



HelixCore

Helix Core P4Java Developer Guide

2020.1
July 2020

PERFORCE

www.perforce.com



Copyright © 2009-2020 Perforce Software, Inc..

All rights reserved.

All software and documentation of Perforce Software, Inc. is available from www.perforce.com. You can download and use Perforce programs, but you can not sell or redistribute them. You can download, print, copy, edit, and redistribute the documentation, but you can not sell it, or sell any documentation derived from it. You can not modify or attempt to reverse engineer the programs.

This product is subject to U.S. export control laws and regulations including, but not limited to, the U.S. Export Administration Regulations, the International Traffic in Arms Regulation requirements, and all applicable end-use, end-user and destination restrictions. Licensee shall not permit, directly or indirectly, use of any Perforce technology in or by any U.S. embargoed country or otherwise in violation of any U.S. export control laws and regulations.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce.

Perforce assumes no responsibility or liability for any errors or inaccuracies that might appear in this book. By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Any additional software included within Perforce is listed in "[License Statements](#)" on page 48.

Contents

How to use this guide	4
Syntax conventions	4
Feedback	4
Other documentation	4
P4Java Programming	6
System Requirements	6
Installation	6
Documentation	6
Sample programs	7
Java package roadmap	7
Basic P4Java usage model	7
Typical usage patterns	9
The IOptionServer and IClient interfaces and the ServerFactory class	10
Exception and error handling	12
Helix server file operations	13
Summary vs. Full Objects	14
Using option classes	16
Advanced usage notes	17
Helix server addresses, URIs, and properties	18
SSL connection support	18
The IServerResource interface	19
P4Java properties	19
Character Set Support	20
Error Message Localization	21
Logging and tracing	21
Standard implementation classes	23
I/O and file metadata issues	24
Threading issues	24
Authentication	25
Other Notes	25
Glossary	29
License Statements	48

How to use this guide

This guide provides information on installing and using P4Java. It assumes basic knowledge of both Java (JDK 7 or later) and Helix server.

This section provides information on typographical conventions, feedback options, and additional documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
literal	Must be used in the command exactly as shown.
<i>italics</i>	A parameter for which you must supply specific information. For example, for a <i>serverid</i> parameter, supply the ID of the server.
<code>[-f]</code>	The enclosed elements are optional. Omit the brackets when you compose the command.
<code>...</code>	Previous argument can be repeated. <ul style="list-style-type: none"> ▪ <code>p4 [g-opts] streamlog [-l -L -t -m max] stream1 ...</code> means <code>1</code> or more stream arguments separated by a space ▪ See also the use on <code>...</code> in Command alias syntax in the <i>Helix Core P4 Command Reference</i> <div style="border: 1px solid #ccc; background-color: #f0f0f0; padding: 5px; margin-top: 10px;"> <p>Tip <code>...</code> has a different meaning for directories. See Wildcards in the <i>Helix Core P4 Command Reference</i>.</p> </div>
<code>element1 element2</code>	Either <i>element1</i> or <i>element2</i> is required.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See <https://www.perforce.com/support/self-service-resources/documentation>.

Tip

You can also search for Support articles in the [Perforce Knowledgebase](#).

Earlier versions of this guide: [2019.1](#)

To find earlier versions of this guide, use the following URL and replace *v17.2* with the version number you are looking for: <https://www.perforce.com/manuals/v17.2/p4java/index.html>

P4Java Programming

P4Java is a Java API developed by Perforce that enables applications to access Helix server, the Perforce enterprise version control system in a “Java natural” and Java-native way. P4Java presents Helix server services and managed resources and files as first-class Java interfaces, classes, methods, and objects, rather than as simple strings or command-line-style functions. This approach makes it easier to integrate the API into Java applications and tools, and is particularly useful for integrating Helix server into model-view-controller (MVC) contexts and workflows.

P4Java is aimed mostly at the following types of Java development:

- Standalone applications that need to access Helix server services from within the application
- Plug-ins for Java tools such as Eclipse, ant, Mylyn, Cruise Control, and so on, that need to communicate with Helix server
- J2EE applications, where P4Java can be embedded within a servlet and/or presented as a web service or an AJAX binding for client-side use

System Requirements

P4Java requires Java 7 or later.

Due to current US export control restrictions for some countries, the standard JDK package only comes with 128 bit encryption level cyphers. In order to use P4Java to connect to an SSL-enabled Helix server, those living in eligible countries may download the unlimited strength JCE (Java Cryptography Extension) package and replace the current default cryptography jar files with the unlimited strength files. For more information about using P4Java with an SSL enable Helix server, see <https://community.perforce.com/s/article/2620>.

For details, refer to the [P4Java Release Notes](#).

Installation

Download the P4Java ZIP file from the Perforce web site, see <https://www.perforce.com/downloads/helix-core-api-java>.

