com.palantir.api.workspace Subpackages for Applications and Helpers

Subpackages	Description
`applications`	Interfaces to each specific application: Browser, Graph, Map, and so forth.
`graph`	Interfaces for Graph events, traversing nodes, working with edges, and additional utility methods. You also will find interfaces for adding listeners to the graph.
`map`	Interfaces for working adding listeners to the Map and working with its events.

The following packages contain APIs you can use to customize the Object Explorer application:

- The `com.palantir.api.workspace.applications` package (listed under Table 11) includes an interface to the Object Explorer application. Object Explorer helpers and other applications that interact with Object Explorer implement `ObjectExplorerApplcationInterface` to acccess the `UserInterfaceManager` in `com.palantir.api.objectexplorer.v1` and the `AnalysisModel` in `com.palantir.api.objectexplorer.v1.model`.
- The `com.palantir.api.objectexplorer.v1` package contains interfaces to perform analysis operations, and create and load sets into Object Explorer.
- The `com.palantir.api.objectexplorer.v1.dnd` package contains an enumeration of data transfer types (data set, operation, and object ids) used for data transfer both within Object Explorer and to/from other applications.
- The `com.palantir.api.objectexplorer.v1.image` package contains interfaces used to store, register, and retrieve images for Object Explorer plugins.
- The `com.palantir.api.objectexplorer.v1.menu package` contains interfaces to add menu items to the Object Explorer user interface, including visualizations, analysis menu, and drill-down context menus.
- The `com.palantir.api.objectexplorer.v1.model` package contains interfaces used by Object Explorer components to obtain details of the analysis models and domain (ontology) model.
- The `com.palantir.api.objectexplorer.v1.vis` package contains interfaces to create and customize Object Explorer visualizations and interact with the analysis model.

You can use these APIs to:

- Add your own implementation of an existing visualization (histogram, chart, timeline).
- Build your own data visualization.
- Add an operation (for example, as a drill-down option on a view).
- Add menu items and buttons to launch custom data views (for example, for new drill-down operations, options, or filters).

The Object Explorer application leverages Horizon object APIs and server connections for scalability and performance. Use the Object Explorer APIs in conjunction with the Horizon APIs for customizations that filter objects, perform calculations on object sets, or retrieve information from objects and object properties. (See also, Horizon Objects on page 64.)

Selection, Search, and Filter

The `com.palantir.api.workspace` package contains the `selection`, `search`, and `filters` subpackges to assist you in the construction of helpers and applications.

Table 12 Key Selection, Search, and Filter APIs

API	Description
`SelectionAgent`	Provides support for selection in a helper.
`SelectionAgentEvent`	Maintains the state related to a selection event. Use this class to get filter information associated with a selection event.
`SelectionAgentListener`	Implement this interface to receive selection events from a `SelectionAgent`.
`SelectionAgentSupport`	Handles item filtering tasks for applications and helpers that support visual filtering of their contents. This class contains methods for adding and removing listeners.
`IFilterCriterionFactory`	Generates new instance of filter criteria, modifies existing filters, or populates new filters. Use the `IFilterManager.getFilterCriterionFactory()` to obtain an instance.
`IFilterManager`	Tracks the filters in the current investigation.
`FilterListener`	Represents listener for filter operations.
`FilterEvent`	Represents a filter operation.

The `com.palantir.api.workspace.dataitem` package contains the interfaces for working with objects returned from filters or selection.

Table 13 APIs for Data Returned from Filters and Selections

API	Description
`DataItemModel`	Represents an individual item associated with a group. Typically this class represents a `PTObjectContainer` but can represent other items as well, such as an edge.
`DataItemGroup`	Represents a resizable group of `DataItemModel` objects.
`DataItemModelType`	Enumerates the possible types that can be modeled.

Data Services APIs

Data sources represent the source for information that is in Palantir. When working with Palantir objects, you need to understand and manage data sources. You might, as you customize and extend Palantir, encounter situations where you need to bring data sources from outside parts of your organization into Palantir. Data services provide APIs for working with data sources and the data integration APIs provide a means for bringing data into Palantir.

Data Sources

The Palantir API includes a number of APIs for working with data sources. The key data source APIs are located in the `com.palantir.api.dataintegration` package.

Table 14 Key Data Source APIs

API	Description
`IDataSource`	Represents the source for information in the Palantir Platform. For example, a database.
`IDataSourceMaker`	Provides an extensibility point for the integration of custom validation and parsing code for the generation of user-created data sources.
`DataSource`	Provides the methods for a data source for information in the system such as a property or link. Do not implement this interface. Instead use it to interact with `IDataSource` instances you create or obtain through other means.

A major API for data sources, the `DataSourceRecord` is found in the `com.palantir.services.ptobject` package. This API indicates where in a data source a particular information is located. You cannot reuse instances of this object.

Data Integration

The data integration APIs supply the means for integrating data from any external source into the Palantir Platform. The following packages contain APIs for importing data into the Palantir Platform:

- `com.palantir.api.dataintegration`
- `com.palantir.api.dataintegration.crawl`
- `com.palantir.api.dataintegration.detect`
- `com.palantir.api.dataintegration.extract`
- `com.palantir.api.dataintegration.transform`
- `com.palantir.api.dataintegration.util`

For a detailed list of these APIs and information on how to use them, see the *Data Integration Guide*.

Object Model APIs

Palantir allows customers to model a Dynamic Ontology that is specific their own needs. The object model API provides a programmer with the building blocks to interact with objects returned from custom code. These building blocks include objects, properties, and links. The following sets of API have direct bearing on common object model interactions:

- Objects
- Properties, Links, and Other Object Components
- Services
- Horizon Objects

Objects

The `com.palantir.services.ptobject` package is the central package for interacting with Palantir objects.

Table 15 Key Object APIs

API	Description
`IPTObject`	Interface to a Palantir object. An object is a container for an object's components: properties, notes, media, and links. `PTObjectContainer` is the preferred method for working with objects.
`PTObjectContainer`	Wraps `PTObject` instances and provides mechanisms for accessing the information in a Palantir object. Use this class to manage an object and its relationships.
`PTObjectType`	Represents the type of Palantir object as it is defined in the dynamic ontology.
`PTObjectLinker`	Manages relationships between objects.
`PTObjectTypeUtilities`	Provides utilities for working with object types.

Properties, Links, and Other Object Components

Objects typically have property and link components. Properties are the object's attributes such as a phone number. Links are the connections between objects that represent any connection. Such as a parent to child relationship between two people. Objects can also have other elements that are best thought of as specialized properties — media and notes. The are several API packages you have access to for manipulating these components, including:

```
com.palantir.services.impl.property
com.palantir.services.impl.property.approx
com.palantir.services.impl.property.config
com.palantir.services.impl.property.enumeration
com.palantir.services.impl.property.formatter
com.palantir.services.impl.property.formatter.modifier
com.palantir.services.impl.property.invalid
com.palantir.services.impl.property.maker
com.palantir.services.impl.property.validator
com.palantir.services.ptobject
```

Table 16 Key Component APIs

API	Description
`Property`	Represents a single attribute or piece of information about an object. This class contains specialized methods for creating and managing properties. Properties are all backed by a data source and tied to the source by a `DataSourceRecord`. The `Locator` class links a `Property` to its location in a data source record.
`PropertyType`	Represents a property's dynamic ontology type. Provides access to the information, approxes, makers, and so forth defined in the ontology.
`Role`	Defines the direction of a tie between and object and any information.
`Link`	Represents a connection between two objects.
`LinkType`	Provides the type used to compare and identify Palantir links.
`Media`	Represents arbitrary binary data, such as an image, video, or audio file associated with an object.

Property classes have a number of APIs that allow for specialized customization of properties with the Palantir Platform. The `com.palantir.services.property` package contains these property customization APIs.

Table 17 Specialized Property Customization APIs

API	Description
`IPropertyValidator`	Creates custom validators to process the data in a property in a specific way.

Table 17 Specialized Property Customization APIs (continued)

API	Description
`IPropertyFormatter`	Constructs a human-readable string from the data in a property.
`IPropertyMaker`	Extracts the property information from a string input.
`IPropertyApproxGenerator`	Normalizes data in a property data map to enable fuzzy-match searches.

Services

Services interfaces and classes reside in the `com.palantir.services` package and provide services for base objects in the Palantir Platform.

Table 18 Key Services APIs

API	Description
`Locator`	Represents the location of a data item in the data repository.
`LocatorFactory`	Defines methods for factories that can create `Locator` in the data repository. The `PalantirConnection` extends this interface.
`BasicPropertyUtils`	Provides basic utilities for working with `Property` objects. For example, this class contains a method to pre-test the creation of a property from a value.
`Realm`	Represents a repository for data items. Realm refers to either a specific investigation or the base realm.
`User`	Represents a user.
`UserSessionToken`	Supplies a token representing a long-lived session.
`AbstractDBObject`	Supplies the default implementation for a base persistence object. Contains methods to get basic information such as an object's ID or `Locator`.
`PTTimeInterval`	Represents a time interval.
`AbstractPTType`	Provides comparison services for comparing and identifying Palantir types.

Horizon Objects

The Horizon API provides access to an alternate object model that is independent of `PTObject` and its components. Horizon packages provide what are, in effect, light-weight renditions of Palantir objects (`PTObject`) and associated components. For example, the Horizon `HObject` provides a parallel to `PTObject`, and there are Horizon specific links, link types, properties, property types, server connections, and so forth. The Horizon server provides search support for these objects, enabling quick response times for some searches.

Horizon allows the platform to scale in scenarios that require high performance analysis of very large data sets. The scalability and performance gains are achieved by providing light-weight views of objects that support, for example, extracting values from properties without loading the full object into memory. Notes, media, data sources, and other components are not needed in this context, and ignoring them speeds up query processing.

The Object Explorer application leverages Horizon objects and servers to assemble and analyze data sets consisting of millions or billions of objects simultaneously. Developers can use the Horizon API to customize data views and add options for Object Explorer, and other applications. For example, a light-weight selection helper built on the Horizon API can retrieve only object properties needed for a given table or other data visualization. By skipping the PTObject model, which would require loading the full object in memory, a light-weight helper works nicely for applications like Object Explorer where scalability is a priority.

The Horizon Object Model

The following packages contain APIs for the Horizon object model.

```
com.palantir.api.horizon.v1
com.palantir.api.horizon.v1.common
com.palantir.api.horizon.v1.lang
com.palantir.api.horizon.v1.lang.function
com.palantir.api.horizon.v1.object
com.palantir.api.horizon.v1.object.approx
com.palantir.api.horizon.v1.object.extractor
com.palantir.api.horizon.v1.object.paging
com.palantir.api.horizon.v1.operation
com.palantir.api.horizon.v1.operation.algebra
com.palantir.api.horizon.v1.operation.match
com.palantir.api.horizon.v1.persistence
com.palantir.api.horizon.v1.view
```

The connection and factory APIs reside in the `com.palantir.api.horizon` package. You use these methods to create a client-level connection to the Workspace. The key APIs in this package include:

Table 19 Methods for Creating Horizon Connections

API	Description
`HorizonConnection`	Represents the connection to the Palantir Horizon Server and the database. Contains methods to search for, load, and store data, compute results of view calculations and operations, and get information about objects and object properties for data views.

Table 19 Methods for Creating Horizon Connections (continued)

API	Description
`HorizonConnectionFactory`	Defines a method for factories that can create Horizon server connections.

The `com.palantir.api.horizon.v1.object` is the central package for interacting with Horizon objects, properties, and links. Key APIs in this package include:

Table 20 Key Horizon Object APIs

API	Description
`HObject`	Defines the interface for Horizon objects.
`HLink`	Represents a Horizon link (connection between two objects).
`HProperty<V>`	Represents a Horizon property.

The package `com.palantir.api.horizon.v1.view` contains interfaces for implementing custom views as Horizon plugins to show the result of a given calculation across a set of objects.

Extensibility Points

Developers can create custom operators, functions, and data by leveraging the following Horizon APIs.

Table 21 Horizon APIs for Creating Plugins and Custom Data Views

API	Description
`Operator<T extends OperationArgument>`	Object representing a Horizon operator, which is applied to sets of objects to produce an `Operation`. Plugins implementing this interface must also provide an implementation of `OperatorFactory` to specify the URI and class of the `Operator`. In general, this object is not referenced directly in client code; rather an `Operator` is specified by its corresponding `OperatorUri`. Package: `com.palantir.api.horizon.v1.operation`
`OperatorFactory<A extends OperationArgument>`	Interface for specifying the URIs and classes of an Operator. Package: `com.palantir.api.horizon.v1.operation`
`OperationArgument`	Object representing the argument, or parameter, to an `Operation`. All implementations of this interface MUST contain a public default (no argument) constructor. Package: `com.palantir.api.horizon.v1.operation`
`Function`	Package: `com.palantir.api.horizon.v1.lang`

Table 21 Horizon APIs for Creating Plugins and Custom Data Views (continued)

API	Description
`FunctionFactory`	Package: `com.palantir.api.horizon.v1.lang`
`View`	Together with `ViewResult`, provides an interface for creating custom views of data (results of calculations). Package: `com.palantir.api.horizon.v1.view`
`ViewFactory`	Together with `View`, provides an interface for creating custom views of data (results of calculations). Package: `com.palantir.api.horizon.v1.view`

Core Feature Services

You can use the search, security, and job APIs to interact with these key services within Palantir.

Search

Most search related interfaces reside in the `com.palantir.services.search` package and include the following key classes and interfaces:

Table 22 Key Search APIs

API	Description
intermediary template APIs	Represent a potential commonality between two Palantir objects. For example, a template could represent the possibility that two people attended the same event. These classes and interfaces exist for objects and their components. For example, there is an `IDocumentIntermediaryTemplate` and an `ILinkIntermediaryTemplate` interface.
intermediary instance APIs	Represent the commonality between two Palantir objects. For example, both Sally and John attended the birthday party. A set of classes and interfaces exist for objects and their components. These classes and interfaces exist for objects and their components. For example, there is an `IDocumentIntermediaryInstance` and an `ILinkIntermediaryInstance` interface.

Table 22 Key Search APIs (continued)

API	Description
`GeoQuery`	Acts as the base class for all types of geographic information system queries.
`GeoSearchFilter`	Restricts the results of a GeoSearch.
`GeoSearchResults`	Contains the results of a GeoSearch.
`ISearchQuery`	Represents an object query. This interface belongs to the `com.palantir.services.impl.search` package.
`ISearchFactory`	Creates new search queries. You obtain an instance using the `PalantirClientContext.getSearchFactory()` method.
`ResultsPage`	Contains the actual search results.
`SearchResultsPager`	Allows easy access to search results. Implements `ResultsPage` but does not itself contain results.
`GeoSearchResultsPager`	Allows easy access to search results. Implements `ResultsPage` but does not itself contain results.
`SearchAroundQuery`	Container class for `SearchAroundTemplate` objects from JAXB.

The `GraphApplicationInterface.performLinkBy()` method does a link by search given a collection of nodes and a `SearchAroundQuery` instance. The `PalantirConnection.searchAround()` method searches over the current connection using a specified `SearchAroundQuery`.

Security

The key security APIs for working with user permissions and access control are found in the `com.palantir.commons.security` package and include the following:

Table 23 Key Security APIs

API	Description
`AccessControlItem`	Represents a set of group permissions associated with a particular component.
`AccessControlList`	Represents a set of `AccessControlItem` objects for a single component; commonly called an ACL.
`SecurityContext`	Allows you to register or unregister security providers.
`Permissions`	Represents the set of permissions that a group has for any piece of information.

Table 23 Key Security APIs (continued)

API	Description
`UserPermissions`	Represents the permissions granted to a user by virtue of the user's group membership.

The following additional security APIs are also available.

Table 24 Additional Security APIs

API	Description
`AclCreationConnection`	Defines the methods for classes that provide a connection to Palantir Platform servers for the purpose of creating ACLs. The `com.palantir.api.client` package contains this class.
`InstanceAcl`	Represents the connection between a `Property` and its associated data source records. This is in the `com.palantir.services.ptobject` package.
`SecureComponent`	Describes the components in the system. All object component classes are subclasses of `SecureComponent`. This is part of the `com.palantir.services.ptobject` package.

Jobs

The `com.palantir.api.job` package contains packages you can use to interact with jobs. For example, you can use these APIs to see submit jobs, get a status on a current job, or learn how long a job took to run.

Table 25 Key Job APIs

API	Description
`Job`	Represents a scheduled task in the Workspace.
`JobArgs`	Represents the parameters passed to the job. Each job type might have its own type of `JobArgs` object.
`JobStatus`	Holds the current state of a job.
`JobResults`	Contains the results of a job execution.
`JobSpec`	Specifies the job itself. For example, you use this API to specify whether a job runs immediately or if it is restartable.
`JobStatusMonitor`	Contains a periodically updated `JobStatus` instance.

Geographic Information

The Palantir Map application is a sophisticated and highly customizable geographic application. By default, the application ships without a default tile set. You use the API to configure your own tile sets.

```
com.palantir.gis
com.palantir.gis.api.export
com.palantir.gis.api.geom
com.palantir.gis.api.geom.impl
com.palantir.gis.api.model
com.palantir.gis.api.model.impl
com.palantir.gis.gazetteer
com.palantir.gis.importer
com.palantir.gis.integration
com.palantir.gis.kml
com.palantir.gis.kml.export
com.palantir.gis.proj
com.palantir.gis.shp
com.palantir.gis.shp.export
com.palantir.gis.tiles
com.palantir.gis.utils
```

Basic Map Objects

Key map APIs are located in the `com.palantir.api.workspace.map` package.

Table 26 Key Tile Source APIs

API	Description
`Gazetteer`	Represents the Gazetteer plugin.
`IMapModel`	Represents the Map application.
`MapLayersInterface`	Methods for working with nodes on the Map.
`MapModelListener`	Provides methods for responding to events from the `MapModel`.

The basic Map application objects are the geographic points on the Map itself. These points are made up of coordinates that are themselves objects. A gazetteer is an object that takes strings and returns geographic coordinates. Palantir provides a single `Gazetteer` interface and some implementations.

Table 27 Key Map APIs

API	Description
`GeoCoordinate`	Specifies a coordinate with latitude and longitude.
`GeoMath`	Static functions related to doing geographic math.
`PTGeometry`	Interface marking the GIS objects supported by the revisioning database.

Table 27 Key Map APIs (continued)

API	Description
`PTPoint`	Implements a point on the map.
`PTPointParser`	Parses geographic points.
`SimpleGeoCoordinate`	Provides a simple implementation of `GeoCoordinate`.

Utilities and Plugins

The `com.palantir.gis.integration` package contains interfaces and classes for working with GIS plugins.

Table 28 Key GIS Utilities and Plugin APIs

API	Description
`GISPluginManager`	Adds and removes GIS plugins. You can use this API to get a list of the configured plugins.
`GISFeature`	Represents a GIS exportable item.
`GISExportPlugin`	Manages export plugin initialization, opening export streams, and plugin metadata.

Web REST APIs

Palantir Web is an application that allows your users to access the features of Palantir through a web browser. Palantir provides a representational state transfer (REST) API that you can use to access features of this browser. For more information on using these APIs, contact your organization's Forward Deployed Engineer.

6 Working with Palantir Object APIs

This chapter discusses working with objects, their components, and data sources. The following topics are covered:

- Understanding Object Concepts
- Developing Programs with the Object Model APIs
- Example of Object APIs

Understanding Object Concepts

This section discusses basic concepts, terms, and relationships you need to know when working with objects. The following topics are discussed:

- Dynamic Ontologies and Object Models
- Data Repository and Object Storage
- Data Lineage with Sourcing

Dynamic Ontologies and Object Models

Within Palantir, a Dynamic Ontology describes the semantics of a social organization or business domain. For example, consider a bank. A bank has tellers, branches, money, CDAs and many other people and things. During the course of doing business, events such as deposits, transfers, and robberies might occur.

Further, an event like a robbery might have associated footage of the robbery from the security camera or a recording of the police negotiator. Additionally, there are relationships inherent in the bank business such as a teller works for a bank manager. There are also relationships between people who participate in events at the bank such as a security guard is the brother of a bank robber.

Regardless of whether a Dynamic Ontology describes a bank, a networking company, or any other social or business structure, Palantir models that ontology using the same objects. This means a programmer need only be familiar with a single set of object APIs. Moreover, once you know these object APIs, you can apply your knowledge between different business domains with distinctly different ontologies.

An object in the object model is a container for multiple component parts. An object's components describe that object or provide associated data. Palantir categorizes components as:

properties	Text attributes on an object such as a person's name or a bank's address.
media	Images, video, documents or other binary formats associated with an object. For example, the security footage associated with an event is a media component.
notes	Free text about an object. For example, an analyst might note "This robbery appears to be an inside job" on an event object.
links	Relationships between objects. For example, the security officer is the brother of the bank robber or the teller is an employee of the bank. Typically, one side of a link is a parent object and the other side is a child. The parent and child sides are defined by the relationship type. When the relationship has equality such as "works with" relationship, the system arbitrarily chooses a parent and a child.

Just as with the objects themselves, components in the object model are independent of the semantics in the Dynamic Ontology. A bank might track a property called routing number while a law enforcement organization might track a driver's license. The underlying object model relies on the same property API regardless.

Data Repository and Object Storage

When an object or property is brought into Palantir it stores the object in the underlying Data Repository. Unlike other analysis software, Palantir does not create a schema to represent the Dynamic Ontology. Instead, the Palantir object model is an abstraction that sits between the ontology (the user view) and a system's physical storage (Data Repository).

This means the underlying data schema is independent of the Dynamic Ontology and echoes instead the simplified data model. The following lists the major 6 tables in this simple schema:

- `PT_Object`
- `PT_Property`
- `PT_Node`
- `PT_Media`
- `PT_Object_Object`

Using these five tables, it is possible to access the content of each object and its associated components. Moreover, as with the object model, the database schema is fixed, meaning that it is used across every deployment and organization.

Data Lineage with Sourcing

Every piece of information in Palantir originates with a data source. A common adage in programming is garbage in garbage out or information is only as good as its source. Data sources provide the lineage or pedigree for the components in the Data Repository. Objects are sourced through their components; they do not themselves have data sources.

Data sources can be anything such as a newspaper, an email, a web page, or television news program. A fairly common data source is manually created data. This is data an analyst enters into the Workspace directly, for example; adding a tattoo property to a bank robber entity.

An object component in Palantir must have at least one, and might have more, data sources. Palantir not only tracks which data source originated an object's component but it also tracks the location of the source in the data. For example, an email between two brothers might mention a bank, the bank manager's name, and of course it would contain the senders email address and the email address of the recipient. This single data source, an email, has locations that source multiple properties.

Developing Programs with the Object Model APIs

This section takes a closer look at the object model APIs and what you can do with them. The following topics appear here:

- Creating Objects and Obtaining a Locator
- Creating Components (Properties, Notes, Media, and Links)
- Referencing Data Sources
- Loading Objects into an Investigation

Creating Objects and Obtaining a Locator

The `com.palantir.services.ptobject.PTObject` interface represents a Palantir object. In the underlying Data Repository, each Palantir object is in the `PT_OBJECT` table. You should never work directly with `PTObject` instance. Instead, create and work with objects using the `com.palantir.services.ptobject.PTObjectContainerFactory` class. Key methods in this class include:

API	Description
`createBlankObject`	Creates objects that have no components. This overloaded method requires a `Locator`.
`createPTObjectContainer()`	Creates a container for an object and its components.

Once an object exists, you manage it and its components through the `PTObjectContainer` interface. (Experienced Palantir integrators often refer to these objects as "ptocs.")

The `PTObjectContainer` allows you to obtain an object's properties, media, notes, and relationships (links). However, only `PTObjectContainer` allows you to quickly retrieve all of the related child or parent objects linked to the object. Moreover, these related objects themselves return as `PTObjectContainer` instances that you can immediately begin to use. The following table lists some key methods on the `PTObjectContainer`:

API	Description

`addProperty()`
`addMedia()`
`addNote()`
	Adds a single `Property`, `Media`, or `Note`. These methods all return a `ChangesMade` instance that marks whether the object was changed as a result of the operation.
	There are corresponding remove methods.

`addChildObjectContainer()` `addParentObjectContiner()`	Adds a child or parent object respectively with the specified `Role`. These methods return a `Link` object. There are corresponding remove methods.
`getAllObjects()`	Returns all related objects.
`getAllChildren()` `getAllParents()`	Returns all children or parent objects respectively.
`getLocator()` `setLocator()`	Gets or sets the `Locator` object that locates this data item in the data source.
`getID()`	Returns the ID of this object in the database.

`getChildObjectLinks()`
`getNotes()`
`getMedia()`
`getProperties()`
`getParentObjectLinks()`
	Returns collections of the corresponding component type. These methods are notable in that you provide an `ObjectComponentFilter.PTObjectFilter` to restrict the results returned.

Before you can create an object or its components programmatically, you have to have a `com.palantir.util.LocatorFactory`. The `PalantirConnection` in the `com.palantir.api.workspace` package implements `LocatorFactory` and should act as your `LocatorFactory`. For headless clients, you can obtain `PalantirConnection` by calling `PalantirClientContext.getPalantirConnection()`. If you are developing a custom helper or an approx, you would use get the connection from the `PalantirContext` which extends the `PalantirClientContext`.

Creating Components (Properties, Notes, Media, and Links)

In the previous section, you saw that the `PTObjectContainer` contains several methods of managing object components. All component objects inherit from a single abstract class, `com.palantir.services.ptobject.SecureComponent`. When creating a `Note` or a `Media` you should use the corresponding `createValue()` method inherited from `SecureComponent`.

`Link` and `Property` objects require special processing. A `Link` represents a relationship between two objects. To create a `Link` instance you can use the methods on the `PTObjectLinker` for adding links `addLink()` and `addChildParentLink`.

Properties are slightly more complex. Recall that a `Property` is a text attribute such as a phone number, a date of birth, or an address. Simple properties are of type `Number`, `String`, `Date`, or `Enumeration`. For example, an age property is a `Number` type and eye color is a `String`. A date of birth for example is defined with a `Date` base type.

Other properties are more complex and are a composite of several component strings, such as an address. You can determine which category a property falls into, simple or composite, by calling:

```
...
property.getType().getBaseType() == PropertyType.PROP_NUMBER_BASE_TYPE
...
```

As text strings, properties often have various restrictions on their format, for example, an address might require a zip code, a phone number typically requires a certain number of digits. When you want to create a property on an object, call `Property.attemptToCreate()`. This method attempts to create a property in the proper format and throws a `PropertyMakerException` if it cannot.

Referencing Data Sources

Each property has one or more associated `DataSourceRecord` (often called DSR) objects. Each `DataSourceRecord` has an associated access control list represented by a `InstanceAcl` object. This object is associated with a property through its DSRs. A property, as a secure component, has a collection of `InstanceACL` objects associated with it. The collection will include at least ACL object and might have many as more sources.

Regardless of whether you are working with structured or unstructured data sources, you cannot apply the same `DataSourceRecord` to two components. Attempting to reuse a DSR causes data problems. If you are running with assertions enabled, as recommended, attempting to reuse a DSR will result in an assertion error. Instead, you must create a `DataSourceRecord` either by creating one from scratch or copying an existing one.

The `com.palantir.api.ptobject.DsrFactory` interface contains the methods you should use for generating a DSR. You obtain an instance of this interface from `PalantirConnection.getDsrFactory()`. The factory class contains several methods that you can use to generate a DSR. These methods generally take some value type that points to a location in a data source:

API	Description
`copyDsr()`	Copies an existing DSR.
`createDsr()`	Creates a new DSR.
`createManuallyEnteredDsr()`	Creates a manually entered DSR using the information for the current investigation.
`createSplDsr()`	Creates a DSR given a string position locator (SPL).

The `SecureComponent` class contains several methods for adding data source records to objects:

API	Description
`addDataSourceRecord()`	Adds a single data source record.
`addDataSourceRecords()`	Adds a collection of data source records.
`deepCopy`	Creates a data source record by copying another.

Note: There are not any methods for adding a data source to an object. This is because an object is sourced through its components. Think of an object's sourcing as the cumulative data sources of that object's components. Objects do have a single property, a title, which is used as a display name. This title takes a data source.

Loading Objects into an Investigation

After constructing an object, you must store it into an investigation and in the repository. Objects can have information such as properties, links, and large media attachments that are not needed for every operation. Additionally, an operation might or might not require data source records. Load levels control the granularity of an object load.

Once an object is in an investigation, you can load it and change it. After changing an object, you must store the changes into the investigation. Until you store your object changes, they are only in memory and do not exist in an investigation's repository. If you want the changes in the investigation visible in the base realm and available to other investigations, you must publish them.

Getting Load Levels

Passing a `LoadLevel` instance determines how much of an object's information the system will load. You get an instance using the static instance methods on the `LoadLevelFactory` class. This class also contains a number of other utility methods.

LoadLevel Method	Use when you need
`getBaseLoadedInstance()`	None of the information in the database.
`getFullDsrLoadedInstance()`	All of the object's information including the data source records.
`getFullyExceptDSRLoadedInstance()`	Everything but the data source information.
`getIntrinsicPropertyLoadedInstance()`	Only a title or intrinsic date or location if it exists.
`getLightlyLoadedInstance()`	Properties and notes.
`getLinkTitleLoadedInstance()`	The object's title and anything the object is linked to.

LoadLevel Method	Use when you need
`getPropertyDsrLinkTitleLoadedInstance()`	The object's title, links, properties, and data sources.
`getLinkDsrLoadedInstance()`	Links with the data sources.
`getMediaDsrLoadedInstance()`	Media and the data sources.
`getNoteDsrLoadedInstance()`	Notes and the data sources.
`getPropertyDsrLoadedInstance()`	Properties and the data sources.
`getPropertyLoadedInstance()`	The object's properties only.
`getSearchLoadedInstance()`	The properties, media, and notes and their associated data source records.
`getSearchLoadedInstanceWithDsrs()`	The properties, media, and notes.

Loading, Storing, and Publishing an Object

You use the `PalantirConnection.objectLoad` methods to load an object. Objects are loaded into the investigation you define when making a connection. The `PalantirClientContext` contains the following methods for establishing a connection and setting the current investigation:

PalantirClientContext Method	Description
`loadInvestigationByRealmId()`	Loads an existing investigation.
`createInvestigation()`	Creates a new investigation.
`getPalantirConnection`	Provides methods for getting a connection. You can get a connection to the current investigative realm or a particular realm.

`PalantirConnection` contains the many methods for working with objects. The load methods take a `LoadLevel` instance.

PalantirConnection Method	Description
`getRealm()`	Gets the current investigative realm associated with the connection.
`loadDataSource()`	Loads a copy of the data source tree to a specific depth. You can use `loadDataSources()` to load a collection of data sources.
`objectStore`	Stores objects and their associated data to the base realm and the repository.
`objectLoad`	Loads objects into the current investigation's memory.

PalantirConnection Method	Description
`publishObjects()`	Publishes all new changes concerning the given collection in the investigative realm to the base realm.
`resolveObjects`	Resolves the given collection of objects. Resolving takes two or more objects, combines their properties, and consolidates them into a single object that retains all of the attributes of the original objects.

Once object is in an investigation (either as a brand new object or a changed object), you call the `PalantirConnection.publishObjects()` to publish it to the base realm where it becomes available to other users. Objects in the base realm are available to all investigations. You supply a `PalantirDataEventType` to the publish method.

Use `ObjectChangesConnection.getObjectsInRealm()` to get all or some of the objects in a given realm. This method takes a `realmID` and an `objectId` and returns all the objects with an ID greater than the given `objectId`. To get all the objects in the realm, you pass in `null` for the `objectId`.

Example of Object APIs

This example illustrates the following features:

- creating a headless client context
- connecting to the Dispatch Server
- creating an object
- adding properties to the object
- creating a data source
- searching for objects
- changing an existing property

Be sure to run this sample code with assertions enabled. For detailed information about the APIs in this example, refer to the Palantir Javadoc.

The Example Code

```
package com.palantir.headless.simpleobjecttest;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.HashSet;
import com.palantir.client.search.SearchOperator;
import com.palantir.client.search.PalantirException;
import com.palantir.services.ptobject.DataSourceRecord;
import com.palantir.exception.PalantirException;
import com.palantir.services.Locator;
```

```
import com.palantir.services.OperatorType;
import com.palantir.services.impl.search.ISearchQuery;
import com.palantir.services.impl.property.PropertyTypeComponent;
import com.palantir.services.loadlevel.LoadLevelFactory;
import com.palantir.services.ptobject.PTObjectContainer;
import com.palantir.services.ptobject.PTObjectType;
import com.palantir.services.ptobject.Property;
import com.palantir.services.ptobject.PropertyType;
import com.palantir.services.ptobject.Role;
import com.palantir.services.search.SearchResultsPager;
import com.palantir.services.ptobject.PTObjectContainerFactory;
import com.palantir.util.BasicLocatorUtils;
import com.palantir.util.paging.ResultsPage;
import com.palantir.api.workspace.PalantirClient;
import com.palantir.api.workspace.PalantirClientContext;
import com.palantir.api.workspace.PalantirClients;
import com.palantir.api.workspace.PalantirConnection;
import com.palantir.api.workspace.PalantirInvestigation;
import com.palantir.api.dataevent.PalantirDataEventType;
import com.palantir.api.dataintegration.BasicDataSourceRecordUtils;
public class SimpleObjectCreate {

   public static void main(String[] args) {
     try {
       String username = "admin";
       String password = "palantir";
/* This example uses a headless client as opposed to a Workspace client. A
* headless client is useful for developing command-line interfaces while a
* Workspace client is the standard for plugin projects.
*/
       // Instantiate the client
       PalantirClient client = PalantirClients.getHeadlessClient();
       //Log in to the Dispatch Server
       client.login(username, password.toCharArray());
       // Create the client's context
       PalantirClientContext clientContext = client.getClientContext();
/* Create an investigation and establish a connection to it. When you create
* an investigation, you supply only a name and a description. Palantir
* automatically generates a realm ID for this investigation. You can use
* the ID later to connect to and modify the data in the investigation.
*/
       PalantirInvestigation inv = clientContext.createInvestigation("Object
         Create Sample Program", "Description");
       PalantirConnection conn = clientContext.getPalantirConnection(inv);
/* Before creating a new object, you create a DSR to source the object's
* title. This title is used as the display name for the object in
* the user interface. Generating a data source record requires you to
* provide a Locator. In this case, because you are specifying the
* connection as the Locator, the data source appears as manually
* entered when the property is viewed in the Workspace.
*/
DataSourceRecord titleDSR = conn.getDsrFactory().createManuallyEnteredDsr();
Collection<DataSourceRecord> collDSRs = new HashSet<DataSourceRecord>();
       collDSRs.add(titleDSR);
/* In the next section you create a blank object. Creating an object
* requires you to specify a PTObjectType and a locator factory — the
```

```
* connection itself. For a PTObjectType you can specify any of the types
* your ontology. Each object type will have a unique URI. You can browse
* your ontology from the Dynamic Ontology Manager user interface.
*/
PTObjectContainer newObj = PTObjectContainerFactory.
     createBlankObject("Julie Palantirian",
     PTObjectType.getByUri("com.palantir.object.person"),
     conn, collDSRs);
/* The next few lines illustrate how to create a simple property and then a
* complex property. You cannot use a data source record multiple times. Each
* property must be uniquely sourced. You can create a DSR by deep-copying
* another as illustrated in the code below:
*/
//create a simple property and give it a value
Property propHair = null;
propHair = Property.attemptToCreate(conn,
           PropertyType.getByUri("com.palantir.property.HairColor"),
           "dark brown",Role.NONE);

//Create a DataSourceRecord and use it to source both the simple and the
//complex property
propHair.addDataSourceRecord(titleDSR.deepCopy(true))

// Create a complex property's components and then the property itself
Map<PropertyTypeComponent, String> componentValueMap =
             new HashMap<PropertyTypeComponent, String>();

componentValueMap.put(PropertyType.NAME.getComponentByUri("FIRST_NAME"),
               "Julie");

componentValueMap.put(PropertyType.NAME.getComponentByUri("LAST_NAME"),
               "Palantirian");

Property propName = Property.attemptToCreate(conn,
                  PropertyType.getByUri("com.palantir.property.Name"),
                  componentValueMap, Role.NONE);

// Create a DSR by copying the first
propName.addDataSourceRecord(dsrHair.deepCopy(true));

//Add both properties to the object
newObj.addProperty(propHair);
newObj.addProperty(propName);
/* Print out all the information about the newly created object and then add
* the object to a collection.
*/
Collection<Property> theProps = newObj.getProperties();
           System.out.println("the properties of my new object\n");
   for (Property prop : theProps) {
       System.out.println(prop.toTypeValueString());
      }

       Collection<PTObjectContainer> newPtocs = new
ArrayList<PTObjectContainer>();
       newPtocs.add(newObj);
/* The next two lines of code store the object into the investigation.
* The next call publishes the object in the base realm.
```

```
*/
Collection<PTObjectContainer> newPtocs = new ArrayList<PTObjectContainer>();
newPtocs.add(newObj);
// Store the object to the current investigative realm
conn.objectStore(newPtocs, PalantirDataEventType.DATA);
// Publish the object to the base realm
conn.publishObjects(BasicLocatorUtils.getLocatorListFromLocatableList
                    (newPtocs));

/* Build an ISearchQuery to search for the Person entity that was just
* created, loaded, and published. This search query will find only those
* objects that meet all of the search conditions and that are in the data
* repository. In this example, the query should find the object that was
* just created. You can experiment with this search code by commenting out
* the store and publish code and then running the query.
*/
ISearchQuery sq =
          clientContext.getSearchFactory().getNewSearchQuery(
          OperatorType.INTERSECT);
sq.addObjectTypeTerm(PTObjectType.getByUri(
             "com.palantir.object.person"));

PropertyType name =
               PropertyType.getByUri("com.palantir.property.Name");

sq.addPropertyTerm(name, name.getComponentByUri("FIRST_NAME"), "Jane",
               SearchOperator.EQUALS);
sq.addPropertyTerm(name, name.getComponentByUri("LAST_NAME"),
                 "Palantirian", SearchOperator.EQUALS);
```

The `SearchResultsPager` is a convenience class. It contains no results itself; it has only access to the `ResultsPage` so that the while loop is easy to write.

```
// Takes the search results and hands them to a pager
SearchResultsPager pager = conn.search(sq);
// Add the results of the search to a Collection of Palantir Object Containers
ResultsPage<PTObjectContainer,PalantirSearchException> currentPage = pager;
Collection<PTObjectContainer> collPtocs = new ArrayList<PTObjectContainer>();
   while (currentPage.moreResultsAvailable()) {
     currentPage = currentPage.getNextPage();
     collPtocs.addAll(currentPage.getResults());
   }

// Inform the user how we plan to handle the results
if (collPtocs.size() < 1) {
     // Error, nothing found
     System.out.println("Error: no macthing object was found.");
     System.out.println("Please change your search, or
          add an object to your data.");
     System.exit(0);
   } else if (collPtocs.size() > 1) {
       // Too many things found
       System.out.println("Warning: more than one object matched.");
       System.out.println("Only the first object will be changed.");
}

/* Obtain the object locators from the collection of object results.
* Then load them into the current investigation, you will need to load
* the object because the following code changes one of its existing
* properties.
```

```
*/
Collection<Locator> locs =
BasicLocatorUtils.getLocatorListFromLocatableList(collPtocs);

// Load objects into current investigation
collPtocs = conn.objectLoad(locs,
          LoadLevelFactory.getFullDsrLoadedInstance());

// Get the first matching object in the collection
PTObjectContainer obj = collPtocs.iterator().next();


// Print out the properties for each object
// If the object contains an Eye Color property
Collection<Property> props = obj.getProperties();
PropertyType hairType =
            PropertyType.getByUri("com.palantir.property.HairColor");

System.out.println("Original Properties:\n---------------------------------
-----------------");
props = obj.getProperties(hairType);
for (Property prop : props) {
   System.out.println(prop.toTypeValueString());
}
/* Change the hair color property from dark brown to blonde. The change call
* is the same call as the call to create a totally new property.
*/
   Property p = null;

      // Change an existing property
      p = Property.attemptToCreate(conn, hairType,"blonde",Role.NONE);

      for (Property prop : props) {
         System.out.println("\ntrying to update hair color...
\n"+prop.toString());
         boolean up = prop.updatePropertyData(p);
         if(up)
           System.out.println("updated...");
      }

      // Make sure the change happened in memory by printing out a list of
      //new properties
      System.out.println("\nNew Properties:\n-----------------------------
----------------------");
      props = obj.getProperties();
      for (Property prop : props) {
        System.out.println(prop.toTypeValueString());
      }
/* You will need to store and publish the change to the base realm to make
* it available to other investigators on their next update:
*/
   conn.objectStore(collPtocs,PalantirDataEventType.DATA);

      conn.publishObjects(locs);

        client.shutdown();
```

```
            System.out.println("Published Changes.");
          } catch (PalantirException e) {
            System.out.println("Caught an Exception.");
            e.printStackTrace();
          }

          System.exit(0);
        }

    }
```

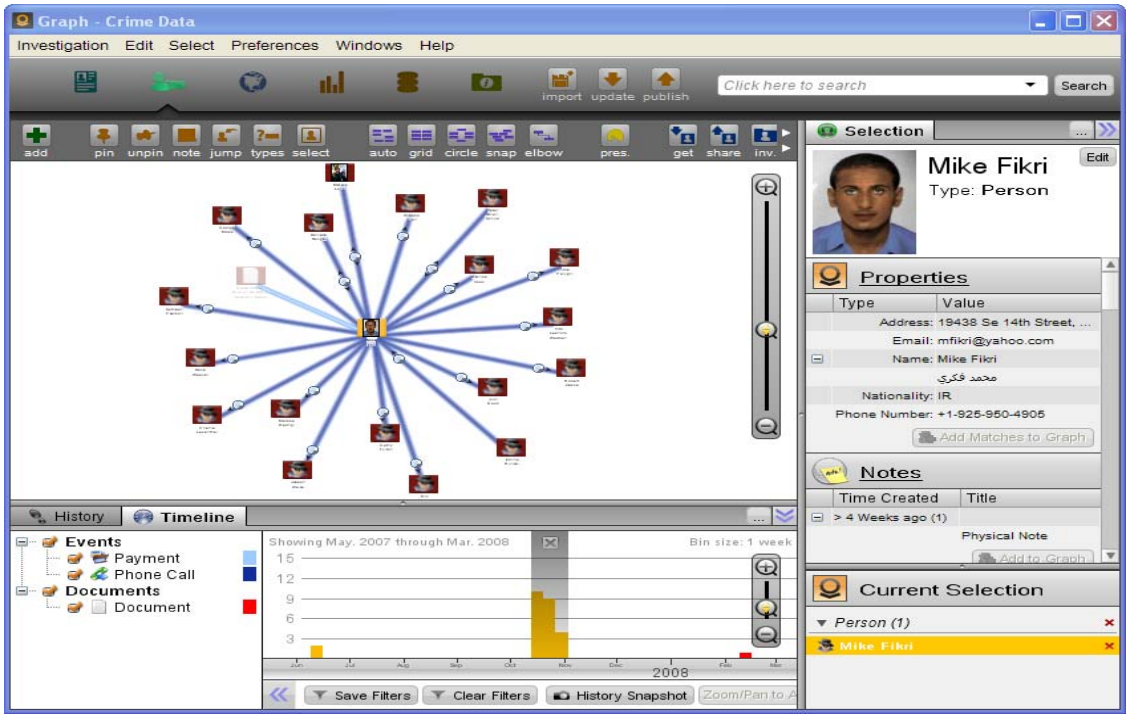
7 Helper Programming

This chapter explains how to add a helper to the Palantir Platform and contains the following topics:

- Understanding Helper Concepts
- Overview of the Helper APIs
- Implementing the HelperInterface
- Example of a Simple Object Property Helper

Understanding Helper Concepts

A helper is a graphical utility on the Workspace. The Workspace has several built-in helpers such as the **History** or **Timeline**. A good way to think of a helper is that it helps a user to analyze data. The **Histogram**, for example, aggregates property information across a selection and allows a user to drill down through the data.



Helpers can also store information about data so that you can access it later. For example, the **History** helper stores snapshots of the graph over time. While most helpers assist in analysis, your helper need not be an analysis tool. You could, for example, write a helper that allows a user to create reminders. In fact, there is virtually no limit to how you can apply helpers, some ideas could involve:

- importing data from external services or applications
- configuring the presentation of objects
- applying transformational operations on selected objects
- collaborating between team members

A helper must be paired with at least one Palantir application and might be paired with multiple applications. You can pair a helper with multiple applications. For example the **History** helper appears with the **Graph** and the **Map** application — among others. Keep this pairing ability in mind when designing your helper as it might be suitable for more than one application.

Considerations in Helper Programming

Helpers, through the `PalantirConnection` API, can access any of the data available in the Data Repository. A custom helper can select objects in a application, manipulate the objects and store objects back into the repository. Through the connection, the platform ensures that the Palantir objects are versioned and stored just as with any standard helper.

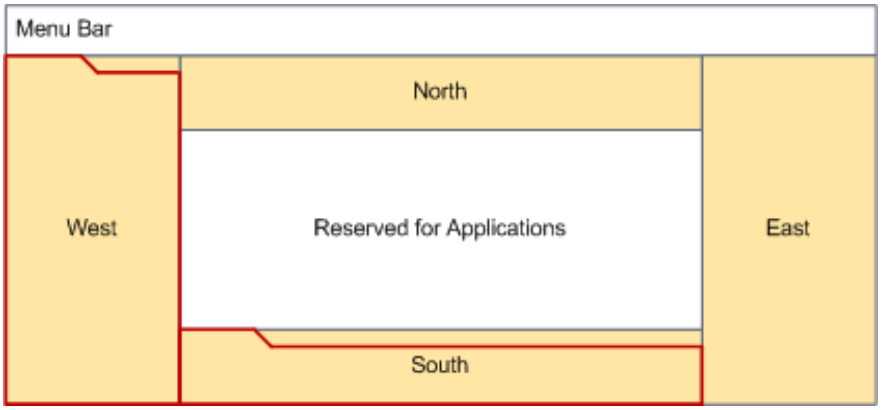
Keep in mind that standard Palantir security applies to your helper. Users cannot use custom helpers to access objects outside of their permission level. However, Palantir does not restrict the type of code you can place inside a helper — this means you can pull external data into the Palantir Workspace. If you pull external data into your helper, you are responsible for ensuring a user has the proper permissions to view the external data.