Extract the enclosed JARs and other files to a temporary directory, then install the `p4java.jar` JAR file into a location that is suitable for use by compilers, JVMs, and other development tools or applications.

Documentation

Included with the P4Java ZIP file is a directory of documentation that contains this document and a full Javadoc document set for all public interfaces and classes.

The Javadoc document set can be found at:

Sample programs

Sample P4Java applications are available on GitHub in the Perforce area, see <https://github.com/perforce/p4java-examples>.

These samples are used throughout this document to illustrate common usage patterns and simple code snippets, and can also be used as the basis for further user experiments with P4Java development.

Java package roadmap

The P4Java API contains the following main public packages:

- **com.perforce.p4java**: the main **p4java** package hierarchy root. Contains a small handful of API-wide definitions and classes for activities like logging, tracing, and package metadata.
- **com.perforce.p4java.server**: contains the server factory class, **IServer**, and **IOptionsServer** server interfaces, and associated classes and interfaces. **IOptionsServer** extends **IServer** and is the preferred interface. This package enables participating applications to connect to Helix servers and start interacting with Helix server services through the **IServer** interface.
- **com.perforce.p4java.client**: defines the **IClient** client interface and associated classes and support definitions. Participating applications typically use the **IClient** interface to access Helix server client services such as syncing and adding, editing, or deleting files.
- **com.perforce.p4java.exception**: defines the main publicly-visible exceptions likely to be encountered in general use, and some specialized and rarely-encountered errors.
- **com.perforce.p4java.core**: contains interface definitions for major Helix server-managed objects such as changelists, jobs, and clients.
- **com.perforce.p4java.core.file**: contains the main **IFileSpec** interface for accessing and defining the various types of files that Helix server manages (for example, **depot**, **local**, and **client**), along with associated definitions.
- **com.perforce.p4java.impl.generic**: root package for “generic” or standard implementations of many useful Helix server client, changelist, job, and similar interfaces. These implementations are available for use by participating applications, but are not mandatory.

Basic P4Java usage model

The following basic model for P4Java reflects typical Helix server usage:

1. A Java application uses the P4Java `ServerFactory` class to obtain an `IOptionsServer` interface onto a specific Helix server at a known network address and port, and connects to this Helix server through the `IOptionsServer` interface that is returned from the factory.
2. The application optionally logs in to the Helix server through the `IOptionsServer`'s login and associated methods.
3. The application obtains a suitable `IClient` interface into a Helix server client workspace through the `IOptionsServer` interface's "getClient()" method.
4. The application syncs the Helix server client workspace through the `IClient` interface's sync method.
5. The application gets and processes (Java `java.util.List`) lists of depot, client, and local files in (or associated with) the Helix server client workspace, through the `IOptionsServer` and `IClient` interfaces.
6. The application adds, edits, or deletes files in the local Helix server client workspace using the `IClient` interface. These files are added to the default or a numbered Helix server changelist represented by one or more `IChangelist` interfaces, which are obtained through the `IClient` or `IOptionsServer` interfaces. There are often several ways to obtain a specific type of object depending on context, but these tend to be convenience methods rather than fundamental.
7. The application submits a specific changelist using the associated `IChangelist` interface. This submission can be linked with one or more Helix server jobs, represented by the `IJob` interface.
8. The application can switch between Helix server workspaces, browse Helix server jobs and changelists, log in as a different user, and add, edit, or delete files, using the relevant `IOptionsServer` or `IClient` interfaces.
9. To disconnect from a Helix server, the application calls the `disconnect` method on the `IOptionsServer` interface.

It is good practice to determine how to obtain and work with data using the `p4` command line client before implementing it in P4Java. In general, the methods and options available on the various P4Java API interfaces map to the basic Helix server commands (or the familiar `p4` command line equivalent), but there are exceptions. Not all Helix server commands are available through the P4Java API.

This usage model relies heavily on representing all significant Helix server objects — clients, servers, changelists, jobs, files, revisions, labels, branches, and so on — as first-class Java interfaces, classes, or enums, and, where appropriate, returning these objects as ordered Java lists so that the developer can iterate across the results using typical Java iterator patterns. P4Java uses JDK 5 (and later) parameterized types for these lists.

P4Java represents most recoverable usage errors and Helix server errors as Java exceptions that are subclassed out of the main `P4JavaException` class, and thrown from nearly every significant `IOptionsServer` and `IClient` interface method (and from subsidiary and associated class methods). Most such errors are connection errors (caused by a network or connectivity issue), access errors (caused by permissions or authentication issues), or request errors (caused by the Helix server detecting a badly-constructed request or non-existent file spec). P4Java applications catch and recover from these errors in standard ways, as discussed in "Exception and error handling" on page 12.