Your helper might need to store some application state associated with external objects. This is permitted. However, you should be careful not to persist any Palantir objects (`PTObject`) with your application state. Palantir objects should only be stored through a `PalantirConnection`.

If your application state requires some correlation to Palantir data, obtain the object IDs of the pertinent Palantir objects and store those together with your external data. The IDs will allow you to restore your helper to the appropriate state.

Helper Layout

Helpers display in a tabbed pane within the Workspace. You can decide how to orient your helper within an application. Palantir uses a quadrant layout similar to the Java `BorderLayout` class:



When you construct your helper, you specify whether the helper has a horizontal orientation, vertical orientation, or both. The orientation constrains the quadrants to which a you can drag a helper. Horizontal helpers can go into the North or South quadrant. Vertical helpers can go into the West or East quadrant. Helpers that can take both orientations can appear in any quadrant.

Overview of the Helper APIs

A helper is a graphical utility on the Workspace such as **Search** or **History**. Helpers display in a tabbed pane within the Workspace. A helper can be associated with one or more Palantir applications. For example the **History** helper appears with the **Graph** and the **Map** application — among others. The following is a list of the key APIs you need to use when programming a helper:

API	Description
`AbstractHelperFactory`	Contains the default helper factory members. Your custom `HelperFactory` class should extend this API.
`HelperInterface`	Represents the default helper interface. You custom helper must implement all the methods of this class.
`HelperFactory`	Acts as the factory for helper objects. Use the `createHelper()` method to create an instance of your custom `HelperInterface`.

API	Description
`ApplicationInterface`	The interface to the Workspace applications such as the **Browser**, **Graph**, **Map**, and so forth. When you call `createHelper()` you pass it an application interface. When you create a helper, you provide it with the URI of the application on which it appears. You can reach this using the `APPLICATION_URI` field on each subclass of `ApplicationInterface`.
`PalantirContext`	Represents an instance of a client in the Palantir Platform.

Components of a Simple Helper Implementation

You can construct a simple helper in a public class that extends the `AbstractHelperFactory`. Within the class, you implement the `HelperInterface` as a `protected static` class. The following example code shows the essential components of a helper:

```
import java.awt.Dimension;
import javax.swing.KeyStroke;
import com.palantir.api.workspace.AbstractHelperFactory;
import com.palantir.api.workspace.ApplicationInterface;
import com.palantir.api.workspace.HelperInterface;
import com.palantir.api.workspace.PalantirContext;
public class APropertyHelperFactory extends AbstractHelperFactory {
   public APropertyHelperFactory(String title, String[] pairWith,
      Integer[] orientations, Dimension dim, KeyStroke keyStroke,
      String uri) {
     super(title, pairWith, orientations, dim, keyStroke, uri);
     // TODO Auto-generated constructor stub
   }
   public HelperInterface createHelper(PalantirContext arg0,
      ApplicationInterface arg1) {
     // TODO Auto-generated method stub
     return null;
   }
   protected static class PropertyHelper implements HelperInterface {
     // TODO Implement the inherited methods
   }
```

Within the implementation of the `HelperInterface` is where you build the actual helper user interface. It is here that the main work of building the application takes place.

Implementing the HelperInterface

Your custom helper will always implement all the methods of the `HelperInterface`. This is the core interface for custom helpers and contains the following methods:

Method	Description
`dispose()`	Unregisters the helper from an application.
`getDefaultPosition()`	Returns the default position of the helper in the interface.
`getDisplayComponent()`	Returns a fully configured `JComponent` representing this helper.
`getFactory()`	Returns the `HelperFactory` associated with this helper.
`getFrameIcon()`	Gets the image used when constructing the helper's `JFrame`.
`getIcon()`	Icon used on the tabs of the `JTabbedPane` and on the header bar of the `HelperFrame`.
`getTitle()`	Determines the text used in the tab to identify the helper.
`initialize()`	Registers the helper with the application. This method is called in preparation for added the helper to an application/tab. Generally, you add any global listeners through this call.
`setConstraint()`	Tells the helper what constraint to expect when the helper is added.
`setOwners()`	Sets the owner of the helper interface. The owner of the interface is a window within which exists the helper on a tabbed pane.

Listener APIs

You will need to add a Palantir listener to your helper if you want it to react to Palantir application events on the Workspace. The following Palantir-specific listeners are available for use with helpers:

API	Description
`PalantirInvestigationListener`	Use this to capture investigation open and close events. Also, use this listener to learn when the user navigates forward or backward in the investigation's history.
`PalantirObjectListener`	Use this to capture when a `PTObject` is created, changed, or deleted.

API	Description
`SelectionAgentListener`	Use this to capture when objects in the Workspace are selected. This also notifies when a selection changes or is filtered.

The HelperInterface Implementation

The following example code shows an empty implementation of the `HelperInterface`. Typically, in a small application, you implement your listener as an inner class (named or anonymous) in the initialize method of your helper.

```
protected static class PropertyHelper implements HelperInterface {
    public void dispose(ApplicationInterface arg0) {
      // TODO Auto-generated method stub
    }
    public String getDefaultPosition() {
      // TODO Auto-generated method stub
      return null;
    }
      public JComponent getDisplayComponent() {
      // TODO Auto-generated method stub
      return null;
    }
      public HelperFactory getFactory() {
      // TODO Auto-generated method stub
      return null;
    }
      public Image getFrameIcon() {
      // TODO Auto-generated method stub
      return null;
    }
      public Icon getIcon() {
      // TODO Auto-generated method stub
      return null;
    }
      public String getTitle() {
      // TODO Auto-generated method stub
      return null;
    }
      public void initialize(ApplicationInterface arg0) {
      // TODO Auto-generated method stub
    }
      public void setConstraint(String arg0) {
      // TODO Auto-generated method stub
    }
      public void setOwners(PalantirFrame arg0, ApplicationContext arg1) {
      // TODO Auto-generated method stub
    }
}
```

Example of a Simple Object Property Helper

This example constructs a helper that displays the properties associated with objects a user has selected on the Graph. This example illustrates both how to add a listener and use it to select objects in an application. You can download a ZIP package containing the project code for this example.

The first set of imports represent the graphical AWT and SWing elements that the sample uses to build the helper pane:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Image;
import javax.swing.BorderFactory;
import javax.swing.Icon;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
```

Finally, you must import the helper APIs as well as the supporting Palantir APIs you need:

```
import com.palantir.api.workspace.AbstractHelperFactory;
import com.palantir.api.workspace.ApplicationContext;
import com.palantir.api.workspace.ApplicationInterface;
import com.palantir.api.workspace.HelperFactory;
import com.palantir.api.workspace.HelperInterface;
import com.palantir.api.workspace.PalantirContext;
import com.palantir.api.workspace.applications.GraphApplicationInterface;
import com.palantir.api.workspace.PalantirFrame;
import com.palantir.api.workspace.dataitem.DataItemModel;
import com.palantir.api.workspace.selection.*;
import com.palantir.services.ptobject.*;
```

Your custom helper must extend the `AbstractHelperFactory` and, of course, provide a constructor. The constructor must reference the application with which the helper appears. This value will be a subclass of the `ApplicationInterface`.

```
public class PropertyHelperFactory extends AbstractHelperFactory {
    public PropertyHelperFactory() {
      super("Property Helper",
        new String[] { GraphApplicationInterface.APPLICATION_URI },
        new Integer [] { SwingConstants.VERTICAL },
        new Dimension(330,500),
        null,
        "com.palantir.helper.MyPropertyHelper");
}
```

You can see the constructor requires the helper's URI. The `ptplugin.xml` file should reference the same URI in the `uri` element.

Next, you implement the `HelperFactory.createHelper()` method to create an instance of your helper:

```
public HelperInterface createHelper(PalantirContext palantirContext,
     ApplicationInterface application) {
   return new PropertyHelper(this, palantirContext, application);
```

Within your factory, construct a `PropertyHelper` by implementing the `HelperInterface`. It is within this implementation that you lay out the helper graphically and instantiate listeners to interact with the application.

```
protected static class PropertyHelper implements HelperInterface {
private HelperFactory factory;

private JPanel panel;
private JLabel label;

@SuppressWarnings("serial")
public PropertyHelper(HelperFactory factory,
    PalantirContext palantirContext, ApplicationInterface application) {
      this.factory = factory;
      panel = new JPanel(new BorderLayout());
      JPanel topPanel = new JPanel(TableLayouts.create(
                                    "1,p,p","p,p,p,p", 8, 8));
      opPanel.setBackground(Colors.UI.getSecondaryBackgroundColor());
      topPanel.setBorder(BorderFactory.createMatteBorder(0, 0, 1, 0,
           Color.GRAY));
    label = new JLabel("<b>No Objects Selected</b>");
    FontSizeManager.registerComponent(+0, label);
    topPanel.add(label, "1,0,2,0");

    panel.add(topPanel, BorderLayout.NORTH);
}
public String getDefaultPosition() {
    return BorderLayout.EAST;
}
public JComponent getDisplayComponent() {
    return panel;
}
public Image getFrameIcon() {
    return null;
}
public Icon getIcon() {
    Icon myicon=null;
    try {
       myicon = new
            ImageIcon(ImageIO.read(SimpleHelperFactory.class.getResource
               ("/small_world.gif")));
       return myicon;
    } catch (Exception e) {
         e.printStackTrace();
    }
    return myicon;
}
public HelperFactory getFactory() {
    return factory;
}
public String getTitle() {
    return "Simple Helper";
}
```

The `initialize()` method is called when the helper is registered with the application. Within this method you instantiate an anonymous implementation of `SelectionAgentListener`.

```
public void initialize(ApplicationInterface app) {
    SelectionAgentListener sal = new SelectionAgentListener() {
```

```
     public void handleFilterEvent(SelectionAgentEvent event) {
}
```

When the `handleSelectionEvent()` is triggered, a collection of items (`DataItemGroup`) is sent with the event to the helper.

```
public void handleSelectionEvent(SelectionAgentEvent event) {
     String text = "<html><b>Objects Selected</b><br><br>";
     int counter = 0;
```

In this section, the helper converts the items to `PTObjectContainer` objects and then traverses each object to obtain its title and properties:

```
for (DataItemModel dim : event.getItemGroup().getList()){
     if (dim.isPTOC()){
       PTObjectContainer tempPTOC = dim.getPTOC();
            text += "<b>" + tempPTOC.getTitle() + "</b><ul>";
       for(Property p : tempPTOC.getProperties()){
       text += "<li>" + p.getTypeDisplayName() + ": " +
       p.getDisplayString()+ "</li>";
      }
       text += "</ul>";
       counter++;
      }
      if(counter == 0)
       label.setText("No Objects Selected");
      else
       label.setText(text + "</ul>" + counter + " objects selected</html>");
      }
       public void handleUpdateEvent(SelectionAgentEvent event) {
       // do nothing
      }
};
```

The following code attaches this `SelectionAgentListener` to the `ApplicationInterface` passed to through the `initialize()` method:

```
   if (app.getSelectionAgent() != null &&
     app.getSelectionAgent().getSelectionAgentSupport() != null) {
     app.getSelectionAgent().getSelectionAgentSupport().
        addSelectionAgentListener(sal);
     }
   }

   public void dispose(ApplicationInterface arg0) {
     // do nothing
   }

   public void setOwners(PalantirFrame arg0, ApplicationContext arg1) {
     // do nothing
   }

   public void setConstraint(String constraint) {
     // do nothing
   }
}
```

The ptplugin.xml File

The following illustrates the `ptplugin.xml` file you would use to describe this sample within the Palantir Platform:

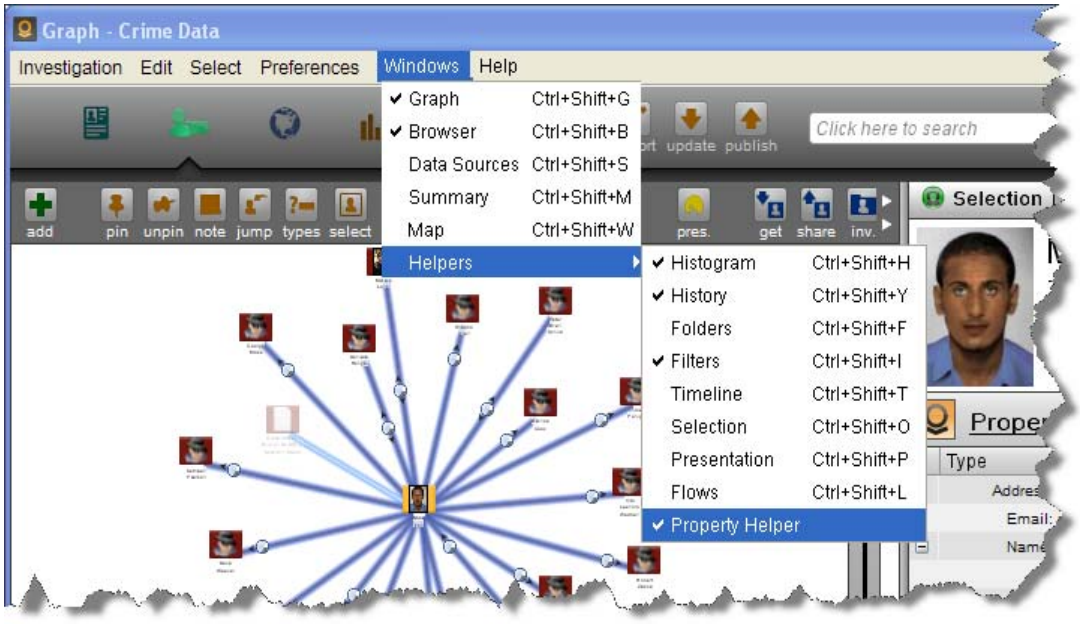
```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantir.com/schemas/ptplugin/v2.1.0">
    <uri>com.palantir.helpers.PropertyHelper</uri>
    <displayName>Simple Helper</displayName>
    <version>
      <major>1</major>
      <minor>0</minor>
    </version>
    <targetVersion>3.0.3</targetVersion>
    <component>
      <interface>com.palantir.api.workspace.HelperFactory</interface>
      <implementation>com.palantir.helper.example.PropertyHelperFactory</
implementation>
    </component>
</ptplugin>
```

Testing Your New Helper

Build and deploy your helper. Then, launch the Workspace using the **Run As > Palantir Workspace Plugin** in your project and do the following:

1. Open an investigation that has one or more objects.
2. Select one or more investigational objects.
3. Choose **Windows > Helpers > Property Helpers**.

   Your completed program should result in a view similar to the following:



8 Managing Application State and Plugin Preferences

This section explains how to write plugins that make use of application state or plugin preferences. The following topics are covered:

- Understanding Application State
- Using Application State in Your Plugin
- Using Plugin Preferences

Understanding Application State

This section explains what application state Palantir supports. You should read this section to understand important concepts and limitations you must be aware of when working with application state.

Application State in Palantir

The Palantir Platform allows you to write custom code that interacts with Palantir data. You can also write custom code that has its own data or state. Palantir provides utility code that you can use to store and retrieve state between Workspace sessions. You can store state for:

- a specific investigation and specific user
- a specific investigation shared by all users
- a specific user independent of any particular investigation
- globally, shared by all users and investigations

Palantir segregates your application state in its own location of the repository. It is important that you do not persist any Palantir objects within your application state.

If your application state requires some correlation to Palantir data, obtain the object IDs of the pertinent Palantir objects and store those together with your plugin's data. The IDs will allow you to restore your helper to the appropriate state.

When you create application state, you must specify an identifier for an access control list (ACL). This ACL defines the permission to Palantir data; the Palantir Platform ensures that it is respected. You are responsible for ensuring a user has the proper permissions to view the state or other external data that you display in your plugin.

Note: If you wish to store a file in application state, be aware that there is a 10 MB limit on the file size.

Plugin Preferences and State

Plugin preferences represent a special case of state. Preference state is stored and managed by Palantir *entirely separately* from application state. Unlike application state, you must define the preferences for your plugins via the `preferences` section in the `ptplugin.xml` file.

The `ptplugin.xml` file specifies the initial default set of plugin preferences. Once a plugin is loaded, preference values are maintained in the backend database. Changes made with the command line replace the initial value(s) specified in the plugin's `ptplugin.xml` file.

Once you have loaded a plugin and made a change with the CLI, the only way to change a preference using the Palantir Platform is with the CLI or an API call. You cannot delete a preference with the CLI, through an API call, or by reloading a `ptplugin.xml` file. As global parameters, preferences apply to all users; changing a preference value changes it for all users.

Note: Defaults that are unchanged by the CLI are updated by new values in the `preferences` section of the `ptplugin.xml` file.

Keep in mind, the association between the plugin URI and the preference URI persists in the database. This means if you remove a preference from the `ptplugin.xml` file and add a preference with the same URI back through a later version of your plugin, the platform will reuse the last value stored in the database regardless of any changes in the `ptplugin.xml` file.

You can read about writing a helper with preferences Using Plugin Preferences on page 102.

To Note when Developing Plugins with State or Preferences

If you are developing plugins with application state or preferences, you need to anticipate this when setting up your plugin project. A Quickstart environment does not automatically support application state or plugin preferences. You must modify the QuickStart's `dispatch.prefs` file and add a `DEVELOPER_APP_STATE_URIS` preference.

The `dispatch.prefs` file is in the Quickstart `INSTALL_DIR` folder. To use this preference, specify a comma separated list of URIs for your plugins that contain application state or plugin preferences. For example:

```
DEVELOPER_APP_STATE_URIS=com.mycompany.plugin1, com.mycompany.plugin2
```

This registers the necessary URIs with the server so that it can support the requisite states or preferences for your plugin.

If you do not specify this preference in `dispatch.prefs`, your plugin might not load properly using the **Run As > Palantir Workspace Plugin** feature of the Eclipse plugin. If you do load the plugin either directly through the **Run As** feature the state or preferences will not behave correctly.

Using Application State in Your Plugin

This section introduces the APIs that support application state. This section also explains the various ways you can use these APIs. Finally, this section includes a code example you can download for Eclipse and a walk through of the application state code in the example.

The State APIs

The `PalantirConnection` API is the main API that you will use for plugins that appear as helpers in the Workspace and that have state. This API contains the following key methods for working with application state:

Method Name	Description
`changeApplicationStateAcl()`	Changes the access control list associated with the state.
`checkApplicationStateExists()`	Determines if state exists for a given application.
`deleteApplicationState()`	Deletes application state from the data repository.
`getApplicationState()`	Reads the data stored in application state.
`putApplicationState`	Writes an application state to the repository. This is an overloaded method. There are two versions of this method, concurrent and standard. The concurrent method takes an old and new value as parameters. It is intended for situations where there is a potential race condition. This method returns true if the put was successful or false if it was not. The standard method takes a single, new value. It simply replaces the old value with the new.

The application state methods all take the following parameters:

`uri`	The plugin identifier as defined in the ptplugin.xml file.
`key`	A String value similar to a hash table key.

`globalUser`	A boolean specifying whether the key-uri pair is per user or for all users.
`globalRealm`	A boolean specifying whether the key-uri pair is per investigation or for all investigations.

You use these values to compartmentalize the state. The `key` represents the compartment while the `globalUser` and `globalRealm` combination determines how the key applies. The following table illustrates the various combinations and what they mean

Type	Description
`globalUser=true` `globalRealm=true`	globally, shared by all users and investigations
`globalUser=false` `globalRealm=true`	a specific user independent of any particular investigation
`globalUser=true` `globalRealm=false`	a specific investigation shared by all users
`globalUser=false` `globalRealm=false`	a specific investigation and specific user

You are responsible for managing the permissions associated with the state you create. Application state is secured by means of an access control list. You use the `aclId` parameter to define this access. You are responsible for ensuring this value is the correct value for the data in your state. After you have created application state, you can change the permissions on it using the `changeApplicationStateAcl()` method. You must have ownership permissions on the plugin to change its `aclId`.

Note: Palantir recommends that application state which is global realm should set an ACL with Administrator-own and Everyone-write. Application state which is per investigation should use the investigation's creation manually entered data ACL.

Plugins Not Requiring the Workspace

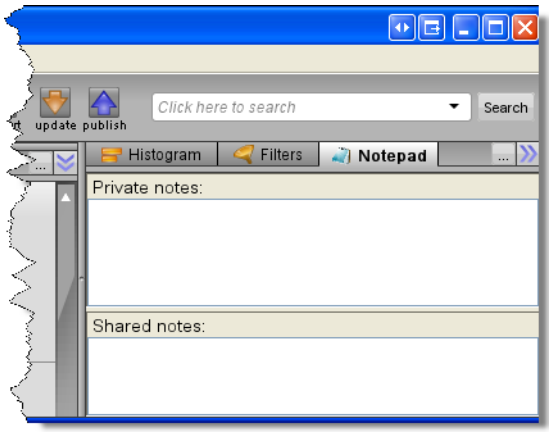
The `PalantirClientContext` represents a client that has a connection to a Palantir server but not necessarily appearing as a helper in the Workspace. This API has the following methods:

- `changeApplicationStateAclGlobalRealm()`
- `checkApplicationStateExistsGlobalRealm()`
- `deleteApplicationStateGlobalRealm()`
- `getApplicationStateGlobalRealm()`
- `putApplicationStateGlobalRealm`

These methods have the same purpose as the methods on `PalantirConnection`. The difference is that the `PalantirClientContext` methods all assume the global realm is true and only take a `globalUser` value. As a result, you cannot store investigation-specific data with the `PalantirClientContext` methods.

Walk Through of the Sample

The `NotepadHelper` project was created to illustrate important points of application state. If you like, you can download a ZIP package that contains the entire sample project. After the plugin is installed, the Notepad helper provides an area where you can enter public or private notes about an investigation. The following dialog shows how the Notepad helper appears in the Workspace:



The `NotepadHelper` project class has three source files, `NotepadHelper.java`, `NotepadHelperFactory.java`, and the `NotepadHelperRegistry.java`. The `NotepadHelperRegistry.java` contains all the code for tracking and managing state among the different helpers.

NotepadHelper

The `NotepadHelper.java` file contains the constructor for the user interface. An enumerator `Type` defines controls the availability of the application data stored in each pane:

```
class NotepadHelper implements HelperInterface {
    enum Type {
      PRIVATE(false), SHARED(true);
      final boolean isGlobalUser;
      PrivacyType(boolean isGlobalUser) { this.isGlobalUser = isGlobalUser; }
    }
...
```

Note: As you would expect, private data is stored per user and shared data is available to all users. You can see this enumerator passed in the construction of each panel:

```
...
NotepadHelper(NotepadHelperFactory factory, NotepadHelperRegistry registry) {
this.factory = factory;
```

```
    this.registry = registry;
    this.palantirContext = palantirContext;

    displayComponent = new JSplitPane(
       JSplitPane.VERTICAL_SPLIT,
       true /* newContinuousLayout */,
       buildPanel(PrivacyType.PRIVATE, " Private notes:"),
       buildPanel(PrivacyType.SHARED, " Shared notes:"));
    displayComponent.setResizeWeight(0.5);
  }
```

The following method builds the panels in the UI and, you will notice it uses a getter and setter on the `NotepadRegistry` instance to populate the notes in their respective panels.