Exceptions are not used in methods that return multiple files in lists, because the server typically interpolates errors, informational messages, and valid file specs in the same returns. P4Java provides a single method call as a standard way of identifying individual returns in the (often very long) list of returns, discussed in detail in "Helix server file operations" on page 13.

Unlike the Helix C/C++ API or the `p4` command-line client, P4Java is not intended for direct end-user interaction. Rather, P4Java is intended to be embedded in GUI or command-line applications to provide Helix server client/server communications, and P4Java assumes that the surrounding context supplies the results of user interaction directly to P4Java methods as Java objects. Consequently, many of the environment variables used by command-line client users (such as `P4PORT` or `P4USER`) are deliberately ignored by P4Java. The values they usually represent must be explicitly set by appropriate `IOptionsServer` methods or other methods.

The standard default P4Java server and client implementations are basically thread-safe. To avoid deadlock and blocking, refer to "Threading issues" on page 24.

Typical usage patterns

This section briefly describes typical usage patterns and provides a starting point for developers using P4Java for the first time. Many examples below are snippets from (or refer to) the P4Java sample programs available on GitHub in the Perforce area, see <https://github.com/perforce/p4java-examples>.

In this section:

The IOptionsServer and IClient interfaces and the ServerFactory class	10
Exception and error handling	12
Helix server file operations	13
Summary vs. Full Objects	14
Using option classes	16

The IOptionsServer and IClient interfaces and the ServerFactory class

The `com.perforce.p4java.server.IOptionsServer` interface represents a specific Helix server in the P4Java API, with methods to access typical Helix server services. Each instance of a `IOptionsServer` interface is associated with a Helix server running at a specified location (network address and port), and each `IOptionsServer` instance is obtained from the P4Java server factory, `com.perforce.p4java.server.ServerFactory`, by passing it a suitable server URI and optional Java properties.

The snippet below is from the `ServerFactoryDemo` class in the sample package, and shows a very simple way to prompt the user for a Helix server URI, connect to the server at the URI, and get basic information about that server. This is the basic “Hello World!” P4Java application, and works like the `p4 info` command (with suitable attention being paid to formatting details with the `formatInfo` method below).

```
BufferedReader lineReader = new BufferedReader (
    new InputStreamReader (System.in));

try {
    for (;;) {
        System.out.print (PROMPT);
        String serverUriString = lineReader.readLine();

        if ((serverUriString == null) || serverUriString.equalsIgnoreCase
(QUIT)) {
            break;
        } else {
            IOptionsServer server =
                ServerFactory.getOptionsServer (serverUriString, null,
                    new UsageOptions (null).setProgramName
("P4JavaDemo")
                        .setProgramVersion ("alpha0.9"));
            server.connect ();
            IServerInfo info = server.getServerInfo ();

            if (info != null) {
                System.out.println ("Info from Perforce server at URI '"
                    + serverUriString + "' for '"
```

```

        + server.getUsageOptions().getProgramName() +
        ":");
        System.out.println(formatInfo(info));
    }

    if (server != null) {
        server.disconnect();
    }
}
}
} catch (RequestException rexc) {
    System.err.println(rexc.getDisplayString());
    rexc.printStackTrace();
} catch (P4JavaException exc) {
    System.err.println(exc.getLocalizedMessage());
    exc.printStackTrace();
} catch (IOException ioexc) {
    System.err.println(ioexc.getLocalizedMessage());
    ioexc.printStackTrace();
} catch (URISyntaxException e) {
    System.err.println(e.getLocalizedMessage());
    e.printStackTrace();
}
}

```

Multiple **IOptionsServer** objects can represent the same physical Helix server, and this approach is recommended for heavyweight usage and for multi-threaded applications.

The Java properties parameter passed to the factory in the first example is null, but you can pass in a variety of generic and implementation-specific values as described in ["Character Set Support" on page 20](#).

Helix server client workspaces are represented by the **com.perforce.p4java.client.IClient** interface, which can be used to issue Helix server client workspace-related commands such as sync commands, file add/delete/edit commands, and so on. A **IClient** interface is typically obtained from an **IOptionsServer** interface using the **getClient()** method, and is associated with the **IOptionsServer** using the **setCurrentClient()** method as illustrated below:

```

IOptionsServer server = null;
try {
    server = getOptionsServer(null, null);
}

```

```

server.setUsername(userName);
server.login(password);

IClient client = server.getClient(clientName);
if (client != null) {
    server.setCurrentClient(client);
    // use the client in whatever way needed...
}
} catch (Exception exc) {
    // handle errors...
}

```

Note

- The use of the `setUsername` and `login` methods on the server to establish the current user and log them in, respectively.
- Unlike the `p4` command line client, there are no defaults for user and workspace. Your application must explicitly associate a workspace (an `IClient` client object) and user with the server using the `IOptionsServer.getClient` and `IOptionsServer.setCurrentClient` methods.

Exception and error handling

P4Java uses a small set of Java exceptions to signal errors that have occurred in either the Helix server as a result of issuing a specific command to the server, or in the P4Java plumbing in response to things like TCP/IP connection errors or system configuration issues. (These exceptions are not used to signal file operation problems at the individual file level — see ["Helix server file operations" on the next page](#) for details about individual file error handling.)

In general, P4Java exceptions are rooted in two different classes: the `P4JavaException` classes are intended for “normal” (that is, recoverable) errors that occur as the result of things like missing client files, a broken server connection, or an inappropriate command option; the `P4JavaError` classes are intended for more serious errors that are unlikely to be recoverable, including unintended null pointers or P4Java-internal errors. The `P4JavaException` class hierarchy is rooted in the normal `java.lang.Exception` tree, and any such exception is always declared in relevant method “throws” clauses; the `P4JavaError` classes, however, are rooted in `java.lang.Error`, and consequently do not need to be declared or explicitly caught. This allows a developer to catch all such `P4JavaErrors`, for example, in an outer loop, but to process “normal” `P4JavaExceptions` in inner blocks and loops as they occur.

Typically, application code should report a **P4JavaError** exception and then terminate either itself or whatever it was doing as soon as possible, as this exception indicates a serious error within P4Java. **P4JavaException** handling is more fine-grained and nuanced: A **P4JavaException** almost always signals a recoverable (or potentially-recoverable) error, and should be caught individually or at the class level. The following snippet represents a common pattern for **P4Java** error and exception handling around major functional blocks or processing loops:

```
try {
    // issue one or more server or client commands...
} catch (P4JavaError err) {
    panic(err); // causes app to exit after printing message to stderr...
} catch (RequestException rexc) {
    // process server-side Perforce error...
} catch (ConnectionException cexc) {
    // process Perforce connection exception...
} catch (P4JavaException exc) {
    // catchall...
} catch (Exception exc) {
    // Other-exception catchall...
}
```

Note the way **RequestException** and **ConnectionException** events are handled separately: **RequestException** exceptions are almost always thrown in response to a Helix server error message and therefore include a severity and generic code that can be used or displayed (other **P4JavaExceptions** do not usually contain these), and **ConnectionExceptions** should normally result in the enclosing app explicitly closing or at least re-trying the associated connection, as processing can no longer continue on the current Helix server connection.

Helix server file operations

To define common Helix server-managed file attributes and options, P4Java uses the **com.perforce.p4java.core.file.IFileSpec** interface. Attributes like revisions, dates, actions, and so on, are also defined in the **core.file** package, along with some key helper classes and methods. In general, most Helix server file-related methods are available on the **IOptionsServer** and **IClient** interfaces, and might also be available on other interfaces such as the **IChangelist** interface.

Because Helix server file operations can typically run to a conclusion even with errors or warnings caused by incoming arguments, and because the server usually interpolates error and informational messages in the list of file action results being returned, most file-related methods do not throw exceptions when a request error is encountered. Instead, the file-related methods return a Java list of results, which can be scanned for errors, warnings, informational messages, and the successful file specs normally returned by the server. P4Java provides helper classes and methods to detect these errors.

P4Java file methods are also designed to be composable: the valid output of one file method (for instance, `IOptionsServer.getDepotFileList`) can usually be passed directly to another file method (such as `IClient.editFiles`) as a parameter. This approach can be very convenient in complex contexts such as ant or Eclipse plug-ins, which perform extensive file list processing.

The snippet below, from the sample `ListFilesDemo` class, illustrates a very common pattern used when retrieving a list of files (in this case from the `getDepotFiles` method):

```
List<IFileSpec> fileList = server.getDepotFiles(
    FileSpecBuilder.makeFileSpecList(new String[] {"//..."}), false);
if (fileList != null) {
    for (IFileSpec fileSpec : fileList) {
        if (fileSpec != null) {
            if (fileSpec.getOpStatus() == FileSpecOpStatus.VALID) {
                System.out.println(formatFileSpec(fileSpec));
            } else {
                System.err.println(fileSpec.getStatusMessage());
            }
        }
    }
}
```

Note

The use of the `FileSpecBuilder.makeFileSpecList` helper method that converts a String array to a list of `IFileSpec` objects. The `formatFileSpec` method referenced above simply prints the depot path of the returned `IFileSpec` object if it's valid.

Summary vs. Full Objects

The 2009.2 release of P4Java introduced the notion of “summary” and “full” representations of objects on a Helix server. In many cases, the Helix server only returns summaries of objects that it's been asked to list. For example, if you issue a `p4 clients` command to a server, what comes back is a list of client metadata for known client workspaces, but not the associated workspace views. For things like changelists, jobs, branches, and so on, to obtain the full version of the Helix server object (such as a specific client workspace), you typically do a `p4 client -o` with the workspace's name.