```
private JPanel buildPanel(final PrivacyType type, String labelText) {
    JPanel panel = new JPanel(new BorderLayout());
    JLabel label = new JLabel(labelText);
    panel.add(label, BorderLayout.NORTH);

    palantirContext.registerComponentForFontManagement(1, label);
    @SuppressWarnings("serial")
    JTextArea textArea = new JTextArea(registry.getText(type)) {
...
    textArea.addFocusListener(new FocusListener() {
      public void focusLost(FocusEvent event) { registry.setText(type,
           getText(type)); }
      public void focusGained(FocusEvent event) { /* Do nothing. */ }
    });
...
  return panel;
  }
```

NotepadHelperRegistry

Recall that application state is secured by means of an access control list. This is defined in `NotepadHelperRegistry.java` as `com.google.common.collect.ImmutableList`:

```
class NotepadHelperRegistry {
  private static final List<AccessControlItem> ACIS = ImmutableList.of(
     new AccessControlItem(NativeGroup.ADMINISTRATORS.getGroupId(),
               Permissions.DRWO_PERMISSIONS),
     new AccessControlItem(NativeGroup.EVERYONE.getGroupId(),
               Permissions.DRW_PERMISSIONS));
```

The application includes a KEY and a URI value for use by the application state methods.

```
  private static final String KEY = "notes";
  private static final String URI = NotepadHelperFactory.URI;
```

There are two private methods `loadTextFromDatabase()` and `saveTextToDatabase()` that control the helper's loading and storage of state. The load method appears as follows:

```
private String loadTextFromDatabase(PrivacyType type) {
  String text = "";
  PalantirConnection conn = context.getPalantirConnection();
  try {
    if (conn.checkApplicationStateExists(URI, KEY, type.isGlobalUser,
         IS_GLOBAL_REALM)) {
      byte[] bytes = conn.getApplicationState(URI, KEY, type.isGlobalUser,
                   IS_GLOBAL_REALM);
      try {
        text = new String(bytes, "UTF-8");
```

```
        } catch (UnsupportedEncodingException e) {
          throw new AssertionError(e);
        }
      }
    } catch (PalantirException e) {
      throw new RuntimeException(e);
    }
    notepadTypeToNoteText.put(type, text);
    return text;
}
```

The method `checkApplicationStateExists()` passes in the `URI` and `KEY`. The method pulls the a `TYPE.globalUser` value from the panel and always specifies a `globalRealm` of false. From this combination, you can see that the application state is stored as follows

Type	Associates
`globalUser=true` `globalRealm=false`	**Shared notes** with a specific investigation that all users can view.
`globalUser=false` `globalRealm=false`	**Private notes** with a specific user and investigation combination.

The save method uses the standard version of the `putApplicationState()` method.

```
private synchronized void saveTextToDatabase(PrivacyType type) {
      byte[] newValue;
    try {
      newValue = getText(type).getBytes("UTF-8");
    } catch (UnsupportedEncodingException e) {
      throw new AssertionError(e);
    }
    PalantirConnection conn = context.getPalantirConnection();
    long aclId = conn.getAccessControlList(ACIS).getId();
    try {
      conn.putApplicationState(URI, KEY, newValue, aclId, type.isGlobalUser,
                               IS_GLOBAL_REALM);
    } catch (PalantirException e) {
      throw new RuntimeException(e);
    }
}
```

The sample uses the standard method as opposed to the concurrent version because this sample unconditionally stores the text that each user has typed. The sample will never encounter a race condition.

NotedpadHelperFactory

The `NotepadHelperFactory.java` file fulfills the requirements of every helper implementation. These are described in Chapter 7, Helper Programming. This helper's panel is supported in all the Workspace's applications.

```
public class NotepadHelperFactory implements HelperFactory {
   static final String URI = "com.palantir.plugin.helper.notepad";

   private static final List<String> APPLICATIONS = ImmutableList.of(
      BrowserApplicationInterface.APPLICATION_URI,
      DataSourcesApplicationInterface.APPLICATION_URI,
      GraphApplicationInterface.APPLICATION_URI,
      MapApplicationInterface.APPLICATION_URI,
```

```
      SummaryApplicationInterface.APPLICATION_URI);
...
```

Edits entered in one Workspace application are carried through to all applications. The factory includes the creation of the registry and a keyboard accelerator:

```
private NotepadHelperRegistry registry;
public synchronized HelperInterface createHelper(
      PalantirContext palantirContext, ApplicationInterface application) {
    if (registry == null) {
      registry = new NotepadHelperRegistry(palantirContext);
    }

    return new NotepadHelper(this, registry, palantirContext);
  }
  public KeyStroke getAccelerator() {
    return KeyStroke.getKeyStroke("control shift N");
  }
...
```

Update dispatch.prefs and Run the Sample

This code is only necessary for the QuickStart development environment. The `dispatch.prefs` file defines the `DEVELOPER_APP_STATE_URIS` value as follows:

```
DEVELOPER_APP_STATE_URIS=com.palantir.plugin.helper.notepad.NotepadHelper
```

After updating `dispatch.prefs` restart the server and do the following:

1. Start Eclipse if it is not already started.
2. Go to the Package Explorer.
3. Right click the **NotepadHelper** project.
4. Choose **Run As > Palantir Workspace Plugin** from the context menu.

Were you to move this plugin to a production environment, you would not need to modify the `dispatch.prefs` file. Instead, you would load the file as specified Deploying Custom Code in a Staging or Production Environment in Chapter 4.

Using Plugin Preferences

Preferences are created through a `ptplugin.xml` specification. You can only interact with preferences for plugins you have installed. Once you have installed a plugin, you can use the Palantir APIs for managing their preferences. For example, you might create a dialog for you helper that allows a user to change a helper's preferences. This section explains how to use the preferences APIs.

Plugin Preferences APIs

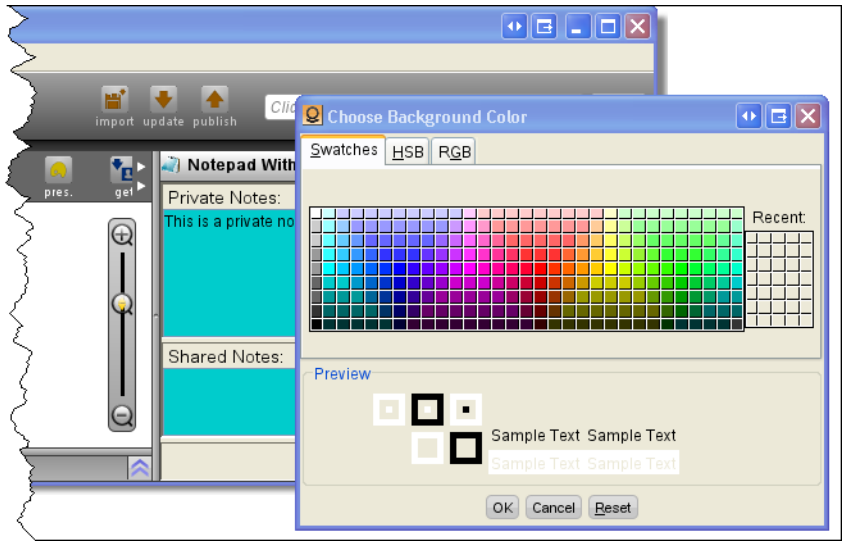
The `com.palantir.api.pluginpreferences` package contains APIs for interacting with plugin preferences. You use `PluginPreference` to get information about a specific preference. To manage plugin state, use the `PluginPreferences` API.

The `PalantirClientContext.getPluginPreferences()` allows you to obtain the preferences associated with a particular plugin. If you are implementing the `com.palantir.api.workspace.map.Gazetteer` interface, you must call the `Gazetteer.initialize()` method to access and store a `PluginPreferences` object that can be used by calls to a gazetteer.

The Palantir Platform stores preference data as owned by the ontology editor's group and readable by everyone. You can write a plugin A that modifies the preferences for a plugin B as long as plugin B is installed and you have the appropriate URI.

Walk Through a Sample Application

The `NotepadHelperwithPreferences` project (you can download a ZIP package of the project) expands on the earlier `NotepadHelper` to include a plugin preference that allows users to set their preference for notepad's background color using the Java `JColorChooser` dialog. The result of this helper appears in the Workspace as follows:



The new code in for this helper all resides in the `NotepadHelper.java` file and the `ptplugin.xml` file.

The ptplugin.xml File

The `ptplugin.xml` file contains a `preferences` section that defines a color preference:

```
<?xml version="1.0" encoding="UTF-8"?>
<ptplugin xmlns="http://www.palantir.com/schemas/ptplugin/v2.1.0">
    <uri>com.palantir.plugin.helper.notepadwithpreferences</uri>
    <displayName>Notepad Helper with Preferences</displayName>
    <version>
      <major>1</major>
      <minor>1</minor>
    </version>
    <targetVersion>3.0.3</targetVersion>
    <component>
      <interface>com.palantir.api.workspace.HelperFactory</interface>
      <implementation>
```

```
com.palantir.plugin.helper.notepadwithpreferences.NotepadHelperWithPreferenc
esFactory
</implementation>
    </component>
    <preferences>
      <preference
uri="com.palantir.plugin.helper.notepadwithpreferences.color">
        <name>Color as RGB Value</name>
        <type>integer</type>
        <default><integer>-1</integer></default>
      </preference>
    </preferences>
</ptplugin>
```

NotepadHelperWithPreferencesRegistry.java

The API for preferences access through the `PalantirContext`. Since this sample places all the code for getting and setting preferences in the UI code. The context is obtained through the registry code:

```
...
class NotepadHelperWithPreferencesRegistry {
    ...
    private static final boolean IS_GLOBAL_REALM = false;
    private final PalantirContext context;
    ...
    NotepadHelperWithPreferencesRegistry(PalantirContext context) {
      this.context = context;
...
```

NotepadHelperWithPreferences.java

The `NotepadHelperWithPreferences.java` file contains the majority of code for adding preferences to the original Notepad plugin.

After the initial enumerators, a `pluginprefs` variable is defined that will allow the code to reference the preferences:

```
...
class NotepadHelper implements HelperInterface, ActionListener {
     enum Type {
       PRIVATE(false), SHARED(true);
       final boolean isGlobalUser;
       Type(boolean isGlobalUser) { this.isGlobalUser = isGlobalUser; }
    }
    private PluginPreferences pluginPrefs;
...
```

This code adds a panel below the two note areas to hold the new **Change Notepad Background Color** button. The `JSplitPanel` for the two notes was nested inside another `JSplitPanel` and the button panel becomes the bottom of the main pane:

```
...
notepadComponent = new JSplitPane(
        JSplitPane.VERTICAL_SPLIT,
        true /* newContinuousLayout */,
        buildNotepadPanel(PrivacyType.PRIVATE, " Private Notes:"),
        buildNotepadPanel(PrivacyType.SHARED, " Shared Notes:"));
     notepadComponent.setResizeWeight(0.5);
```

```
     prefPanel = buildPrefPanel();
     displayComponent = new JSplitPane(
        JSplitPane.VERTICAL_SPLIT,
        true /* newContinuousLayout */,
        notepadComponent,
        prefPanel);
     displayComponent.setResizeWeight(1);
...
```

This button panel is specified simply as:

```
...
private JPanel buildPrefPanel() {
        JPanel panel = new JPanel(); //use FlowLayout
        JButton button = new JButton("Change Background Color");

        button.addActionListener(new ColorChoosingListener());
        panel.add(button);

        return panel;
    }
...
```

The retrieval of the initial color is done in the construction of a new `ColorChoosingListener`:

```
...
try {
public class ColorChoosingListener implements ActionListener {
     public void actionPerformed(ActionEvent e) {
       Color newColor = JColorChooser.showDialog(
          getDisplayComponent(),
          "Choose Background Color",
          displayComponent.getTopComponent().getBackground());

       if (newColor == null) {
         return;
       }

       int colorAsRGB = newColor.getRGB();
       try {
         if (pluginPrefs != null) {
          pluginPrefs.setIntegerPreference(COLOR_URI, colorAsRGB);
         }
       } catch (PalantirException exception) {
         JOptionPane.showMessageDialog(null,
            "You do not have appropriate permissions to modify the background
color.",
            "Error Modifying Background Color",
            JOptionPane.ERROR_MESSAGE);
         return;
       }

       Collection<JTextArea> areas = notepadTypeToTextArea.values();
       for (JTextArea area : areas){
         area.setBackground(newColor);
        }
        }
    }
      ...
```

You will notice that the code must handle the return of a `PalantirException` from `setIntegerPreference()`.

The original value of the preference is obtained when the `JPanel` is constructed.

```
...
if (pluginPrefs != null) {
       PluginPreference colorPreference =
pluginPrefs.getPreference(COLOR_URI);
       int colorAsInteger = colorPreference.getInteger();
       textArea.setBackground(new Color(colorAsInteger));
     } else {
       textArea.setBackground(Color.WHITE);
     }
...
```

Update dispatch.prefs and Run the Sample

This code is only necessary for the QuickStart development environment. The `dispatch.prefs` file defines the `DEVELOPER_APP_STATE_URIS` value as follows:

```
DEVELOPER_APP_STATE_URIS=com.palantir.plugin.helper.notepadwithpreferences
```

After updating `dispatch.prefs` restart the server and do the following:

1. Start Eclipse if it is not already started.
2. Go to the Package Explorer.
3. Right click the **NotepadHelper** project.
4. Choose **Run As > Palantir Workspace Plugin** from the context menu.

Were you to move this plugin to a production environment, you would not need to modify the `dispatch.prefs` file. Instead, you would load the file as specified Deploying Custom Code in a Staging or Production Environment in Chapter 4.

Did You Spot the Flaw in the Sample Project Design?

There is a flaw in this sample project's overall design. If you have read this far you should have a good enough understanding of application state and plugin preferences to spot it. What is the design flaw?

Application state has the ability to store state per user. You cannot store preferences per user; preferences are always global. If you run the `NotepadHelperWithPreferences` plugin as admin user and as another user, both users can change the preference globally.

Plugin preferences are intended for global data that an administrator would set for all users. For example, you might use them to specify a resource location or performance tweaking. If you wanted a graphical plugin helper that an administrative user would use to interact with such global settings, you would need to restrict the plugin to that user.

9 Customizing Classification Based Access Control

This chapter introduces you to the interfaces you need to implement when customizing classification-based access control features. The following topics are discussed:

- Overview of Classification Customization
- Configuring Classifications in a QuickStart Environment
- Writing a Custom Classification Chooser
- Writing a Java-based Rules Engine
- Writing a Classification String Handler

This documentation assumes you are already familiar with the administration of classification-based access control in the Palantir Platform. If you are not familiar with this feature, see Working with Classification Strings in the *Palantir Administrator's Guide*.

Overview of Classification Customization

The following sections introduce the interface you can use when customizing classification. Deploying the final customizations is also covered.

The Classification Customization APIs

The `com.palantir.commons.security` package contains the APIs you can use for your customization. You can extend the classification at three different points:

`ClassificationChooserFactoryPlugin`	Provides a classification chooser. This is the user interface users use to combine markings into strings.
`ClassificationRulesEnginePlugin`	Provides a Java-based rules engine. This is useful when the XML rules are too restrictive for your implementation.
`ClassificationStringHandlerPlugin`	Provides a means to define a custom classification string handler.

All three interfaces are singletons, you can only have one of each plugin registered in your system at a time.

Note: You can also create a custom investigation defaults provider. This last customization is not specific to classifications.

Deploying a Custom Classification Plugin

Both the `ClassificationStringHandlerPlugin` and the `ClassificationRulesEnginePlugin` are ontology plugins. You load ontology plugins using the `ontologyResourceEditor` CLI or through the Palantir Dynamic Ontology Manager user interface. For information about deploying an ontology plugin, see Deploying Ontology Plugins on page 49 in this guide.

The `ClassificationChooserFactoryPlugin` is a non-ontology plugin. You deploy this plugin using the `managPlugins` CLI or through the Palantir Enterprise Manager GUI. For information on deploying a standard plugin, see Deploying Non-Ontology Plugins on page 43 in this guide.

Configuring Classifications in a QuickStart Environment

You should develop any custom classification code in the QuickStart environment. The QuickStart environment does not include the Palantir Enterprise Manager. For this reason, the process of configuring classifications in QuickStart is different than it is in a full deployment. Configuration will consist of the following procedures:

- Set the Classification-related System Properties
- Import the Sample Classification Configuration and Rules
- Add Users to the Classification Marking Groups
- Install the ClassifiedInvestigationDefaultsProvider Plugin (Optional)

Set the Classification-related System Properties

You will need to use the utilities found in the QuickStart bin directory to configuration classifications. First start the Palantir Dispatch Server and then do the following:

1. Set the system's PRODUCTION_MODE value to false using the `dist_system_property_util.bat` CLI.

```
C:\Palantir\3.2.1.0\bin>dist_system_property_util.bat -s PRODUCTION_MODE
false
log4j:WARN No appenders could be found for logger
(org.apache.commons.configuration.ConfigurationUtils).
log4j:WARN Please initialize the log4j system properly.
Enter username: admin
Enter password: ********
Connecting to localhost:3280 using SSL
************************************************
Setting PRODUCTION_MODE to false. . .
```

```
**********************************************
The current value of PRODUCTION_MODE is false.
```

2. Set the `ENABLE_CLASSIFICATION_UI` to true.

   ```
   dist_system_property_util.bat -s ENABLE_CLASSIFICATION_UI true
   ```

3. Repeat Step 2 for the remaining system properties if you wish to use them:

Property	Description
`IMPORTER_PERM_EDITOR_POPUP_MODE`	Causes the Data Importer to automatically prompt users to set a classification string. The default OFF prevents the automatic prompting.
`CLASSIFICATION_MAXIMUM_BANNER`	Replaces the reference to this property in the `CLASSIFICATION_BANNER_TEMPLATE` to form a default banner.
`CLASSIFICATION_BANNER_TEMPLATE`	Defines the message appears when an investigation or object component does not have a classification value.
`ENABLE_CLASSIFICATION_TAGGING`	Allows user to select classification markings when tagging documents.
`ENABLE_CLASSIFICATION_COMPARISON`	Determines if the lists of most recently used objects and system favorites includes classification markings that are higher than the document's marking.
`CLASSIFICATION_TAGGING_FAVORITES`	Defines up to ten semicolon-separated classification strings for use in tagging.

4. Restart the Dispatch Server.

   Changing the `PRODUCTION_MODE` requires a restart of the Dispatch Server. The other system properties do not require this.

Import the Sample Classification Configuration and Rules

Make sure the Dispatch Server is still running and do the following.

1. Open a DOS command window in the `INSTALLDIR\bin\windows` directory.
2. Use the `dist_ontology_resources_editor.bat` command to install the sample classification configuration.

   ```
   C:\Palantir\3.2.1.0\bin>dist_text_resource_editor.bat -t
   classificationConfig -m put -f c:\Palantir\3.2.1.0\sampleConfig\
   ClassificationConfig.xml
   Configured logging with file located at: C:\Palantir\3.2.1.0\bin\lib
   \..\..\cli.log.properties Java Version: 1.6.0_20 Sun Microsystems Inc. -
   Java HotSpot(TM) Client VM build 16.3-b01 32-bit
   The user must be in the Ontology Editors group.
   Enter username: admin
   Enter password: ********
   Connecting to localhost:3280 using SSL

   Validation complete.
   ```

```
Successfully saved the classification configuration as an ontology
resource. NOTE: You must restart the server to use the updated
classification configuration.
```

3. Restart the Dispatch Server to put your new configuration into effect
4. In the DOS command window, use the `dist_text_editor.bat` command to install the sample classification rules file.

```
C:\Palantir\3.2.1.0\bin>dist_text_resource_editor.bat -t
classificationConfig -m put -f c:\Palantir\3.2.1.0\sampleConfig\
ClassificationRules.xml
Configured logging with file located at: C:\Palantir\3.2.1.0\bin\lib
\..\..\cli.log.properties Java Version: 1.6.0_20 Sun Microsystems Inc. -
Java HotSpot(TM) Client VM build 16.3-b01 32-bit
The user must be in the Ontology Editors group.
Enter username: admin
Enter password: ********
Connecting to localhost:3280 using SSL

Validation complete.
Successfully saved the classification rules as an ontology resource.
NOTE: You must restart the server to use the updated classification
rules.
```

5. Restart the Dispatch Server to put your new configuration into effect.

Add Users to the Classification Marking Groups

After you create a set of markings, you must add users to the marking groups. To grant access, associate one or more users and/or groups with a marking group. Before you set up this association, you identify the groups or users you want to associate with a particular marking.

Make sure you have started the Dispatch Server and do the following to configure your groups:

1. Open a DOS command window in the `INSTALLDIR\bin\windows` directory.
2. Create classification users using the `dist_create_user.bat` command.