Similarly, P4Java distinguishes between the summary objects returned from the main list methods (such as `IOptionsServer.getClients()`) and the full objects returned from individual retrieval methods (such as `IOptionsServer.getClient()`).

The snippet below, edited from the `ListClientDemo` sample app, illustrates a typical usage pattern for summary and full object retrieval:

```
try {
    IOptionsServer server = ServerFactory.getOptionsServer
("p4java://localhost:1666", null);

    server.setUserName(userName);
    server.login(password);
    System.out.println("Clients on Perforce server at URI '"
        + serverUri + "' for user '" + userName + "':");

    List<IClientSummary> clientList = server.getClients(
        new GetClientsOptions().setUserName(userName));

    if (clientList != null) {
        for (IClientSummary clientSummary : clientList) {
            // NOTE: list returns client summaries only; need to get the
            // full client to get the view:

            IClient client = server.getClient(clientSummary);
            System.out.println(client.getName() + " "
                + client.getDescription().trim() + " "
                + client.getRoot());
            ClientView clientView = client.getClientView();

            if (clientView != null) {
                for (IClientViewMapping viewMapping : clientView) {
                    System.out.println("\t\t" + viewMapping);
                }
            }
        }
    }
}
```

```

} catch (RequestException rexc) {
    System.err.println(rexc.getDisplayString());
    rexc.printStackTrace();
} catch (P4JavaException jexc) {
    System.err.println(jexc.getLocalizedMessage());
    jexc.printStackTrace();
} catch (URISyntaxException uexc) {
    System.err.println(uexc.getLocalizedMessage());
    uexc.printStackTrace();
}

```

Note

Only clients owned by *username* are returned, and that in order to print the associated client workspace view for each retrieved summary client workspace, we get the full client object. This is more common in cases where a user might iterate across a list of all workspaces known to the Helix server in order to find a specific client workspace, then retrieve that client (and only that client) workspace in full.

Using option classes

This section shows some common usage patterns you can use when coding with P4Java.

Server options

There are a variety of properties you can set for your Perforce P4Java application when acquiring an **IOptionsServer** object.

Setting your application's program name and version:

Setting your program name and version is recommended. You can see these values in your Perforce server log for commands submitted by this application.

```

IOptionsServer server =
    ServerFactory.getOptionsServer("p4java://perforce:1666", null,
        new UsageOptions(props).setProgramName
            ("JavaBuilder").setProgramVersion("1.0"));

```

Properties

See the [P4Java API Javadoc](#) for information on the [PropertyDefs](#) and [RpcPropertyDefs](#) classes.

PropertyDefs.WRITE_IN_PLACE_KEY

Setting the `PropertyDefs.WRITE_IN_PLACE_KEY` to `true` can improve sync performance to Windows platforms due to a known Java limitation.

```
Properties props = System.getProperties();
props.put(PropertyDefs.WRITE_IN_PLACE_KEY, "true");

IOptionsServer server =
    ServerFactory.getOptionsServer("p4java://perforce:1666", props, null);
```

IClient

Setting your P4CLIENT

```
IClient client = server.getClient("myclient");
server.setCurrentClient(client);
```

FileSpecBuilder

Using FileSpecBuilder to create IFileSpec Lists:

```
FileSpecBuilder.makeFileSpecList
("///...@2009/05/10:00:00,2010/05/10:05:00");

FileSpecBuilder.makeFileSpecList ("//depot/dev/...@release_12.1_label");
```

IFileSpec

Using chainable setters when using Options objects:

```
List<IFileSpec> syncFiles =
    client.sync(FileSpecBuilder.makeFileSpecList ("//depot/..."),
        new SyncOptions().setNoUpdate(true));

List<IFileSpec> openedFiles =
    server.getOpenedFiles(FileSpecBuilder.makeFileSpecList ("//..."),
        new OpenedFilesOptions().setUserName(user).setClientName(client);
```

Advanced usage notes

The following notes provide guidelines for developers using features beyond the basic usage model.

Helix server addresses, URIs, and properties

P4Java uses a URI string format to specify the network location of target Helix servers. This URI string format is described in detail in the server factory documentation, but it always includes at least the server's hostname and port number, and a scheme part that indicates a P4Java connection (for example, `p4java://localhost:1666`). Note that:

- P4Java does not obtain default values from the execution environment or other sources for any part of the URI string. All non-optional parts of the URI must be filled in. (For example, P4Java does not attempt to retrieve the value of `P4PORT` from a Unix or Linux environment to complete a URL with a missing port number.)
- P4Java's factory methods allow you to pass properties into the `IOptionsServer` object in the server's URI string as query parts that override any properties that are passed in through the normal properties parameter in the server factory `getServer` method. This feature is somewhat limited in that it doesn't currently implement URI escape sequence parsing in the query string, but it can be very convenient for properties passing. See "[P4Java properties](#)" on the next page for an explanation.

SSL connection support

Helix server at release 2012.1 or higher supports 256-bit SSL connections and trust establishment by accepting the fingerprint of the SSL certificate's public key.

Due to current US export control restrictions for some countries, the standard JDK package only comes with 128 bit encryption level cyphers. In order to use P4Java to connect to an SSL-enabled Helix server, those living in eligible countries may download the unlimited strength JCE (Java Cryptography Extension) package and replace the current default cryptography jar files with the unlimited strength files. For more information about using P4Java with an SSL enable Helix server, see <https://community.perforce.com/s/article/2620>.

To make a secure connection using P4Java, append `ssl` to the end of the P4Java protocol (for example, `p4javassl://localhost:1667`). For a new connection or a key change, you must also (re)establish trust using the `IOptionsServer`'s `addTrust` method. For example:

```
// Create a P4Java SSL connection to a secure Perforce server
try {
    String serverUri = "p4javassl://localhost:1667";
    Properties props = null;
    IOptionsServer server = ServerFactory.getOptionsServer(serverUri,
        props);

    // assume a new first time connection
    server.addTrust(new TrustOptions().setAutoAccept(true));
}
```

```
// if all goes well...

IOptionServerInfo serverInfo = server.getServerInfo();
} catch (P4JavaException e) {
    // process P4Java exception
} catch (Exception e) {
    // process other exception
}
```

The IServerResource interface

P4Java represents Helix server objects (such as changelists, branch mappings, job specs, and so on) to the end user through associated interfaces (such as **IChangelist**, **IBranchSpec**, and so on) onto objects within P4Java that mirror or proxy the server-side originals. This means that over time, the P4Java-internal instances of the objects may get out of date with the server originals, or the server originals may need to be updated with corresponding changes made to the P4Java versions.

P4Java's **IServerResource** interface is designed to support such proxying and to allow refreshes from the server or updates to the server as necessary. Virtually all useful P4Java objects or interfaces that proxy or represent Helix server-side objects extend the **IServerResource** interface, and unless otherwise noted in the individual Javadoc comments, the interface methods can be used to update server- and client-side versions accordingly.

P4Java properties

P4Java uses Java properties to set various operational values for specific **IOptionsServer** instances and/or for P4Java as a whole. These properties are typically used for things like preferred temporary file directory locations, application version and name information for Helix server usage, and the location of a suitable Helix server authentication tickets file (see "[Authentication](#)" on page 25 for details). A full list of publicly-visible properties (with default values) is given in the **PropertyDefs** Javadoc.

Properties intended for P4Java use can have "long form" or "short form" names. Long form names are canonical, and are always prefixed by the string represented by **PropertyDefs.P4JAVA_PROP_KEY_PREFIX** (normally **com.perforce.p4java.**, for example, **com.perforce.p4java.userName**). Short form names are the same string without the standard prefix (for example, **userName**). Use long form names when there's any chance of conflict with system or other properties; short form names, on the other hand, are convenient for putting property values onto URI strings as long as you know the property name won't collide with another property name in use by the app or system.

Properties can be passed to P4Java in several ways:

- As properties originally passed to the JVM using the usual Java JVM and system properties mechanisms.

Passing properties in this way is useful for fundamental P4Java-wide values that do not change over the lifetime of the P4Java invocation and that do not differ from one `IOptionsServer` instance to another. A typical example of such a property is the `com.perforce.p4java.tmpDir` property, which is used by P4Java to get the name of the temporary directory to be used for P4Java `tmp` files (and which defaults to `java.io.tmpdir` if not given).

- As properties passed in to an individual `IOptionsServer` instance through the server factory `getServer` method's properties parameter.

Properties passed in this way override properties passed in through the JVM. This mechanism is useful for any properties that are (or may be) server-specific, such as `userName`, `clientName`, and so on.

- As properties passed in through the server factory's URI string parameter query string.

Properties passed in this way override properties passed in through the properties parameter and the JVM. This mechanism is useful for ad hoc property-passing and/or overriding less-changeable properties passed in through the properties parameter.

The following code shows an example of passing properties to an `IOptionsServer` instance using the URI string query mechanism:

```
IOptionsServer server = ServerFactory.getServer(
    "p4java://test:1666?userName=test12&clientName=test12_client&"
    + "autoConnect=y", null);
```

Assuming no errors occur along the way, this code returns an `IOptionsServer` object connected to the Helix server host `test` on port 1666 with the Helix server client name `test12_client` and Helix server user name `test12` logged in automatically (note that the login only works if the underlying authentication succeeds — see ["Authentication" on page 25](#) for details).