```
C:\Palantir\3.2.1.0\bin>dist_create_user.bat
Configured logging with file located at:
C:\Palantir\3.2.1.0\bin\lib\..\..\cli.log.properties
Java Version: 1.6.0_20 Sun Microsystems Inc. - Java HotSpot(TM) Client VM
build 16.3-b01 32-bit
Enter server host[:port] [localhost]:
Enter username: admin
Enter password: ********
Connecting to localhost:3280 using SSL

Enter user: jphelps
Enter first name: john
Enter last name: phelps
Enter password: **********
Confirm password: **********
Successfully created user jphelps
Add another user [y/n]: n
```

3. Get an idea for your available marking groups by listing the groups with the `dist_list_groups.bat`.

   Marking groups have the following distinctive format:

   `Classification Marking: portionMarking`

4. Add users (or groups) to the classification marking groups with the `dist_group_membership.bat` command.

   You can call the command interactively or you can supply an input file with the list of users and groups you wish to add to them. The following illustrates an example `groupstoadd.txt` input file that adds the user `jphelps` and the group `Building 5` to the MTS marking group:

   ```
   add,Classification Marking: MTS,jphelps
   addGroupToGroup,Classification Marking: MTS,LDAP\Building 5
   ```

   To call the command with this input file to the `dist_group_membership.bat` command you would:

   ```
   C:\Palantir\3.2.1.0\bin>dist_group_membership.bat -f
            C:\Palantir\3.2.1.0\bin\groupstoadd.txt
   Configured logging with file located at:
   C:\Palantir\3.2.1.0\bin\lib\..\..\cli.log.properties
   Java Version: 1.6.0_20 Sun Microsystems Inc. - Java HotSpot(TM) Client VM
   build 16.3-b01 32-bit
   Enter server host[:port] [localhost]:
   Enter username: admin
   Enter password: ********
   Connecting to localhost:3280 using SSL
   Successfully logged in

   Press the Enter key to exit...
   ```

5. Verify the users were added successfully using the `dist_group_membership.bat` command:

   ```
   C:\Palantir\3.2.1.0\bin>dist_group_membership.bat -c list,,jphelps
   Configured logging with file located at:
   C:\Palantir\3.2.1.0\bin\lib\..\..\cli.log.properties
   Java Version: 1.6.0_20 Sun Microsystems Inc. - Java HotSpot(TM) Client VM
   build 16.3-b01 32-bit
   Enter server host[:port] [localhost]:
   Enter username: admin
   Enter password: ********
   Connecting to localhost:3280 using SSL
   Successfully logged in

   Found 6 group(s) for user jphelps:
   Classification Marking: MTS
   Everyone
   Publishers
   Web Users
   Workspace Users
   User: jphelps [system 13]
   Found 6 direct membership group(s) for user jphelps:
   Classification Marking: MTS
   Everyone
   Publishers
   Web Users
   Workspace Users
   ```

```
User: jphelps [system 13]
Press the Enter key to exit...
```

Install the ClassifiedInvestigationDefaultsProvider Plugin (Optional)

Install the `ClassifiedInvestigationDefaultsProvider` plugin. This plugin simplifies the creation of investigation authorization values for users. It is an example of an investigation defaults provider plugin. You can obtain a copy of the plugin from your Forward Deployed Engineer (FDE). Place the plugin JAR in a directory accessible to your QuickStart installation machine.

To install this plugin, make sure you have started the Dispatch Server and do the following to install the plugin:

1. Open a DOS command window in the `INSTALLDIR\bin\windows` directory.
2. Use the manage_plugins.bat command to install the JAR:

```
C:\Palantir\3.2.1.0\bin>dist_manage_plugins.bat -i -f
c:\Palantir\3.2.1.0\bin\classificationinvestigationdefaultsprovider.jar
Configured logging with file located at:
C:\Palantir\3.2.1.0\bin\lib\..\..\cli.log.properties
Java Version: 1.6.0_20 Sun Microsystems Inc. - Java HotSpot(TM) Client VM
build 16.3-b01 32-bit
Enter server host[:port] [localhost]:
Enter username: admin
Enter password: ********
Connecting to localhost:3280 using SSL
Successfully installed plugin
'com.palantir.plugin.ClassificationInvestigationDefaultsProvider-2.0'.
```

3. Confirm the installation by listing the your installed plugins:

```
C:\Palantir\3.2.1.0\bin>dist_manage_plugins.bat -l
Configured logging with file located at:
C:\Palantir\3.2.1.0\bin\lib\..\..\cli.log.properties
Java Version: 1.6.0_20 Sun Microsystems Inc. - Java HotSpot(TM) Client VM
build 16.3-b01 32-bit
Enter server host[:port] [localhost]:
Enter username: admin
Enter password: ********
Connecting to localhost:3280 using SSL
Currently installed plugins (2):
com.palantir.plugin.ClassificationInvestigationDefaultsProvider-2.0
com.palantir.workspace.timeline.TimelineHelperFactory-1.0
```

Writing a Custom Classification Chooser

This section provides an overview of the APIs you need to implement in order to write your own classification chooser. When writing a custom chooser you need to implement the `ClassificationChooserFactoryPlugin` interface. This interface, in turn, implements the `IClasificationChooserFactory`. Your custom chooser should implement the following methods:

```
import java.util.Collection;
import com.palantir.commons.security.ClassificationCategory;
import com.palantir.commons.security.ClassificationChooserFactoryPlugin;
import com.palantir.commons.security.ClassificationMarking;
import com.palantir.commons.security.IClassificationChooser;
import com.palantir.services.auth.Group;
public class CustomClassificationChooser implements
     ClassificationChooserFactoryPlugin {
   public void initialize(Collection<ClassificationMarking> markings,
      Collection<ClassificationCategory> categories) {
     // TODO
   }
   public IClassificationChooser createChooser() {
     // TODO
     return null;
   }
   public IClassificationChooser createChooser(Collection<Group> memberOf) {
     // TODO
     return null;
   }
}
```

Your method implementations should do the following:

`initialize()`	Initializes the chooser with a set of markings.
`createChooser`	An overloaded method to create a chooser. You can create a choose that permits selection of an valid classification string or you can create a chooser whose requirements are met by the specified groups. These methods are defined on the `IClassificationChooser` interface.

Writing a Java-based Rules Engine

This section is an overview of the APIs you need to implement in order to write your own Java-based rules engine.

```
import com.palantir.commons.security.*;
import java.util.Collection;
import java.util.Set;
public class MyClassificationRulesEngine implements
ClassificationRulesEnginePlugin {
   public void initialize(Collection<ClassificationMarking> markings,
      Collection<ClassificationCategory> categories) {
     // TODO
```

```
  }
  public Set<ClassificationMarking> combine(Set<ClassificationMarking>
          markings1, Set<ClassificationMarking> markings2) {
    // TODO
    return null;
  }
  public Set<ClassificationMarking> getDisallowed(
      Set<ClassificationMarking> markings) {
    // TODO
    return null;
  }
  public Set<ClassificationMarking> getImplied(Set<ClassificationMarking>
            markings) {
    // TODO
    return null;
  }
  public Set<Set<ClassificationMarking>> getRequired(
      Set<ClassificationMarking> markings) {
    // TODO
    return null;
  }
  public boolean isValid(Set<ClassificationMarking> markings) {
    // TODO
    return false;
  }

}
```

With the exception of the `initialize()` method, the methods you implement are defined on the `IClassificationRulesEngine`. Your method implementations should do the following:

`initialize()`	Initializes the engine with the given markings and categories.
`combine`	Combines the specified set of classification markings into a single set of classification markings. If one set is empty, but the other one is not, the non-empty set is returned. Classification marking combination forms the basis of banner computation.
`getDisallowed()`	Returns the set of classification markings that are incompatible with those in the specified set (e.g., MOCK UNCLASSIFIED cannot be combined with any MOCK SCI markings). Disallowed markings might be found in the specified collection, as the supplied list of markings might be inconsistent. If this is the case, the contents of the set returned by this method are non-determinant.

`getImplied()`	Returns the set of classification markings that are implied by those in the specified set. Implication exists to support hierarchies (e.g., MOCK TOP SECRET implies MOCK SECRET, CONFIDENTIAL and MOCK UNCLASSIFIED) and disjunctive composition (e.g., MOCK REL TO USA implies MOCK REL TO FVEY). Implication influences the markings that are visible to users and that appear in investigation authorizations.
`getRequired()`	Reports additional markings required by the specified set of markings (e.g., MOCK HCS requires MOCK NOFORN). In some cases, the specified set of markings might require any one of a disjunctive set of markings. To account for that case, this method returns a set of marking sets, as opposed to a set of markings. Each entry in the set returned from this method represents a disjunctive set of markings, any of which will satisfy a requirement of the specified set of markings.
`isValid()`	Returns true if the specified set of markings represent a valid classification. Otherwise, returns false.

Writing a Classification String Handler

This section walks you through the APIs you need to implement in order to write your own classification string handler.

```
import com.palantir.commons.security.*;
import java.util.Collection;
import java.util.Set;
public class MyClassificationStringHandler implements
ClassificationStringHandlerPlugin {
   public void initialize(Collection<ClassificationMarking> markings,
      Collection<ClassificationCategory> categories) {
     // TODO

   }
   public Set<ClassificationMarking> getAllMarkings() {
     // TODO
     return null;
   }
   public Set<ClassificationMarking> parse(String classificationString)
      throws ClassificationStringParserException {
     // TODO
     return null;
   }
   public IClassificationStringParseResult parsePermitPartial(String
classificationString) {
     // TODO
     return null;
   }
   public String make(Set<ClassificationMarking> markings)
      throws ClassificationStringMakerException {
     // TODO
```

```
    return null;
  }
}
```

With the exception of the `initialize()` method, the methods you implement are defined on the `IClassificationStringHandler`. Your method implementations should do the following:

`initialize()`	Initializes the handler with the given markings and categories.
`getAllMarkings()`	Report all markings known to this handler.
`parse()`	Parses a classification string, returning a set corresponding to its constituent markings. Throws an exception if an error occurs during parsing.
`parsePermitPartial()`	Parses a classification string. No exception is throw on error. The first problem encountered is recorded in the result, along with all recognized and unrecognized markings.
`make()`	Converts a set of markings into a classification string. throws an exception if an error occurs while making the string.

10 Working with the Job APIs

This chapter explains how to write custom programs that use Palantir's Job Framework APIs. The following topics are discussed:

- Understanding Job Concepts
- Understanding a Custom Job Implementation
- Walking through a Custom Job Example

Understanding Job Concepts

This section discusses the concepts you need to understand when working with custom jobs:

- Interacting with the Job Server
- Understanding the Job Life Cycle
- Locking via the Job Arguments
- Deploying a Job

The `com.palantir.api.job` package contains all the methods for creating custom jobs. The `PalantirConnection` and `PalantirContext` APIs contain methods for submitting and managing jobs.

Interacting with the Job Server

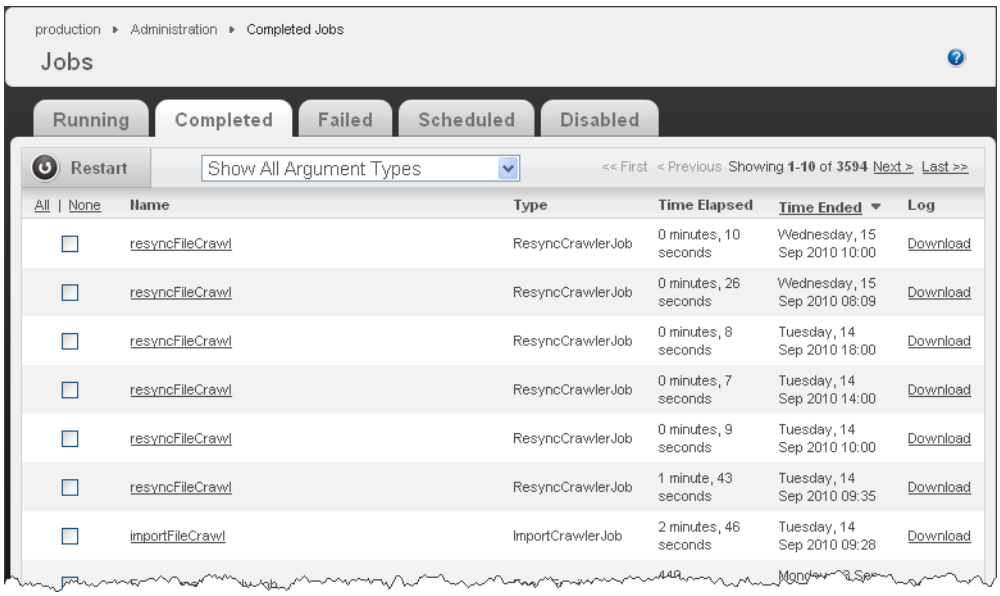
Your installation must include at least one Job Server but might contain a cluster of Job Servers. The Job Server(s) allows for asynchronous and synchronous submission and execution of individual jobs. This means you can submit a job to the server regardless of the server state. Typical jobs that run on the server include:

- large back-end data imports
- recurring import/resync operations
- persistent searches that occur on a given schedule
- group synchronizations with an external LDAP server

All jobs run with a specified frequency — how many times the job is scheduled to run. A job's frequency can be one run or a run every hour 365 days a year. For example, a resynchronization job can run every 12 hours. You specify a frequency using the cron format. Jobs that run a single time have a null frequency.

When you create a custom job, Palantir stores the job description in the back-end database. Then, each time the job runs, whether one time or more frequently, Palantir tracks each execution and creates an entry in the back-end database for that execution. Each `Job` and each of its runs has a `Locator` associated with it. You can use these `Locator`s to retrieve information about the original `Job` submission or an individual execution of that job.

Using the Palantir Enterprise Manager you can manage your custom jobs just as you would the other jobs on the system:



The Enterprise Manager allows you to see the scheduled and running jobs. You can also see the outcome of jobs including which jobs completed and which failed. Using the job APIs you can get these results programmatically as well.

Programmatically, you can query a job's `JobStatus` to get information about a job's execution. If you want your job to execute in the context of the user interface, you can use the `JobStatusMonitor` and `ProgressBarJobStatusMonitor` to display a job's status to a user.

Understanding the Job Life Cycle

A custom job's life cycle involves a client that builds and submits a job and the Job Server that receives and processes the job. The job client must do the following:

- create a `JobArgs` to contain the job parameters
- create a `JobSpec` that specifies the job scheduled
- associate the `JobSpec` with a set of `JobArgs`
- submit the `Job` via a `PalantirConnection` or `PalantirContext`
- monitor the job via the `PalantirClientContext` and `PalantirConnection`
- request the `JobResults` from the Job Server

The Job Server receives the job and then does the following:

- creates an instance of the `Job` subclass for this particular `Job`

- applies the arguments in the `JobArgs` to the `Job`
- invokes the `doJob()` method for the `Job`
- stores the `JobResults`

Locking via the Job Arguments

Jobs can be read-only or they can modify and write changes to objects in an investigation. If your custom job expects to write objects it will need to obtain a lock for the investigative realm it is writing to. If your job does not obtain a realm lock and attempts to write the changes to a repository, it will fail and return an error. Jobs that only read data do not require a lock.

Deploying a Job

The Palantir Platform allows you to run custom code within the Workspace client. You deploy a job as part of a helper plugin. When you do this, use the standard Palantir plugin facilities. If you are writing a job helper for use in the Workspace and within an investigation realm, you should take into account the user interface.

If your job writes data, it must lock the investigative realm in which it runs. If a user has the Workspace open to that investigation, the user should not be modifying the investigation. For this reason, it is a good idea to lock the Workspace windows when running a job in the context of a Workspace client. The job API includes a method for locking the Workspace window when a job is running.

Understanding a Custom Job Implementation

The process of implementing a custom job requires the same thing each time:

- Subclassing Job
- Specifying Job Arguments
- Creating a Job Specification
- Submitting a Job
- Monitoring a Job

Subclassing Job

`Job` is an abstract class that represents a one-time or a recurring task for the Job Server to execute. Your custom job must subclass `Job` and provide implementations for the following methods:

Method	Description
`applyArgs()`	Applies a `JobArgs` to a `Job`. By default, this method does nothing.
`doJob()`	Contains the actual job instructions. This method is analogous to `main()` in a stand-alone program. The code in this method runs on the Job Server each time the job executes. You do not call this method directly; submitting the job causes the system to call this method for you. The system automatically supplies the single `PalantirContext` that this method requires. By default, this method returns null. You must provide an implementation.
`getLogger()`	Gets the logger associated with a `Job` instance. By default this method returns null. You must provide an implementation.

Specifying Job Arguments

`JobArgs` is an abstract class that represents the set of arguments for running the job and for cleaning up after a job completes. The code in the arguments runs on the client and can include the following methods:

Method	Description
`requiresLockGrant()`	Indicates whether or not the job requires a lock on the Data Repository while it is running. The default implementation returns false.

Method	Description
`cleanUp()`	Cleans up after a job's execution. For example, you can use this method to clean up data placed on the Job Server by the job.
`writeDataToJobShare()`	Writes data required for your job's execution to the Job Server's shared disk. For example, if your job processes some files, you can copy the files to the job share using this method.

Creating a Job Specification

The `JobSpec` class in a simple public class that you instantiate within your custom code. You use this class to specify the scheduling information for a job. Through this class you can set and get the following values:

- job name
- user associated with the job
- realm the job runs against
- job frequency
- job status
- memory requirements
- locking requirements

Constructing a `JobSpec` requires you to supply a `JobArgs` instance.

Submitting a Job

You can submit a job for synchronous or asynchronous execution. To submit an synchronous job, call `PalantirConnection.runAndWaitForJob()`. This call submits the job to the Job Server where it is immediately scheduled for execution. For this reason, do not make this call on the AWT or event dispatch thread (EDT). The call returns a `Locator` for the `Job` that you can use with the `PalantirConnection.getJobStatus(Locator)` method.

To submit a job asynchronously, call any of the following methods:

- `PalantirConnection.submitJob()`
- `PalantirConnection.runJob()`
- `PalantirContext.runJobAndLockWindows()`

Caution: If you use `submitJob()` to submit a job that manipulates object state, you must lock the realm. If you do not obtain a lock, the job will encounter errors when running.

The `submitJob()` method returns a `JobStatus` instance. The other two methods return a class modeled after Java's `Future` class, the `PollableJobFuture` class. This class allows the calling thread to interact with the `Job` on an extended basis. The following methods are available on this future class:

Method	Description
`get()`	Gets the `JobResults` object for the job. This is a blocking call and will wait if necessary for the job to complete before retrieving the result.
`getLatestStatus()`	Gets the last reported `JobStatus` for this job.
`getPercentComplete()`	Gets an integer representing the percentage complete for this job.

When running asynchronous jobs, be mindful of the locking capabilities of the three methods. Typically, you should use `PalantirContext.runJobAndLockWindows()`. If the current `PalantirContext` is the Workspace client and the job requires a realm lock, this method will lock all the windows in the Workspace for the job's duration. The `runAndWaitForJob()` method does not lock the Workspace for you.

Monitoring a Job

You implement the `JobStatusMonitor` interface to monitor the status of your job. This interface has a single method for you to implement:

```
public void statusUpdate(JobStatus status);
```

The system passes a `JobStatusMonitor` when a `Job` instance is submitted for execution. Each instance has its own monitor or set of monitors — `runJob` takes `varargs` of `JobStatusMonitor`. The system periodically updates a job's `JobStatus` object. You can query this `JobStatus` to get a job's current status.

A `ProgressBarJobStatusMonitor` is a simple implementation of the `JobStatusMonitor`. A `MultiStageProgressMonitor.Stage` instance is returned by this implementation. Multiple `JobStatusMonitor` instances can be rolled into a single `JobStatusMonitor` using the `JobStatusMonitorMultiplexer` class. This class is itself an instance of `JobStatusMonitor`.

Walking through a Custom Job Example

The example in this section illustrates how to write a `HelperFactory` plugin to run a job immediately, a single time. You can download a ZIP package containing the project code for this example.

Implementing JobArgs

The following is the simplest of `JobArgs` implementation. This example implements a very basic constructor that ensures the user supplied the appropriate arguments to the job (an object label and a job duration):

```
package com.palantir.plugin.helper.userjobs;
...
public class UserJobsTestJobArgs extends JobArgs {
    private static final long serialVersionUID = 1L;

    final Integer secondsToRun;
    final String objectLabel;
    public UserJobsTestJobArgs(String objectLabel, Integer secondsToRun) {
      super(UserJobsTestJob.class);

      Validate.notNull(objectLabel,"Must supply an object label");
      Validate.notNull(secondsToRun,"Must supply a duration for the job to
          run.");

      if(secondsToRun.intValue() < 5) {
        throw new IllegalArgumentException("Must run for at least five
        seconds, " + secondsToRun + " is too short.");
      }
      this.secondsToRun = secondsToRun;
      this.objectLabel = objectLabel;
    }
```

Because the `UserJobsTestJob` will be writing objects, it requires a lock on the realm.

```
    @Override
    public boolean requiresLockGrant() {
      return true;
    }
    Integer getSecondsToRun() {
      return secondsToRun;
    }
    String getObjectLabel() {
      return objectLabel;
    }

}
```

Implementing Job

The following imports include supporting APIs for the `doJob()` method which all jobs must implement.

```
package com.palantir.plugin.helper.userjobs;
...
public class UserJobsTestJob extends Job {
    static final Logger log = LogManager.getLogger(UserJobsTestJob.class);

    Integer secondsToRun = null;
    String objectLabel = null;
```

A custom `Job` must implement a `getLogger()` method which, by default, returns null. The example implementation depends on the Apache Log4J package to supply the logging mechanism:

```
@Override
public Logger getLogger() {
  return log;
}
```

The following code applies a set of arguments to this `UserJobsTestJob` instance.

```
@Override
public void applyArgs(JobArgs args) throws BatchException {
  if (args instanceof UserJobsTestJobArgs) {
    UserJobsTestJobArgs userArgs = (UserJobsTestJobArgs) args;
    this.secondsToRun = userArgs.getSecondsToRun();
    this.objectLabel = userArgs.getObjectLabel();
  }
}
```

The `doJob()` method is the "main" of every job and receives a single argument — the context in which the job runs. The job in this example is simply outputting a single object.

```
@Override
  public JobResults doJob(PalantirContext ctx) {
    try {
      PalantirConnection conn = ctx.getPalantirConnection();

      int steps = 1 + secondsToRun;
      float progressStep = 100f / (float)steps;

      // build the object
      setStatusMessage("Creating object with label " + objectLabel);
      PTObjectContainer ptoc =
                  PTObjectContainerFactory.createBlankObject(conn);

      //create the data source records
      DataSourceRecord dsr =
                  conn.getDsrFactory().createManuallyEnteredDsr();
      Collection<DataSourceRecord> dsrs = Collections.singleton(dsr);

      ptoc.setTitle(objectLabel, conn, dsrs, SETTER_STYLE.KEEP_OTHERS,
                Property.MANUALLY_SET_PRIORITY);
      ptoc.setType(PTObjectType.ENTITY);
      setStatusMessage("Storing object: " + ptoc.toCompactString());

    conn.objectStore(Collections.singleton(ptoc),
      PalantirDataEventType.DATA);
      conn.objectStore(Collections.singleton(ptoc),
      PalantirDataEventType.DATA);
      setPercentComplete(getPercentComplete() + progressStep);
      // now sleep to simulate real work
      for(int i = 0; i < secondsToRun; i++) {
        setStatusMessage("Starting sleep phase" + i);
        Thread.sleep(1000);
        setStatusMessage("Woke up after sleep phase " + i);
        setPercentComplete(getPercentComplete() + progressStep);
      }

      setStatusMessage("Done!");
      setPercentComplete(100f);
```

```
      results.setReturnStatus(ReturnStatus.SUCCESS);
    } catch (InterruptedException e) {
      results.setReturnStatus(ReturnStatus.FAILURE);
    }

    return results;

  }
```

Putting Together the Job's Plugin

`UserJobsTestingHelperFactory` code for constructing the helper and submitting job:

```
package com.palantir.plugin.helper.userjobs;
...
public class UserJobsTestingHelperFactory extends AbstractHelperFactory {
   static final Logger log =
LogManager.getLogger(UserJobsTestingHelperFactory.class);

   public UserJobsTestingHelperFactory() {
     super("User Jobs API Testing Helper",
        new String[] { GraphApplicationInterface.APPLICATION_URI },
        new Integer [] { SwingConstants.VERTICAL },
        new Dimension(330,500),
        null,
        UserJobsTestingHelper.class.getCanonicalName());
   }
```

Creates an instance of the helper supplying the helper's Palantir context and the application that it runs in:

```
   public HelperInterface createHelper(PalantirContext palantirContext,
     ApplicationInterface application) {
     return new UserJobsTestingHelper(this,palantirContext);
   }

   protected static class UserJobsTestingHelper implements HelperInterface {

     final HelperFactory factory;
     final PalantirContext ctx;
     JPanel panel = null;

     JTextField labelField = null;
     JTextField durationField = null;
     Stage progress = null;
     JLabel statusLabel = null;

     public UserJobsTestingHelper(HelperFactory factory,PalantirContext ctx)
     {
       this.factory = factory;
       this.ctx = ctx;
       buildGUI();
     }

     private void buildGUI() {
       this.panel = new JPanel(TableLayouts.create("p,p,p,p",
                    "p,p,p,p,p,p", 8, 8));
```

```
this.panel.setBorder(BorderFactory.createEmptyBorder(4, 4, 4, 4));

JLabel labelLabel = new JLabel("Object Label:");
this.panel.add(labelLabel, "0,0,r,c");
labelField = new JTextField("llama");
this.panel.add(labelField, "1,0");
// message
JLabel durationLabel = new JLabel("Number of seconds to run:");
this.panel.add(durationLabel, "0,1,r,c");
durationField = new JTextField("10");
this.panel.add(durationField, "1,1");
```

This section illustrates the creation of a progress bar so that you can display the progress of the job to the user while the windows are locked.