Character Set Support

Character set support is only enabled for Unicode-enabled Helix servers. In this mode, P4Java differentiates between Helix server file content character sets (that is, the encoding used to read or write a file's contents) and the character sets used for Helix server file names, job specs, changelist descriptions, and so on.

This distinction is made due to the way Java handles strings and basic I/O: in general, while file content character set encodings need to be preserved so that the end results written to or read from the local disk are properly encoded, P4Java does not need to know about file metadata or other string value encodings. Because Helix server store and transmit all such metadata and strings in normalized UTF-8 form, and because all Java strings are inherently encoded in UTF-16, the encoding to and from non-UTF-16 character sets (such as `shiftjis`) is done externally from P4Java (usually by the surrounding app), and is not influenced by or implemented in P4Java itself. This means that the character set passed to the `IOptionsServer.setCharsetName` method is only used for translation of file content. Everything else, including all file names, job specs, changelist descriptions, and so on, is encoded in the Java-native Java string encoding UTF-16 (and may or may not need to be translated out of that coding to something like `shiftjis` or `winansi`).

P4Java supports file content operations on files encoded in most of the character sets supported by the Helix server, but not all. The list of supported Helix server file content charsets is available to calling programs through the `PerforceCharsets.getKnownCharsets` method. If you attempt to set a `IOptionsServer` object's charset to a charset not supported by both the Helix server and the local JDK installation, you will get an appropriate exception; similarly, if you try to (for example) sync a file with an unsupported character set encoding, you will also get an exception.

The Helix server uses non-standard names for several standard character sets. P4Java also uses the Helix server version of the character set, rather than the standard name.

Error Message Localization

Error messages originating from the Helix server are localized if the Helix server is localized; error messages originating in P4Java itself are not currently localized. P4Java's internal error messages aren't intended for end-user consumption as-is; your applications should process such errors into localized versions that are presentable to end users.

Logging and tracing

P4Java includes a simple logging callback feature, documented in the `ILogCallback` Javadoc page, that enables consumers to log P4Java-internal errors, warnings, informational messages, exceptions, and so on. Logging is enabled or disabled on a P4Java-wide basis, not on a per-connection or per-server basis.

The logging feature performs no message formatting or packaging. You can put the log message through the surrounding application context's logger as required. In general, your applications should log all error and exception messages. Informational messages, statistics, and warning messages do not need to be logged unless you are working with Perforce Technical Support to debug an issue.

The snippet below is taken from the `LogDemo` sample app and illustrates a typical usage pattern for `ILogCallback`:

```
public static class LogCallback implements ILogCallback {  
  
    private LogTraceLevel traceLevel = LogTraceLevel.NONE; // Don't want
```

```
tracing...
    private PrintStream outputStream = null;

    public LogCallback(PrintStream outputStream) {
        this.outputStream = outputStream;
    }

    public void setTraceLevel(LogTraceLevel traceLevel) {
        this.traceLevel = traceLevel;
    }

    public LogTraceLevel getTraceLevel() {
        return this.traceLevel;
    }

    public void internalError(String errorString) {
        if (this.outputStream != null) {

        }
    }

    public void internalException(Throwable thr) {
        if (thr != null) {
            printMessage(thr.getLocalizedMessage(), "EXCEPTION");
            thr.printStackTrace(this.outputStream);
        }
    }

    public void internalInfo(String infoString) {
        printMessage(infoString, "INFO");
    }

    public void internalStats(String statsString) {
        printMessage(statsString, "STATS");
    }
}
```

```
    public void internalTrace(LogTraceLevel traceLevel, String
traceMessage) {
        // Note: tracing does not work for normal P4Java GA releases, so
you
        // should not see any calls to this method unless you have a
"special"
        // release...

        printMessage(traceMessage, "TRACE");
    }

    public void internalWarn(String warnString) {
        printMessage(warnString, "WARNING");
    }

    private void printMessage(String msg, String pfx) {
        if (msg != null) {
            this.outStream.println(new Date() + " (" + pfx + "): " + msg);
        }
    }
}
```

Standard implementation classes

The `com.perforce.p4java.impl.generic` package is the root for a fairly large set of standard implementation classes such as `Job`, `Changelist`, and so on. These implementation classes are used internally by P4Java, and while usage is not mandatory, you are encouraged to use them as well. This is especially useful if you wish to extend standard P4Java functionality by, for example, adding audit or authentication methods to standard classes.