```
final MultiStageProgressBar bar = new MultiStageProgressBar();
progress = bar.addStage("Backend Job", 0.1f);
this.panel.add(bar.getProgressBar(),"0,3,c,c");

// error messages
statusLabel = new JLabel("");
statusLabel.setForeground(Color.RED);
this.panel.add(statusLabel,"0,4,1,4,c,c");
```

In this section, the code submits the job for processing:

```
// action button
JButton sendButton = new JButton("Send messages");
sendButton.setBackground(Color.CYAN);
sendButton.setActionCommand("FIRE");
sendButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    if("FIRE".equals(ae.getActionCommand())) {
      NotAwt.Utilities.runAsynchronous(new Runnable() {
        public void run() {
          try {
            runAndWaitForJob();
          }catch (Exception e) {
            log.error("Error running job",e);
            statusLabel.setText(e.getMessage() +
                        " -- see log for details");
          }
        }
      });
    }
  }
});
this.panel.add(sendButton,"0,5,1,5");
}
```

In this section, the code puts the pieces of the job together, associating the `JobSpec` with the `JobArgs`, and setting the status.

```
private void runAndWaitForJob() throws Exception {
  statusLabel.setText("Preparing job");

  String label = labelField.getText();
  Integer duration = Integer.parseInt(durationField.getText());
  JobArgs args = new UserJobsTestJobArgs(label,duration);
  JobSpec spec = new JobSpec(ctx.getPalantirConnection(), "MyJob",
```

```
                    args, ctx.getUser(), ctx.getBaseRealm(), false);

      ProgressBarJobStatusMonitor statusMon = new
            ProgressBarJobStatusMonitor(progress);
      JobStatusMonitor updateStatus = new JobStatusMonitor() {
        public void statusUpdate(JobStatus status) {
          statusLabel.setText(status.getStatusMessage());
        }
      };
```

The status monitor will update the progress bar:

```
      final PollableJobFuture jobFuture =ctx.runJobAndLockWindows(spec,
          statusMon,updateStatus);
      statusLabel.setText("Job Submitted");
      JobResults results = jobFuture.get();
      if(ReturnStatus.SUCCESS.equals(results.getReturnStatus())) {
        this.statusLabel.setText("Job finished successfully");
      } else {
        this.statusLabel.setText("Job failed.");
      }
    }

    public void dispose(ApplicationInterface application) {
      // unregister any listeners
    }
    public String getDefaultPosition() {
      return BorderLayout.EAST;
    }
    public JComponent getDisplayComponent() {
      return this.panel;
    }
    public HelperFactory getFactory() {
      return this.factory;
    }
    public String getTitle() {
      return "User Job Testing";
    }
    public void initialize(ApplicationInterface application) {
      // register any listeners

    }
  ..
  }
}
```

11 Extending the IProperty APIs

This chapter introduces you to APIs you can use to customize the storage and presentation of object properties in the Palantir Platform. The following topics are included:

- Overview of the Property APIs
- Customizing Property Validation
- Customizing Property Display Logic
- Customizing Property Parsing
- Customizing Fuzzy Search-Property Approximation

The full Javadoc for these APIs are available from the Palantir Devzone Wiki.

Overview of the Property APIs

During data import or as users add properties through a client, the Palantir Platform ensures each property data is both valid and formatted correctly. The system takes raw String data and constructs `Property` objects. This section discusses how the Palantir Platform ensures that property data is both valid and formatted correctly for your environment. This section also discusses how you can use the Palantir property APIs to customize data validation and formatting.

Before customizing property types, you should make sure you are familiar with the information in the Palantir Dynamic Ontology Manager Guide. In particular, make sure you read through Changes Allowed in a Deployed Ontology in Chapter 11.

How the Platform Supports Properties

Each Palantir object (`PTObject`) has one or more properties. Recall from Creating Components (Properties, Notes, Media, and Links) on page 74 that properties can be either simple (age) or composite (address). Composite properties are any combination of two or more simple base types. Address is defined as a composite — each of its component parts (`PropertyTypeComponent`) is defined with a String type. The `Property` class represents both simple and composite properties within the Palantir Platform. Your extensions will work with instances of this class.

Note: For specific information about Date formats, see Chapter 13, DatePrimitiveMaker: Extending the Ontology's Date Parsing.

The Palantir Platform uses the following types of actions on properties:

validation	Validators ensure that a property value is the correct data type and is formatted properly. If validation fails, you can alert users to correct their input.
display formatting	Formatters display properties in different ways based on the data in any given property.
parsing	Makers determine which portions of the validated input string refer to which components in a composite property.
comparison	Approxes support the use of validated, parsed-property data in comparisons with the data in other similar properties. Approxes define the degree of matching, from exact match (most specific) to various degrees of fuzzy matching.

Administrators can use the **Property Editor** in the Dynamic Ontology Manager to create new, custom properties. These properties have a defined base type. The base types have default property validators, display formatters, parsers, and approxes. Administrators can change or add to the defaults by writing customized actions for validation, formatting, parsing, and approxes.

If you have not already done so, familiarize yourself with the Dynamic Ontology Manager interface. This will give you an idea of the existing capabilities of validators, formatters, makers, and approxes. The *Palantir Dynamic Ontology Manager Guide* describes how to use the interface.

The following table lists the default validators, formatters, parsers, and approxes provided by the Dynamic Ontology Manager:

Table 29 Support for the Default Property Actions

Actor	Supported Actions
validator	● Set Value (list of valid values) ● Regular Expression (Java regex pattern) ● Enumeration ● Integer ● IP Address ● Length (min-max) of numerics ● Number ● Numerical Range (min-max)
formatter	● {Component_name} {Comp_name,optional_format_option}... to display composite properties in various formats. ● {VALUE} to display the entire data value of a simple property.

Table 29 Support for the Default Property Actions

Actor	Supported Actions
maker	• Composite (default for composite properties that are made up of two or more simple components). • Regular Expression (Java regex pattern). • Simple (default for String, Number, Date, and Enumeration base types). • Key Value Matcher (maps keys to a particular value when there are several representations of the value).
approx	• Token (default): Determines the level of matching properties. • Custom Address: Performs Address Verification System (AVS) matching based on the first numeric from an address field and the ZIP code. • Duration: Converts units of time to the same unit for matching. • Filename: Removes slashes from file paths for matching. • Filter Function: Filters properties based on specified substrings of component values. • Metaphone: Performs matching based on pronunciation, or similar sounding names or words. • Regular Expression: Performs matching based on Java 1.5 regex patterns.

Understanding the Property Action APIs

If the Dynamic Ontology Manager does not support all the properties needed in your environment, developers can extend the baseline property functionality by implementing the following APIs:

`IPropertyValidator`	Creates custom validators to process the data in a property in a specific way. See Customizing Property Validation on page 141.
`IPropertyFormatter`	Constructs a human-readable string from the data in a property. See Customizing Property Display Logic on page 138.
`IPropertyMaker`	Create a custom parser to process property instances. See Customizing Property Parsing on page 132.
`IPropertyApproxGenerator`	Normalizes data in a property data map to enable fuzzy-match searches. See Customizing Fuzzy Search-Property Approximation on page 143.

These interfaces all extend both Java `Serializable` interface and the Palantir interface `ConfigurableAdminComponent`. They are called by the Palantir Platform when a user or client process such as data import creates or updates an individual property.

When you create (or update) a property type, the Dynamic Ontology Manager stores the type's configuration. For example, a property of type Address might include a maker that requires a country code argument. Palantir provides an API for configuring property types.

The `ConfigurableAdminComponent` interface defines the possible configuration arguments available for creating and editing custom validators, formatters, makers, and approxes. Key methods of this interface include:

Method	Description
`getArgumentValues()`	Returns the `ConfigurableAdminComponent`'s argument values. Cannot return null; return an empty array instead.
`getPossibleArgs()`	Returns possible arguments for this `ConfigurableAdminComponent` instance as an array of Strings. Cannot return null; return an empty array instead.
`setArgs(Arguments` *`args`*`)`	Sets the arguments passed into the `ConfigurableAdminComponent`.

The configuration of a property is stored in the ontology file (ONT). The `ConfigurableAdminComponent` interface has a subclass, `AdminComponentUtils` that contains two convenience methods for constructing the XML configuration for a property. These convenience methods are:

```
getXMLTag(PropertyComponentType type, String uri, Arguments
arguments)
```

```
getXMLTag(PropertyComponentType type, String uri, Arguments
arguments, String menuText)
```

The first method is used for validators, makers, and formatters. The second method is for approxes as it takes the menu text for the approxer.

Customizing Property Parsing

The `IPropertyMaker` interface provides an extensibility point to integrate your own data parsing code to generate `Property` objects. The transformation from the input value to a `Property` is accomplished with the `make` method. The input value passed to the `make` method is pre-trimmed of whites space before and after.

Like all Property extensibility points, configuration parameters are specified using the `ConfigurableAdminComponent` interface.

Method	Description
`getName()`	Returns the display name of this property maker.
`getUri()`	Returns the URI of this property maker.
`getXMLTag()`	Called by the Dynamic Ontology Manager whenever you create a or edit an existing property type. Use the `AdminComponentUtils.getXMLTag` method to obtain an `IPropertyMaker` configuration.
`make(Object o LocatorFactory lf)`	Makes a property of the given type with the given input.
`make(Object o, LocatorFactory lf, boolean useDefaultComponentValues)`	Makes a property of the given type with the given input, using the component's default values.
`setType(PropertyType type)`	Sets the `PropertyType` for this maker.

For information on writing a maker that returns an error to the Palantir Workspace, see Example of a Maker that Returns a Message to the User on page 137.

Example of an IPropertyMaker Implementation

The Palantir base ontology includes the `com.palantir.property.height` property. This base property is a composite property that includes a height value and a units value. Your deployment might want to simplify the height entry to take, for example, a value using the U.S. feet and inches notation such as 5‘ 4‘‘.

In this section, you replace the base height property in your ontology with a custom height. The process for doing this includes three different tasks:

- Removing the Base Height and Adding a New Height
- Developing the Code for Your New Maker
- Add the Custom Maker to Your Ontology

This section also gives you An Example Maker of a Composite Property Maker on page 137.

Removing the Base Height and Adding a New Height

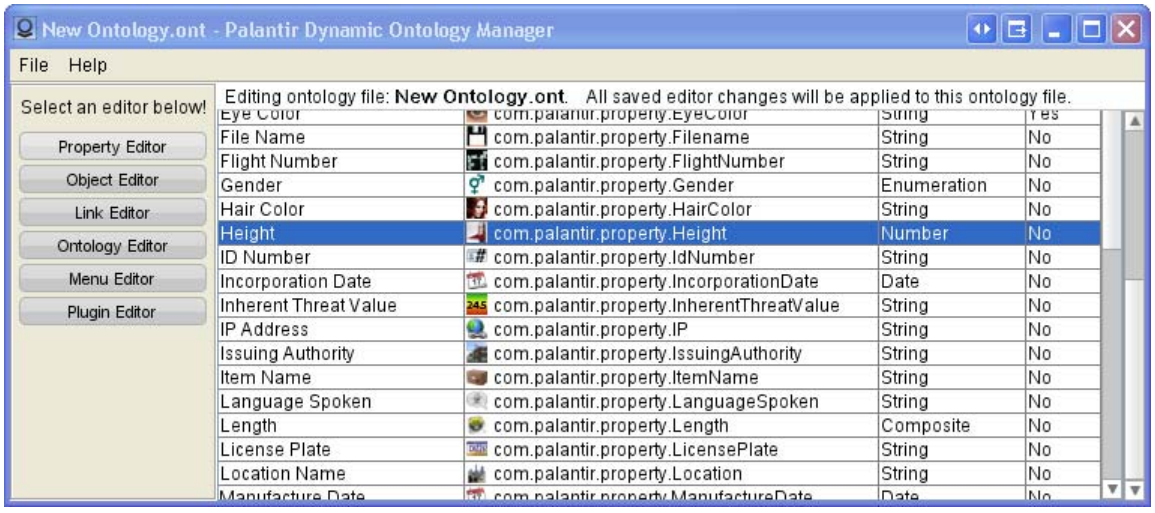
This example requires you to replace the base height property with your own custom property. To do this, you must first start Palantir Dynamic Ontology Manager and download your existing ontology. Then, delete the property by doing the following:

1. Open the **Property Editor** page.
2. Locate the Height property.

3. Select the property and click **Delete Property Type**.
   The system prompts you to confirm your action.
4. Verify that the property was deleted.

Now, add a new **Height** property by doing the following:

1. Click **Add Property Type**.
2. Enter the following information:
   - **Display name** — Height
   - **URI** — com.palantir.property.height
   - **Base type** — Number
3. Click **Choose existing icon** for the **Details Icon**.
   The icon chooser appears.
4. Select the base Height icon and click **Use Selected Icon**.
   The system prompts you to derive an edge icon.
5. Choose **No**.
6. Click **Choose existing icon** for the **Edge Icon**.
7. Scroll to the base Height edge icon and click **Use Selected Icon**.
8. Click **Next** four time to move to the **Approxes** page.
9. Click **Save Property Type** to save your new property.
10. Verify your new **Height** icon appears in the ontology.



11. Save your updated ontology but do not update the servers.

Developing the Code for Your New Maker

The example code consists of two classes. The `ParseTools` class is a utility class with various parsing methods. The `HeightMaker` class implements the `IPropertyMaker` interface. You can download a ZIP package containing the project code for this example.

The sample code creates a custom height parser by using `HeightMaker.make(Object o, LocatorFactory lf)` method.

```
...
public class HeightMaker implements IPropertyMaker {
    protected static final long serialVersionUID = 1L;
    ...
    protected PropertyType type;
    ...
    public Property make(Object o, LocatorFactory lf)
      throws PropertyMakerException {
        if (o instanceof String) {
          String s = (String) o;
            // try to parse out inches:
            return Property.getInstance(lf, type, parseInches(s), Role.NONE);
      }
        return null;
    }
```

Recall that you created `Height` as a simple String property. The Palantir Platform passes a String to the `HeightMaker`. The system strips the white spaces from the beginning and end of the String. The `make` method creates the property instance passing the property's type and the output of the `HeightMaker.parseInches()` method.

```
...
public String[] parseInches(String input) {
     // The system pre-trims (no whitespace before or after) data
     // passed to the make method. This method receives pre-trimmed
     // data from make.
     Pattern myPattern;
     Matcher myMatcher;
     String lcaseIn=ParseTools.prepInput(input);

     // Try feet and inches. Feet appears in fifty so it needs white space.

myPattern=Pattern.compile("(.*)(feet|foot|\\sft\\.?|ft\\.?\\s|')\\s*(.*)");
     myMatcher=myPattern.matcher(lcaseIn);
     if (myMatcher.matches()) {
       return ParseTools.parseFeetAndInches(myMatcher);
     }

     // The input was just inches
     myPattern=Pattern.compile("(.*)(in\\.?|inch|inches|\")");
     myMatcher=myPattern.matcher(lcaseIn);
     if (myMatcher.matches()) {
       return new String[] {
         ParseTools.ParseMultiDigitNumber(myMatcher.group(1)).toString() };
     }

     // Entry was in Meters
     myPattern=Pattern.compile("(.*)(m|meter)s?\\.?");
     myMatcher=myPattern.matcher(lcaseIn);
     if (myMatcher.matches()) {
     return ParseTools.parseMetricLength(myMatcher);
```

```
  }

  // Try to parse out a number; assume that < 10 means feet;
  // otherwise, inches
 // note that this covers the case where the input is "60 inches"
  myPattern=Pattern.compile(ParseTools.NumberInWordsPattern);
  myMatcher=myPattern.matcher(lcaseIn);
  if (myMatcher.matches()) {
    Float ret=ParseTools.ParseMultiDigitNumber(myMatcher);
    if (defaultToHumanSize) {
      if (ret<10) // assume feet
        ret*=12;
      else if (ret>100) { // assume cop style: 602 = 6'2"
        float ft=(float) Math.floor(ret/100);
        float in=ret-ft*100;
        ret = 12*ft + in;
      }
    }
       return new String[] { ret.toString() };
  }

  return new String[0];
  // return null; // renders "property incomplete"
}
```

When you have completed your work, compile the project into a PAR file format. For a walkthrough of how to build your code, see Chapter 2, Making Your First Custom Code Project.

Add the Custom Maker to Your Ontology

In this section, you add the maker to your ontology. If you have not already done so, start the Dynamic Ontology Editor and do the following:

1. In the Plugin Editor page of the Dynamic Ontology Manager, click **Add Plugin**.

   An **Open** dialog appears.

2. In the **Open** dialog, choose the PAR file containing the plugin you want to add.

3. Click **Open**.

   The Dynamic Ontology Manager adds the PAR file to your ontology and saves the change to the ontology.

At this point, you can make use of the `HeightMaker` plugin in your ontology:

1. Open the Property Editor page.

2. Select the Height property and click **Edit Property Type**.

   The **Basic Information** page appears.

3. Click the **Next** button until the **Parsers** page appears.

4. Click **Add New**.

   The **Parser Editor** dialog appears.

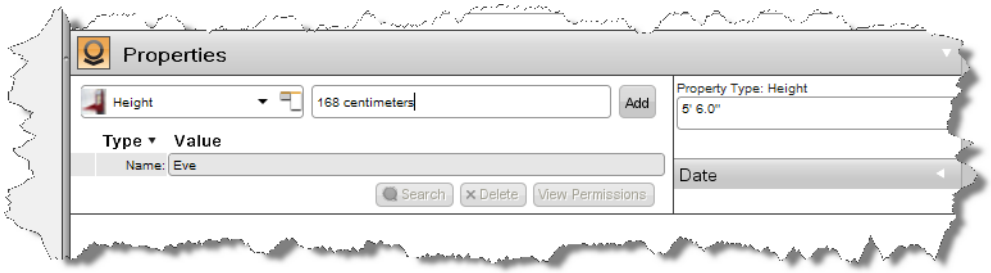
5. Select your custom maker from the **Parser Type** dropdown.

   If you used the example project, you will find **Height (Inches) Maker** on the list.

6. Click **Save** to close the dialog.

7. Click **Next**.
8. Click **Save Property Type** to save your changes.
9. Save your ontology and export it to the Dispatch Server.
10. Start the Dispatch Server.

Launch the Palantir Workspace and test your maker by adding a height property to an entity.



An Example Maker of a Composite Property Maker

The `HeightMaker` is a sample of a simple property maker. Composite properties require you to populate individual components of the property. For example, you might write a `PhoneNumberMaker` that parses a `String` to create the following complex property:

- COUNTRY_CODE
- AREA_CODE
- PHONE_NUMBER

Given an input `String` of `+1-650-555-1212` or `16505551212`, the `PhoneNumberMaker` might segment the value as follows:

- COUNTRY_CODE: 1
- AREA_CODE: 650
- PHONE_NUMBER: 5551212

The code would be responsible for parsing all possible input formats. If you are interested in seeing an example of a composite property maker that does this work, you can download a ZIP package containing the project code for a `CustomAddressMaker`.

Example of a Maker that Returns a Message to the User

You might want your maker to pass a message back to the user. Palantir provides the `RegexMaker` class which implements `IPropertyMaker`. You can override the `RegexMaker.isValidHook()` method to present a message to the user. The method's signature is the following:

```
protected boolean isValidHook(PropertyType propType,
                    Map<String, String> data) throws PropertyMakerException
```

This method returns whether the data map is a valid property. Since it is possible for properties to be created in an invalid state, your method should return true. Returning true allows the validation code to deal with the validation of the actual property. If only valid properties should be parsed, you can override this method to check the validity of the property's data map. Throw a `PropertyMakerUserMessageException` to pass up to the GUI the reason for the parse failure.

This example has three classes `AddressMakerWithMessage`, `CaliforniaZipCityReader`, and `ZipCityAreaCode`. The maker overrides `isValidHook()`:

```
protected boolean isValidHook(PropertyType propType, Map<String, String>
    data) throws PropertyMakerException {
      String city = data.get(CITY);
      String state = data.get(STATE);
      String zip = data.get(ZIP);

      if ("CA".equalsIgnoreCase(state) && city != null && zip != null) {
        // verify the california city and zip
        ZipCityAreaCode result = getReader().getZipCityAreaCode(zip);
        if (result == null) {
          throw new PropertyMakerUserMessageException("Zip code '"+zip+"' is
                 not valid in CA.", propType);
        }
        if (!result.getCity().equalsIgnoreCase(city)) {
          throw new PropertyMakerUserMessageException("Zip code '"+zip+"' does
               not match expected city '"+city+"'.", propType);
        }
      }
      return true;
    }
```

You can download a ZIP package containing the project code for the `AddressMakerWithMessage` project.

Customizing Property Display Logic

Your makers and formatters should work together. When a user or a system process creates a new value or stores a changed value, the maker runs. It ensures the data stores correctly in the underlying database. The formatter takes the same value and determines how the data displays to a user in a Palantir user interface.

Note: When you first install and use a custom display formatter in your ontology, you will need to reindex the search servers.

You will want to create a formatter when there is a difference between how you want to store the data and how you want to display it. For example, you might store data in form that is computer-friendly but display the data in a more user-friendly format. From the stored state to the display your formatter should stay true to the property's value as stored by the maker.

The `IPropertyFormatter` interface provides an extensibility point to integrate your own display logic on a `Property`. The transformation from `Property` to display string is accomplished with the `format(Property p)` method.

Method	Description
`format(Property p)`	Runs the formatter against the given property.

Method	Description
`getName()`	Returns the display name of this property formatter.
`getUri()`	Returns the URI of this property formatter.
`getXMLTag()`	Called by the Dynamic Ontology Manager whenever you create a or edit an existing property type. Use the `AdminComponentUtils.getXMLTag` method to obtain an `IPropertyFormatter` configuration.

Palantir provides a simple `TokenFormatter` class that implements `IPropertyFormatter`. Your plugin can subclass `TokenFormatter`. The `ptplugin.xml` file will show the `component` interface as `IPropertyFormatter`.

Example of an HeightFormatter Implementation

Height is a good example of a property that you might want to store differently than you display it. For example, in the United States most people see height in terms of feet and inches such as 5’ 4”. However, storing that exact value would mean you could not use the stored value to calculate, say, BMI without first translating it to a number to use in the calculation.

The best design for a height property is to create a maker that stores the value as a number (Example of an IPropertyMaker Implementation on page 133) and write a formatter that translates that number for users to see. In this way, you could display the value in feet and inches height for American users and in a different format for other nationalities. The example in this section illustrates a formatter that displays in a format comfortable for American users.

Developing the Code for Your New HeightFormatter

The following example creates a custom height formatter by using `HeightFormatter.processToken(String` *`token`*). This method takes the string and uses the Java Pattern and Matcher classes to return a formatted height in feet and inches. You can download a ZIP package containing the project code for this example.

The `processToken()` method appears in the code as follows:

```
protected String processToken(String token) {
    Pattern myPattern=Pattern.compile("^(~)?(\\d*(\\.\\d*))?$");
    Matcher myMatcher=myPattern.matcher(token);
    String ret=token;
    if (myMatcher.matches()) { // this is a numeric
      // debugMatcher(myMatcher);
      if (myMatcher.group(1)!=null)
        ret="~";
      else
        ret="";
      Float inches=new Float(myMatcher.group(2));
      Integer feet=(int)(inches/12);
      inches%=12f; // truncating is now done in the maker
    ret+=feet.toString().concat("' ").concat(inches.toString()).concat("\"");
    }
```

```
        return ret;
    }
```

Adding the Display Formatter to the Height Property

In this section, you add the formatter to your ontology. If you have not already done so, start the Dynamic Ontology Editor and do the following:

1. In the Plugin Editor page of the Dynamic Ontology Manager, click **Add Plugin**.

   An **Open** dialog appears.
2. In the **Open** dialog, choose the PAR file containing the plugin you want to add.
3. Click **Open**.

   The Dynamic Ontology Manager adds the PAR file to your ontology and saves the change to the ontology.

At this point, you can make use of the `HeightFormatter` plugin in your ontology:

1. Open the Property Editor page.
2. Select the Height property and click **Edit Property Type**.

   The **Basic Information** page appears
3. Click the **Next** button two times until the **Display Formatter** page appears.
4. Click the **Replace with custom type** link.

   The **Formatter** dialog appears.
5. Select your custom maker from the **Formatter Type** dropdown.

   If you used the example project, you will find **Height Formatter** on the list.
6. Enter the `{VALUE}` in the **Token Pattern** field.
7. Click **Save** to close the dialog.

   The system informs you that changing this property requests reindexing the search server.
8. Click **OK**.
9. Click **Next** twice to move to the last page, **Approxes**.
10. Click **Save Property Type** to save your changes.
11. Save your ontology and export it to the database.

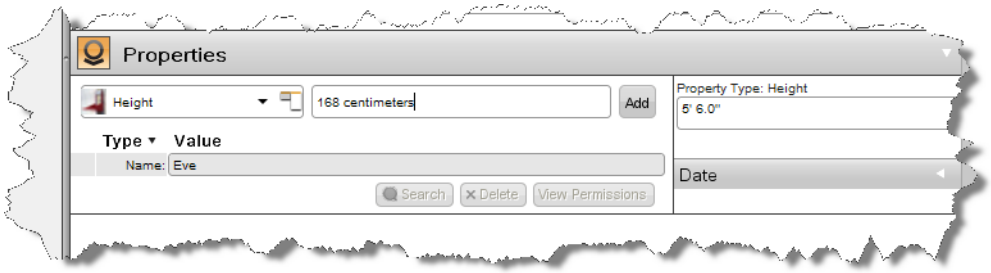
If you are running in the QuickStart environment, you reindex the Search and Horizon Servers with the `dist_dispatch_reindex.bat` command. If you are in a staging or production environment, you use the `dispatchReindex` command. To reindex and test your plugin, do the following.

1. Ensure that you have stopped the Dispatch Server, Search Server, and Horizon Servers.
2. Move or delete the current search indexes on both the Search Server and the Horizon Server.

   Each server has an `indexes` subdirectory in their installation directory.
3. Restart the Search and Horizon Servers.

4. Log into the machine where the Dispatch Server is installed and run the reindex process command.
5. Restart the Dispatch, Search, and Horizon Servers.

Launch the Palantir Workspace and test your formatter by adding a height property to an entity.



Example of a Custom Address Formatter

The following example creates a custom address formatter by using `CustomAddressFormatter.format(Property prop)`. The custom formatter iterates over the `PropertyTypeComponent` instances, formats the output as *`DISPLAYNAME`* : *`URI`*, and returns the formatted string.

- COUNTRY_CODE: 1
- AREA_CODE: 650
- PHONE_NUMBER: 5551212

The formatter converts the components into the display string `+1-650-555-1212`. You can download a ZIP package containing the project code for this example.

Customizing Property Validation

The `IPropertyValidator` interface provides an extensibility point to integrate your own data validation for `Property` objects. The validation of a `Property` or input String is accomplished with the `validate(Object o)` method. For example, you might write a `PhoneNumberValidator` that parses a String or `Property` to accept valid phone numbers. Given an input String of `+1-650-555-1212` or `16505551212`, the validator would determine that both are valid inputs.

When you are working with a simple property, the system passes this method a `Property` object. If the property has one or more components, you must apply the validator to an individual component. For components, the system passes a `String` object to the `validate()` method.

Like all `Property` extensibility points, configuration parameters are specified using the `ConfigurableAdminComponent` interface. For this interface, you implement the following:

Method	Description
`getName()`	Returns the display name of this property validator.
`getUri()`	Returns the URI of this property validator.
`getXMLTag()`	Called by the Dynamic Ontology Manager whenever you create a or edit an existing property type. Use the `AdminComponentUtils.getXMLTag` method to obtain an `IPropertyValidator` configuration.
`validate(Object o)`	Validates the given argument against this property validator.

Example of an IPropertyValidator Implementation

The default `Address` property has a `ZIP` component that represents a U.S. zip code of 5 characters. This example creates a custom validator to verify that the ZIP component is present using the `CustomZipValidator.validate` (`Object` *`o`*) method.

```
public String validate(Object o) {
    String msg = "";

    if(o instanceof String){
     // Check for an empty zip code
     if (((String) o).trim().length() <= 4)
      msg = "The zip is too short.";
    }
    return msg;
  }
```

You can download a ZIP package contain the project code for this example.

To add this validator to the Address property, you would start the Dynamic Ontology manager and do the following:

1. Add the plugin through the **Plugin Editor** panel.
2. Go to the **Property Editor** panel.
3. Select the **Address** property.
4. Click **Next**.

   The **Components** page appears.
5. Select the **ZIP** component.
6. In the **Select Details** panel, click **Add New** from the **Validators** section.

   The **Edit Validator** page appears.
7. Select **CustomAddressValidator** from the **Validator Type** dropdown
8. Click **Save**.
9. Click **Next** until you reach the **Approxes** page and press **Save Property Type** to save your changes.

10. Export your new ontology to the database.
11. Restart the Dispatch, Search, and Horizon Servers.

Launch the Palantir Workspace and test your validator by adding a ZIP property to an entity

Customizing Fuzzy Search-Property Approximation

The `IPropertyApproxGenerator` interface provides an extensibility point to integrate your own search capabilities on a `Property`. `IPropertyApproxGenerator` is typically used to create equivalence classes' `Property` objects. The transformation from the `Property` data to the `String` approximation is accomplished with the `getApproximation(Property p, boolean isApproxZero)` method.

Note: You can only add an approx when you are creating a new property from scratch or cloning an existing property.

Like all `Property` extensibility points, configuration parameters are specified using the `ConfigurableAdminComponent` interface. For this interface, you implement the following:

Method	Description
`getApproximation(Property prop, boolean isApproxZero)`	Returns the generated approx given a `Property`. You must implement this method.
`getHistogramTitle(Property prop)`	Returns the histogram title for the given `Property`. You must implement this method.
`getName()`	Returns the display name of this approx generator.
`getUri()`	Returns the universal resource identifier (URI) of this approx generator.
`getXMLTag()`	Called by the Dynamic Ontology Manager whenever you create a or edit an existing property type. Use the `AdminComponentUtils.getXMLTag` method to obtain an `IPropertyApproxGenerator` configuration.
`isHistogramEligible()`	Determines whether this approx is eligible for display and counting in the histogram panel.
`setMenuText(String menuText)`	Sets the menu text to display when doing a SearchAround.

IPropertyApproxGenerator Coding Example

This example is a `PhoneNumberPropertyApproxGenerator` that takes the following complex property and normalizes it into `COUNTRY_CODE` and `AREA_CODE` to make prefix searches on phone numbers faster.

- `COUNTRY_CODE: 1`
- `AREA_CODE: 650`
- `PHONE_NUMBER: 5551212`

The approx might look like:

1:650

Because all phone number `Property` objects with that `COUNTRY_CODE` and `AREA_CODE` will be transformed similarly by the custom approx, the attribute is searchable.

Developing the Code for Your New Approx

The example creates a custom address approximation (approx) by using `CustomAddressApproxGenerator.generateApprox` (`Property` *prop*, `boolean` *b*). The custom approx iterates over the `PropertyTypeComponent` objects to find the `ZIP` address component and returns the `ZIP` string, if one is found. You can download a ZIP package containing the project code for this example.

The following method returns the generated approx for the `Address` property's ZIP component.

```
...
public String getApproximation(Property prop) {
    StringBuffer sb = new StringBuffer();
    for(PropertyTypeComponent component :
      prop.getType().getComponents())
    {
      // get value and run through the validators
      String compValue = prop.getDataMap().get(component.getUri());
      if(component.getUri().equals("ZIP"))
      {
        if(compValue != null && compValue.length() > 0)
        {
          sb.append(compValue);
        }
      }
}

return sb.toString();
}
```

You will notice that this approx is not histogram eligible:

```
public String getHistogramTitle(Property arg0) {
    return null;
}
...
public boolean isHistogramEligible() {
    return false;
}
public void setMenuText(String menuText) {
    menuText = menuText;
```

```
}
public String getMenuText() {
   return menuText;
}
...
```

Adding the Approx to Your Ontology

You cannot add this approx directly to the `Address` property because that property already exists and is intrinsic to the system. Instead you would need to do the following:

1. Add the plugin through the **Plugin Editor** panel.
2. Go to the **Property Editor** panel.
3. Select the **Address** property.
4. Click **Clone Property Type**.
   The **Basic Information** page appears.
5. Ensure these properties are set as follows:
   - **Display name** — Address
   - **URI** — com.palantir.property.customaddress
6. Click **Next** until you reach the last **Approxes** page.
7. Click **Add New.**
   The **Edit Validator** page appears.
8. Choose **Custom Address Approx** from the **Approx Type** drop down.
9. Enter a **Menu Name** in the field provided.
10. Click **Save**.
11. Click **Save Property Type** to save your new property.
12. Export your new ontology to the database.
13. Restart the Dispatch, Search, and Horizon Servers.

Launch the Palantir Workspace and test your approx by adding a ZIP property to an entity

12 IDataSourceMaker API

This chapter discusses how to use the `IDataSourceMaker` to create a custom code for validating and creating data sources for your deployment. The following topics appear in this chapter:

- Introduction to Data Sources and Sourcing.
- IDataSourceMaker APIs.
- Example IDataSourceMaker Workflow.
- IDataSourceMaker Coding Example.

Introduction to Data Sources and Sourcing

Data sources are files and databases imported into an investigation or added to the Data Repository so that they are available for use in investigations. Objects, links, notes and media all have associated data sources. ***Sourcing*** is the ability to assign a data source and create a data source on the fly if the appropriate data source does not already exist.

Palantir developers can optionally use APIs from the `IDataSourceMaker` interface to write code that validates user-created data sources and customizes how a document object for a data source is created.

If no custom `IDataSourceMaker` is configured, users can still create data sources on the fly, but they are not validated. The data source and a corresponding document object are created. The document object has the same title as the data source, but no additional properties, notes, links, or media.

If you use `IDataSourceMaker` to validate user-created data sources, be sure that the prerequisite tasks to enable custom code are completed. See Development on the Palantir Platform on page 7.

IDataSourceMaker APIs

Method	Description
`getUri()`	Returns a URI identifying the maker. Give the value to the Palantir administrator to add the `DATASOURCE_MAKER_URI = returned_string` to the Dispatch server's `dispatch.prefs` file.

Method	Description
`validate()`	Returns an error message if the input string (data source title) is invalid or the empty string ("") if the title is valid.
`make()`	Returns a collection of properties that should go onto the primary object for a user-created data source.

Example IDataSourceMaker Workflow

Several organizational roles work with Palantir data sources and sourcing. The following list shows a potential workflow. The example values are illustrative, not representative.

1. Optionally, a Palantir developer writes `IDataSourceMaker` validation code and creates package (JAR or PAR) that contains the custom code.

   Without custom `IDataSourceMaker` code, users can create data sources on the fly, but they are not validated. The data source and a corresponding document object are created. The document object has the same title as the data source, but no additional properties, notes, links, or media.

   The rest of this workflow assumes that you have custom validation code.

2. A Palantir administrator updates `dispatch.prefs` on the Dispatch Server to include the `DATASOURCE_MAKER_URI = URI` entry.

3. A Palantir administrator or developer deploys the `.jar` file.

   See Deploying Custom Code in a Staging or Production Environment on page 43 for information about deploying a custom

4. A Palantir administrator creates collection data sources in the Data Repository using the `dataSourceAdmin` CLI.

   A ***collection*** data source holds user-created data sources as its children. When a user creates a data source, its parent must be a collection data source, or the data source must have no parent.

   ```
   1. Add custom resource jar file
   2. Modify custom resource jar file
   3. Add a collection datasource
   4. Modify a collection datasource
   5. Exit
   Please select a choice: 3
   Name: data source
   Description: data source description
   Add datasource with name 'data source'? (y/n) y
   ```

   See the *Palantir Administrator's Guide* for details about additional actions available through the `dataSourceAdmin` CLI.

5. Analysts select an appropriate collection data source as the parent of a data source they create on the fly in the Palantir Workspace.

IDataSourceMaker Coding Example

You can download a ZIP package containing the project code for this example.

```
package com.palantir.custom;
...
public class ClassifiedDocDataSourceMaker implements IDataSourceMaker {
    private static final long serialVersionUID = 1L;

    private static final Logger log =
      LogManager.getLogger(ClassifiedDocDataSourceMaker.class);

    private static final String URI =
"com.palantir.maker.ClassifiedDocDataSourceMaker";

    // Regular expression used for validation and parsing
    //   Group 1 -> ClassifiedDocSerialNumber
    //   Group 2 -> ClassifiedDocClassification
    //   Group 3 -> ClassifiedDocTitle
    private static final String REGEX =
"(\\S+)\\s+\\((\\S+)\\)\\s*:\\s*(.+)$";

    // Property type URIs
    private static final String PROPERTY_TYPE_URI_SERIAL_NUMBER =
      "com.palantir.property.IdNumber";
    private static final String PROPERTY_TYPE_URI_CLASSIFICATION =
      "com.palantir.property.ClassificationLevel";
    private static final String PROPERTY_TYPE_URI_TITLE =
"com.palantir.property.Title";

    public String getUri() {
      return URI;
    }
    /**
     * Validates the string parsed by {@link #make(String, LocatorFactory)}.
     * Valid input has the following form:
     * ClassifiedDocSerialNumber (ClassifiedDocClassification):
ClassifiedDocTitle
     *
     * ClassifiedDocSerialNumber and ClassifiedDocClassification
     * can contain any non-space characters. ClassifiedDocTitle can
     * contain any character.
     *
     * @param input the input string to be validated.
     * @return the empty string ("") if the input is valid. Otherwise, a
message
     * describing the validation error.
     */
    public String validate(String input) {
      if (input == null)
        // Return error string instead of throwing NPE
        return "input not specified";

      if (log.isDebugEnabled())
        log.debug("Validating " + input);

      if (Pattern.matches(REGEX, input)) return ""; // valid input
      return "'" + input + "' is not a valid data source identifier";
```

```
    }

     /**
      * Constructs properties that should go on the primary object
      * for a user-created datasource. A DataSourceMakerException is
      * thrown if either the specified arguments are null / invalid
      * or an error occurs during property creation.
      *
      * @param input string from which the properties should be parsed.
      */
    public Collection<Property> make(String input, LocatorFactory lf)
        throws DataSourceMakerException {
      if (input == null) throw new DataSourceMakerException("input not
specified");
      if (lf == null) throw new DataSourceMakerException("location factory not
specified");

      if (log.isDebugEnabled())
        log.debug("Creating properties for " + input);

      Collection<Property> properties = new ArrayList<Property>();
      Matcher matcher = Pattern.compile(REGEX).matcher(input);
      if (matcher.matches()) { // Attempt to parse input
        // Parsing succeeded
        // Create and return property objects associated with extracted
components
        String serialNumber   = matcher.group(1);
        String classification = matcher.group(2);
        String title = matcher.group(3);
        try {
          properties.add(Property.attemptToCreate(lf, // Serial number
              PropertyType.getByUri(PROPERTY_TYPE_URI_SERIAL_NUMBER),
              serialNumber, Role.NONE));
          properties.add(Property.attemptToCreate(lf, // Classification
              PropertyType.getByUri(PROPERTY_TYPE_URI_CLASSIFICATION),
              classification, Role.NONE));
          properties.add(Property.attemptToCreate(lf, // Title
              PropertyType.getByUri(PROPERTY_TYPE_URI_TITLE),
              title, Role.NONE));
        } catch (PropertyMakerException e) { // Uh-oh!
          String message = "Error occurred while constructing data source
properties";
          log.error(message, e);
          throw new DataSourceMakerException(message, e);
        }
      } else { // Parsing failed
        String message = "'" + input + "' is not a valid data source
identifier";
        log.error(message);
        throw new DataSourceMakerException(message);
      }
      return properties;
    }
}
```

IDataSourceMaker Unit Test Example

```
package com.palantir.custom;
import java.lang.annotation.Annotation;
import java.util.ArrayList;
import java.util.Collection;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.junit.Test;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import com.palantir.dataservices.DataSourceMakerException;
import com.palantir.exception.PropertyMakerException;
import com.palantir.services.ptobject.Property;
import com.palantir.services.ptobject.PropertyType;
import com.palantir.services.ptobject.Role;
import com.palantir.util.LocatorFactory;
public class ClassifiedDocDataSourceMakerTest implements Test{
   private static final long serialVersionUID = 1L;

   private static final Logger log =
     LogManager.getLogger(ClassifiedDocDataSourceMakerTest.class);

   private static final String URI =
"com.palantir.maker.ClassifiedDocDataSourceMaker";

   // Regular expression used for validation and parsing
   //   Group 1 -> ClassifiedDocSerialNumber
   //   Group 2 -> ClassifiedDocClassification
   //   Group 3 -> ClassifiedDocTitle
   private static final String REGEX =
"(\\S+)\\s+\\((\\S+)\\)\\s*:\\s*(.+)$";

   // Property type URIs
   private static final String PROPERTY_TYPE_URI_SERIAL_NUMBER =
     "com.palantir.property.IdNumber";
   private static final String PROPERTY_TYPE_URI_CLASSIFICATION =
     "com.palantir.property.ClassificationLevel";
   private static final String PROPERTY_TYPE_URI_TITLE =
"com.palantir.property.Title";

   public String getUri() {
     return URI;
   }
   /**
    * Validates the string parsed by {@link #make(String, LocatorFactory)}.
    * Valid input has the following form:
    * ClassifiedDocSerialNumber (ClassifiedDocClassification):
ClassifiedDocTitle
    *
    * ClassifiedDocSerialNumber and ClassifiedDocClassification
    * can contain any non-space characters. ClassifiedDocTitle can
    * contain any character.
    *
    * @param input the input string to be validated.
     * @return the empty string ("") if the input is valid. Otherwise, a
message
     * describing the validation error.
```

```
     */
    public String validate(String input) {
      if (input == null)
        // Return error string instead of throwing NPE
        return "input not specified";

      if (log.isDebugEnabled())
        log.debug("Validating " + input);

      if (Pattern.matches(REGEX, input)) return ""; // valid input
      return "'" + input + "' is not a valid data source identifier";
    }

     /**
      * Constructs properties that should go on the primary object
      * for a user-created datasource. A DataSourceMakerException is
      * thrown if either the specified arguments are null / invalid
      * or an error occurs during property creation.
      *
      * @param input string from which the properties should be parsed.
      */
    public Collection<Property> make(String input, LocatorFactory lf)
        throws DataSourceMakerException {
      if (input == null) throw new DataSourceMakerException("input not
specified");
      if (lf == null) throw new DataSourceMakerException("location factory not
specified");

      if (log.isDebugEnabled())
        log.debug("Creating properties for " + input);

      Collection<Property> properties = new ArrayList<Property>();
      Matcher matcher = Pattern.compile(REGEX).matcher(input);
      if (matcher.matches()) { // Attempt to parse input
        // Parsing succeeded
        // Create and return property objects associated with extracted
components
        String serialNumber   = matcher.group(1);
        String classification = matcher.group(2);
        String title = matcher.group(3);
        try {
          properties.add(Property.attemptToCreate(lf, // Serial number
              PropertyType.getByUri(PROPERTY_TYPE_URI_SERIAL_NUMBER),
              serialNumber, Role.NONE));
          properties.add(Property.attemptToCreate(lf, // Classification
              PropertyType.getByUri(PROPERTY_TYPE_URI_CLASSIFICATION),
              classification, Role.NONE));
          properties.add(Property.attemptToCreate(lf, // Title
              PropertyType.getByUri(PROPERTY_TYPE_URI_TITLE),
              title, Role.NONE));
        } catch (PropertyMakerException e) { // Uh-oh!
          String message = "Error occurred while constructing data source
properties";
          log.error(message, e);
          throw new DataSourceMakerException(message, e);
        }
      } else { // Parsing failed
        String message = "'" + input + "' is not a valid data source
identifier";
```

```
            log.error(message);
            throw new DataSourceMakerException(message);
          }
          return properties;
        }
        public Class<? extends Throwable> expected() {
          // FIXME Auto-generated method stub
          return null;
        }
        public long timeout() {
          // FIXME Auto-generated method stub
          return 0;
        }
        public Class<? extends Annotation> annotationType() {
          // FIXME Auto-generated method stub
          return null;
        }
    }
```

13 DatePrimitiveMaker: Extending the Ontology's Date Parsing

This section provides information about customized date and time formats in the Palantir Platform. The following topics are discussed:

- Introduction to the Date/Time Formats
- Default DatePrimitiveMaker Formats
- Adding Patterns to the Default Date Parser
- Writing a Custom DatePrimitiveMaker Plugin

Introduction to the Date/Time Formats

In Palantir, dates can be any of the following logical types:

date	Stored as `MMMMM dd, yyyy`, for example: December 24, 2007
time	Stored as `HH:mm:ss ZZ`, where `HH` is a 24-hour clock (military time). Examples: 10:23:23 -0800 (approximately 10:23am) 16:23:23 -0800 (approximately 4:23pm)
date/time	Combines `Date` and `Time` and is stored as `MMMMM dd, yyyy HH:mm:ss ZZ` for example: December 24, 2007 6:23:23 -0800.

The Palantir `Date` property type has the URI `com.palantir.property.Date` and by default uses the `DatePrimitiveMaker`. The `DataPrimitiveMaker` uses multiple Joda formats for parsing date formats. You can add additional parsing patterns to the default `DatePrimitiveMaker`. You use any date formats from Joda-Time `DateTimeFormat` (see http://joda-time.sourceforge.net/api-release/org/joda/time/format/DateTimeFormat.html).

You can write your own date primitive maker plugin to parse dates. You can write multiple plugins of this type and install them. The system allows you to add and then order parsers. The systems calls the makers when ever it creates a date property. The system calls each parsers in order and returns the results from the first maker that succeeds.

Default DatePrimitiveMaker Formats

`DatePrimitiveMaker` has a default set of `Date` formats. The system processes these in order and the first success is returned. Default formats (not including Z at the end of each format) include the following:

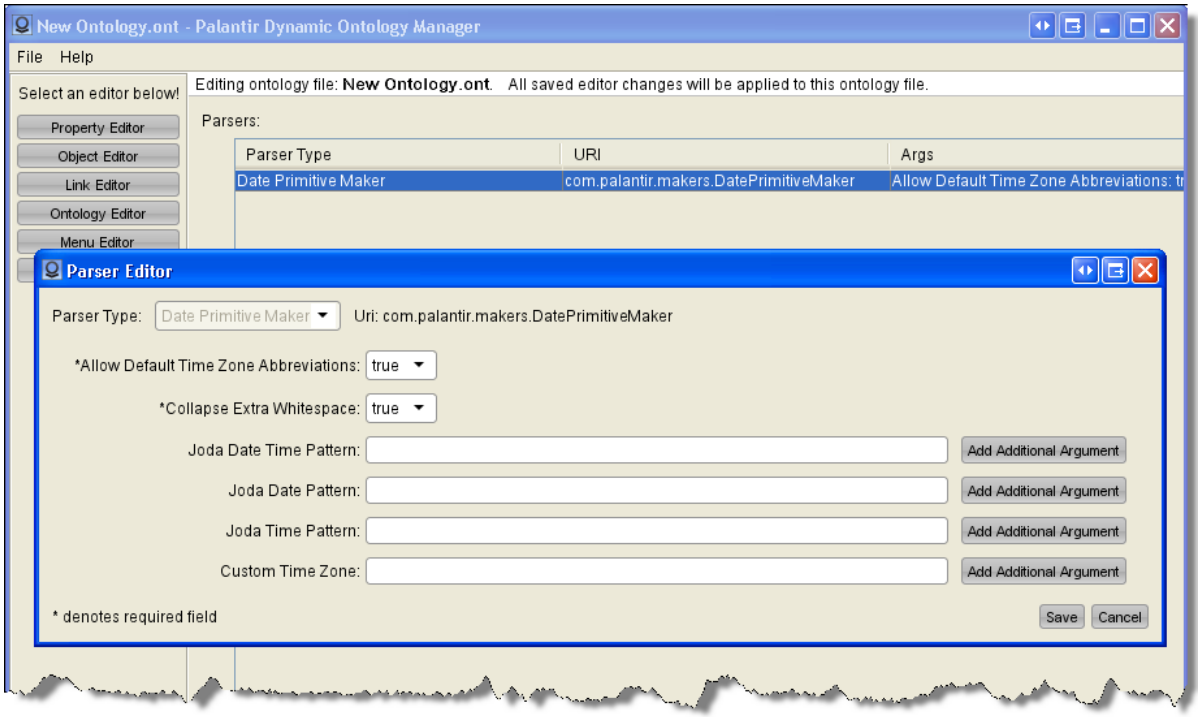
```
"MMMM d, yyyy hh:mm:ss a" // November 15, 2005 11:15:34 AM
"MMMM d, yyyy hh:mm a" // November 15, 2005 11:15 AM
"MM/dd/yy hh:mm:ss a" // 11/15/05 11:15:34 AM
"MM/dd/yy hh:mm a" // 11/15/05 11:15 AM
"M/d/yy H:mm:ss" // 11/15/05 11:15:34
"M/d/yy H:mm"; // 11/15/05 11:15
"M-d-yy H:mm:ss"; // 11-15-05 11:15:34
"M-d-yy H:mm"; // 11-15-05 11:15
"dd-MMM-yy H:mm:ss"; // 15-NOV-05 11:15:34 (also 15-November-05 11:15:34)
"dd-MMM-yy H:mm"; // 15-NOV-05 11:15 (also 15-November-05 11:15)
"dd-MMM-yyyy H:mm:ss" since yy also does yyyy when there are 4 digits
"yyyy-MMM-dd H:mm:ss"; // 2005-NOV-15 11:15:34
"yyyy-MMM-dd H:mm"; // 2005-NOV-15 11:15
"yyyy MMM dd H:mm:ss"; // 2005 NOV 15 11:15:34
"yyyy MMM dd H:mm"; // 2005 NOV 15 11:15
"yyyy-MM-dd H:mm:ss"; // 2005-11-15 11:15:34
"yyyy-MM-dd H:mm"; // 2005-11-15 11:15
"yyyy/MM/dd H:mm:ss"; // 2005/11/15 14:22:33
"yyyy/MM/dd H:mm"; // 2005/11/15 14:22
"MMMM d, yyyy H:mm:ss"; // {November | Nov} 15, 2005 11:15:34
"MMMM d, yyyy H:mm"; // November 15, 2005 11:15 (also Nov 15, 2005 11:15)
"MMM. d, yyyy H:mm:ss"; // Nov. 15, 2005 11:15:34
"MMM. d, yyyy H:mm"; // Nov. 15, 2005 11:15
"MMM't'. d, yyyy H:mm:ss"; // Sept. 15, 2005 11:15:34
"MMM't'. d, yyyy H:mm"; // Sept. 15, 2005 11:15
ddHHmm'Z' MMM yy // (071927Z JUN 07)
ddHHmm'z' MMM yy // (151115z Nov 08)
ISO8601 date formats. See http://en.wikipedia.org/wiki/ISO_8601.
MMMM d, yyyy // (November 15, 2005)
MM/dd/yy // (11/15/2005)
MM-dd-yy // 01-25-07
M-d-yyyy // (11-5-2005)
"dd-MMM-yy"; // 25-JAN-07 (also 25-January-07)
"dd-MMM-yyyy" // 25-JAN-2007 (also 25-January-2007)
"dd MMM yy"; // 25-JAN-07 (also 25-January-07)
"yyyy-MMM-dd"; // 2007-JAN-25 (also 2007-January-25)
"yyyy MMM dd"; // 2007 JAN 25 (also 2007 January 25)
"yyyy-MM-dd"; // 2007-01-25
"yyyy/MM/dd"; // 2007/01/25
"MMM. d, yyyy"; // Jan. 25, 2007 (Nov. 15, 2005)
"MMMM d yyyy"; // September 25 2007 (Nov 15, 2005)
"MMM't'. d, yyyy"; // Sept. 25, 2007
"hh:mm:ss a";
"hh:mm a";
"H:mm"; // 11:15
"H:mm:ss"; // 11:15:34
ddHHmm MMM YY
ddHHmmssZMMMyy
2200 HOURS ON JUNE 11 2008
2200 HOURS ON JUNE 11, 2008
1030 HOURS TANGO TIME ON JUNE 11 2008
    (http://www.timeanddate.com/library/abbreviations/timezones/military/)
```

```
1245 HOURS ON 15 JUNE 2008
0900 HOURS ON 9 JUN 2008 (single-digit day)
1530 ON 17 JUNE 2008
0700 ON JUNE 07 2008
1500 ON 07 JUN 08
08 JUN 08 AT ABOUT 1400
```

The above list is not comprehensive. You should test with sample data to identify whether or not a particular format is supported.

Adding Patterns to the Default Date Parser

A single Parser Type of `DatePrimitiveMaker` exists in a given ontology. This maker parses date values for all date-based components and properties in the ontology. You can define additional patterns for the `com.palantir.property.Date` property to extend its default date-parsing logic.

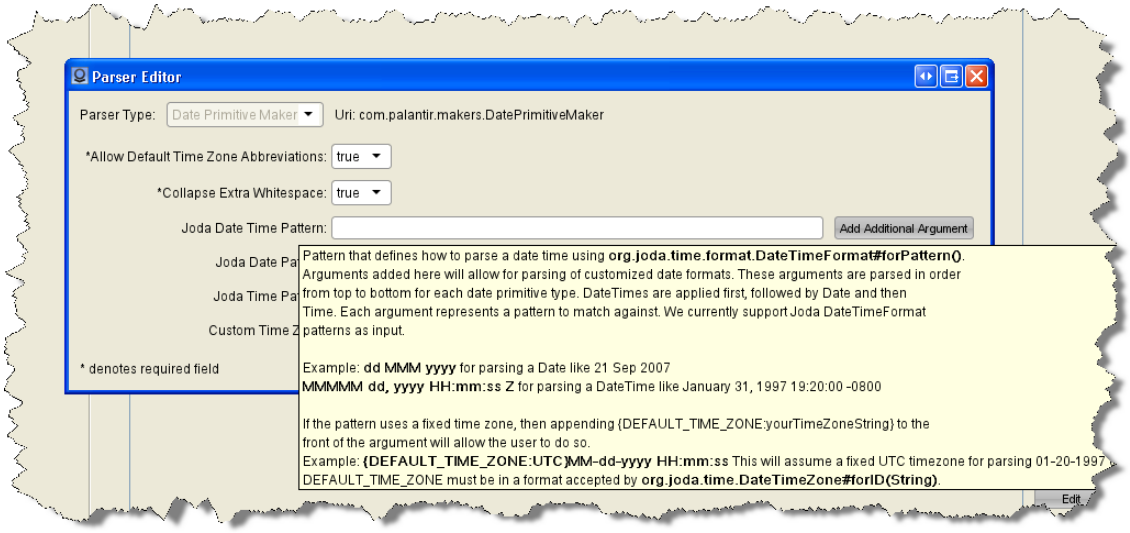


There are different fields for **Joda DateTime**, **Date**, and **Time** patterns. The underlying date parsers are Joda-Time's **DateTimeFormat**. The parsing order is pattern dependent. **DateTime** patterns are processed first, followed by **Date** patterns, and then **Time** patterns.

The **DatePrimitiveMaker** attempts to parse input to determine the input's **Date**, **Time**, or **DateTime** structure. It then formats the particular date instance using that structure's format for storage in the property's data map. Parsing tries to create a **DateTime** first, then a **Date**, and finally a **Time**.

The following procedure explains how to add a pattern to the default `DatePrimitiveMaker`. Before you begin, you must start the Palantir Dynamic Ontology Manager and load in an ontology, then do the following:

1. Open the **Property Editor**.
2. Double-click the **`Date`** property in the **Property Type Name** column to edit it.
   Its URI is `com.palantir.property.Date`. The **Basic Information** appears.
3. Click **Next** until the **Parsers** page appears with a list of parsers.
4. Double-click the **Date Primitive Maker** parser.
5. Click the **Add New** button if there is no parser defined.
   It should have at least **Date Primitive Maker** (which can have multiple parsing patterns).
6. In the **Parser Editor** enter a pattern string in the correct entry field for the pattern type you need.
   Use the detailed tooltips for explanations of each field. For example:



7. If needed, click **Add** to specify more patterns of the same type.
   You can add as many patterns in a specific field as you need. Just click **Add** between each entry.
8. Optionally, specify a **Custom Time Zone** value, if needed.
9. Click **Save** to save your entry and return to the **Parsers** page.
10. Click **Next** to go to the **Approxes** page.
11. Click **Save Property Type**.

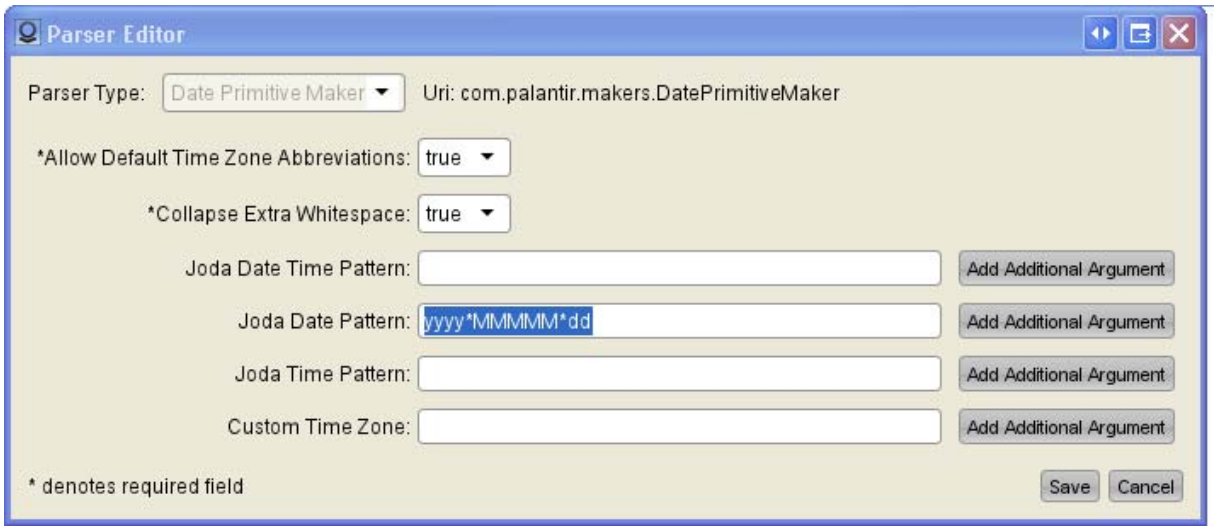
Example DatePrimitiveMaker Parser Customization

Perhaps you want to support the following date format:

```
1996*January*23
```

where * separates the year, month, and day. This is a simple date pattern without the time, so you should use **Joda Date Pattern**.

1. Follow steps 1 through 7 in Adding Patterns to the Default Date Parser on page 157 to open the Parser Editor.
2. Enter the pattern `yyyy*MMMMM*dd` in the **Joda Date Pattern** field.



3. Click **Save**.
4. Click **Next** to go to the **Approxes** page.
5. Click **Save Property Type**.

Writing a Custom DatePrimitiveMaker Plugin

This section explains how to write a `DatePrimitiveMaker` plugin. It first introduces the API and then provides example code for the API.

The AbstractDatePrimitiveMaker API

To write a custom date primitive maker, you extend the `AbstractDatePrimitiveMaker` class. This class implements both the `IDatePrimitiveMaker` and the `IPropertyMaker`. Specify the `com.palantir.services.impl.property.IPropertyMaker` component value in your `ptplugin.xml` file.

The list of `IPropertyMaker` methods you must implement are described by Customizing Property Parsing on page 132. Additionally, you must implement the `IDatePrimtiveMaker.parseDatePrimitiveValue()` method. Your implementation should parse the date from a string value and return a `DatePrimitiveMakerValue` representing the particular date, date-time, or time. Return null for values that cannot be parsed. Your method implementation should be fast.

Example of a CustomDatePrimitive Maker

The example creates a parser that parses eight digit dates converting 200090130 to January 30, 2009. It also parses the format of XXddMMMHHmm where XX are two letters followed by Day, Month, Time an dusing the current year. You can download a ZIP package containing the project code for this example.

Your code should include `POSSIBLE_ARGS` that contain a String map with your parse formulas:

```
...
   private static final long serialVersionUID = 1L;
   private static final Logger log =
Logger.getLogger(CustomDatePrimitiveMaker.class);

   private static final String NAME = "Custom Date Primitive Maker";
   private static final String URI =
"com.palantir.plugin.maker.dateparser.CustomDatePrimitiveMaker";
   private static final String PARSE_EIGHT_DIGIT_DATES =
"parseEightDigitDates";
   private static final String PARSE_LLDDMMMHHMM_DATES =
"parseLLDDMMMTimeDates";

   private static final String[][] POSSIBLE_ARGS = {
     { "Parse 8-digit Dates", PARSE_EIGHT_DIGIT_DATES,
        "Whether to parse eight digit dates eg 20090130 as January 30, 2009",
        Required.TRUE, Multiplicity.SINGLE,
        EditorDisplayType.BOOLEAN, AllowUpdate.TRUE },
     { "Parse LLDDMMMTIME Dates", PARSE_LLDDMMMHHMM_DATES,
        "Whether to parse dates of the XXddMMMHHmm format (Two letters
followed by Day Month Time and using current year) Ex: TJ19JAN1655 as January
19, 2010 16:55:00 -08:00.",
        Required.TRUE, Multiplicity.SINGLE,
        EditorDisplayType.BOOLEAN, AllowUpdate.TRUE }
   };
   private static final Pattern eightDigits =
Pattern.compile("(\\d{2})(\\d{2})(\\d{2})(\\d{2})");
   private static final Pattern llDayMonthTime = Pattern.compile("[a-
z]{2}(\\d{2})([a-z]{3})(\\d{2})(\\d{2})", Pattern.CASE_INSENSITIVE);
   private static final DateTimeFormatter monthFormatter =
DateTimeFormat.forPattern("MMM");
   private static final DateTimeFormatter yyyyMMddFormatter =
DateTimeFormat.forPattern("yyyyMMdd");
   private static final DateTimeFormatter ddMMyyyyFormatter =
DateTimeFormat.forPattern("ddMMyyyy");
   private static final DateTimeFormatter mmddyyyyFormatter =
DateTimeFormat.forPattern("MMddyyyy");

   private boolean parseLLDayMonthTimeFormat = true;
```

```
    private boolean parseEightDigitFormat = true;
...
```

The Palantir Platform passes a String with the date to the maker. The maker processes the string with the `parseDatePrimitiveValue()` method:

```
public DatePrimitiveMakerValue parseDatePrimitiveValue(String value) {
     if (parseEightDigitFormat) {
       Matcher m = eightDigits.matcher(value);
       if (m.matches()) {
         List<DateTime> parsedValues = new ArrayList<DateTime>();
         try {
           parsedValues.add(yyyyMMddFormatter.parseDateTime(value));
         } catch (IllegalArgumentException iae) {
           // do nothing
         }
         try {
           parsedValues.add(ddMMyyyyFormatter.parseDateTime(value));
         } catch (IllegalArgumentException iae) {
           // do nothing
         }
         try {
           parsedValues.add(mmddyyyyFormatter.parseDateTime(value));
         } catch (IllegalArgumentException iae) {
           // do nothing
         }
         for (DateTime dt : parsedValues) {
           if (1900 <= dt.getYear() && dt.getYear() <= 2100) {
             return new DatePrimitiveMakerValue(dt, value,
DatePrimitiveType.DATE, false);
           }
         }
       }
     }

     if (parseLLDayMonthTimeFormat) {
       Matcher m = llDayMonthTime.matcher(value);
       if (m.matches()) {
         try {
           int day = Integer.parseInt(m.group(1));
           String month = m.group(2);
           int hour = Integer.parseInt(m.group(3));
           int min = Integer.parseInt(m.group(4));
           DateTime monthDate = monthFormatter.parseDateTime(month);
           if (monthDate != null) {
             DateTime dt = new DateTime(new DateTime().getYear(),
monthDate.getMonthOfYear(), day, hour, min, 0, 0);
             return new DatePrimitiveMakerValue(dt, value,
DatePrimitiveType.DATE_TIME, false);
           }
         } catch (NumberFormatException nfe) {
           // could not parse day hour or min
         } catch (IllegalArgumentException iae) {
           // could not parse month
         }
       }
     }
     return null;
   }
```

14 IPasswordPolicy API

This chapter contains the following topics:

- Introduction to the Password Policy Interface.
- IPasswordPolicy API.
- Error Handling.
- IPasswordPolicy Sample Implementation.

Introduction to the Password Policy Interface

By default, intrinsic users must have passwords that contain at least eight mixed-case characters (minimum of one upper-case and one lower-case), one number, and one symbol. If the default password complexity options do not meet your deployment's requirements, you can develop a custom `IPasswordPolicy` plugin. `IPasswordPolicy` overrides the default validation of passwords. This policy is then applied when creating or modifying user passwords.

To use a custom `IPasswordPolicy`:

1. Implement the `IPasswordPolicy` interface and package your implementation in a `.jar` file, for example, `pwpolicy.jar`.
2. Create an `/ext` subdirectory in the `PALANTIR_HOME` directory.
3. Copy the `pwpolicy.jar` created in Step 1 to the `PALANTIR_HOME/ext` directory.
4. Add the following setting to `dispatch.prefs`:

   ```
   PASSWORD_POLICY_CLASS = com.palantir.test.none.TestPasswordPolicy
   ```

   where `com.palantir.test.none.TestPasswordPolicy` is your implementation class.
5. Modify wrapper-conf/`dispatch.conf` to add `pwpolicy.jar` to the CLASSPATH. For example:

   ```
   ...
   # Java Classpath (include wrapper.jar) Add class path elements
   # as needed starting from 2
   wrapper.java.classpath.2=%PALANTIR_HOME%/ext/pwpolicy.jar
   ...
   ```
6. Restart the Dispatch Server.

No changes are required on the client side.

IPasswordPolicy API

The API for the `IPasswordPolicy` interface is as follows.

```
import com.palantir.password.WeakPasswordException;
/**
    * Provided a password, this method validates it.
    *
    * @param password the password to check
    * @throws WeekPasswordException describing the reason the password
    *     failed to pass the checks. Should have the user-friendly
    *     message filled in.
    */
    public void checkPassword(char [] password)
      throws WeekPasswordException;
```

Error Handling

To be useful, a policy should be enforced regardless of potential runtime errors. To provide consistent handling of all possible error cases, such as class not found, instantiation exception, or cast exception, the `IPasswordPolicy` API uses a simple failure and error-handling mechanism. If an error occurs, then the:

- User is notified with the `WeekPasswordException` message text.
- Error is logged (for the administrator's benefit).
- Password cannot be changed.

IPasswordPolicy Sample Implementation

You can download a ZIP package containing the project code for this example.

```
package com.palantir.custom;
...
import com.palantir.password.WeakPasswordException;
import com.palantir.services.interfaces.IPasswordPolicy;
...
/**
 * This TestPasswordPolicyFactory is a factory which generates the
TestPasswordPolicy.
 *
 */
public class TestPasswordPolicy implements IPasswordPolicy {
    /**
     * Determine the character class of the argument.
     */
    private static int charClass(final char ch) {
      if (Character.isLowerCase(ch)) {
        return 0;
      }
```

```
  if (Character.isUpperCase(ch)) {
    return 1;
  }
  if (Character.isDigit(ch)) {
    return 2;
  }
  if (!Character.isISOControl(ch)) {
    return 3;
  }
  return 4;
}
/**
 * Determine if argument complies with the spec.
 */
public void checkPassword(char[] password)
throws WeekPasswordException
{
  String arg = new String(password);
  if (arg == null || arg.length () < 8) throw new
     WeekPasswordException("Password must have at least 8 characters.");
  int repeats = 0;
  int classRepeats = 0;
  final int[] histogram = new int[5]; // distribution of char classes
  final char[] array = arg.toCharArray ();
  if (!Character.isLetterOrDigit(array[0])) throw new
     WeekPasswordException("Password must start with a letter or digit.");
  int lastClass = charClass(array[0]);
  histogram[lastClass]++;
  for (int i = 1; i < array.length; i++) {
    if (array[i] == array[i - 1]) {
      if (++repeats > 1) throw new WeekPasswordException("Password cannot
         contain equal consecutive characters.");
    } else {
      repeats = 0;
    }
    final int klass = charClass(array[i]);
    if (klass == lastClass) {
      if (++classRepeats > 3) throw new WeekPasswordException("Password
      cannot contain more than 3 consecutive letters, digits or symbols.");
    } else {
      classRepeats = 0;
    }
    histogram[klass]++;
    lastClass = klass;
  }
  if (histogram[4] > 0) throw new WeekPasswordException("Password contains
    illegal characters. Please use upper and lowercase letters, digits and
    symbols only.");

  int classCount = 0;

  for (int i = 0; i < 4; i++) {
    if (histogram[i] > 0) classCount++;
  }

  if(classCount < 3)
  {
    throw new WeakPasswordException("Password must contain at least one
      uppercase letter, lowercase letter, digit and symbol.");
```

```
    }
  }
  public static void main(String[] args) {
    Object[][] tests = {
        { "", false },
        { "abc", false },
        { "aB1", false },
        { "aaaaaaaa", false },
        { "aaadefgh", false },
        { "abcdefgh", false },
        { "Abcdefgh", false },
        { "A1cdefgh", false },
        { "A1@#$%^2", false },
        { "aaaB11$$", false },
        { "aAbBcCdD", false },
        { "aaBB11$$", true },
        { "A1@#$%B2", true },
        { "A1cdEfgh", true },
        { "A1cdEfgh234", true },
        { "A1cdEfgh234!@#", true },
        { "aA1!bB5%cC3#dD3#", true },
    };
    int bad = 0;
    IPasswordPolicy policy = new TestPasswordPolicy();
    for (Object[] test : tests) {
      boolean fail = false;
      try{
        policy.checkPassword(test[0].toString().toCharArray());
        if(!((Boolean)test[1]).booleanValue())
        {
          fail = true;
        }
      } catch (WeekPasswordException pe)
      {
        if(((Boolean)test[1]).booleanValue())
        {
          fail = true;
        }
      }
      if(fail)
      {
        bad++;
        System.out.println (test[0].toString() + " is not " + test[1]);
      }
    }
    System.out.println(bad == 0 ? "SUCCESS" : "FAILURE");
  }
}
```

Index

W