I/O and file metadata issues

The quality of P4Java's network and file I/O in real-world usage is strongly affected by the quality of implementation of the underlying Java NIO packages. Java's handling of file metadata also affects I/O. Although JDK 6 is an improvement over JDK 5, it can be difficult to manipulate file type and metadata (such as permissions, access/modification time, symlinks, and so on) in pure Java. These are abilities that C programmers take for granted. Issues often arise from JVM limitations such as an inability to set read-only files as writable, reset modification times, observe Unix and Linux umask values, and so on.

Because of these issues, P4Java has a file metadata helper callback scheme, defined in `com.perforce.p4java.impl.generic.sys.ISystemFileCommandsHelper`. This approach enables users to register their own file metadata class helper (typically using something like an Eclipse file helper or a set of native methods) with the server factory, to help in cases where the JDK is not sufficient. See the relevant `ISystemFileCommandsHelper` Javadoc for details.

Threading issues

P4Java is inherently thread-safe when used properly. The following best practices can help to ensure that users do not encounter thread-related problems:

- P4Java's `IOptionsServer` object is partially thread-safe. The only state preserved in the underlying implementation classes is the Helix server client that is associated with the server, and the server's authentication state.
- You can have multiple threads working against a single `IOptionsServer` object simultaneously, but note that changing authentication state (login state, password, user name, and so on) or the client that is associated with the server can have unpredictable results on long-running commands that are still running against that server object. You should ensure that changing these attributes only happens when other commands are not in progress with the particular server object.
- P4Java makes no guarantees about the order of commands sent to the Helix server by your applications. You must ensure that any required ordering is enforced.
- Using a large numbers of threads against a single `IOptionsServer` object can impose a heavy load on the JVM and the corresponding server. To control load, create your own logic for limiting thread usage. Be certain that your use of threads does not cause deadlock or blocking. Consider using a single `IOptionsServer` object for each thread.
- P4Java offers a number of useful callbacks for things like logging, file helpers, progress monitoring, and so on. These callbacks are performed within a live thread context. Ensure that any callbacks that you register or use do not cause blocking or deadlocks.
- To obtain the best resource and memory allocation strategies for your specific threading needs, experiment with JVM invocation parameters. Garbage collection and memory allocation strategies can make quite a difference in raw threading throughput and latency, but often indirectly and unpredictably.

Authentication

P4Java implements both the Helix server tickets-based authentication and the Helix server single sign on (SSO) feature. Both types of authentication are described in detail in the P4Java Javadoc, but some P4Java-specific issues to note include:

- P4Java manages a `p4 tickets` file in a manner similar to that of the P4 command line (under normal circumstances, the two can share the same tickets file). When a ticket value is requested by the Helix server and the current ticket value in the associated `IOptionsServer` object is not set, an attempt is made to retrieve the ticket out of the `p4 tickets` file. If found, the ticket is stored on the `IOptionsServer` object and used as the Helix server authentication ticket.

A successful login causes the ticket value to be added or updated in the tickets file, and a logout causes the current ticket value in the `p4 tickets` file to be removed. The `IOptionsServer` object's ticket should be set to `null` to cause a re-reading of the ticket value from the `p4 tickets` file.

The `p4 tickets` file is usually stored in the same place the `p4` command line stores it, but the `PropertyDefs.TICKET_PATH_KEY` property can be used to specify an alternate tickets file.

- P4Java implements Helix server's SSO scheme using a callback interface described in the `ISSOCallback` Javadoc (in the package `com.perforce.p4java.server.callback`). Ensure that the callback doesn't block, and that it adheres to the expected format of the associated Helix server.

Other Notes

- As documented in the main Helix server documentation, Helix server form triggers can cause additional output on form commands such as "change" or "client", even when the trigger succeeds. This trigger output is available through the P4Java command callback feature, but note that there is currently no way to differentiate trigger output from normal command output, and that such trigger output will also be prepended or appended to normal string output on commands such as `IOptionsServer.newLabel`.
- P4Java's command callback feature, documented in class `com.perforce.p4java.server.callback.ICommandCallback`, is a useful way to get blow-by-blow command status messages and trigger output messages from the server in a way that can mimic the `p4` command line client's output. Usage is straightforward, but note the potential for deadlocks and blocking if you are not careful with callback method implementation.
- P4Java's progress callback feature gives users a somewhat impressionistic measure of command progress for longer-running commands. Progress callbacks are documented in the Javadoc for class `com.perforce.p4java.server.callback.IProgressCallback`. Once again, if you use this feature, ensure that your callback implementations do not cause deadlocks or blocking.

- We strongly recommend setting the `progName` and `progVersion` properties (either globally or for each `IOptionsServer` instance) whenever you use P4Java. Set these values to something meaningful that reflects the application or tool in which P4Java is embedded; this can help Helix server administrators and application debugging.

For example, the following code sets `progName` and `progVersion` via the JVM invocation property flags:

```
$ java -Dcom.perforce.p4java.programName=p4test
      -Dcom.perforce.p4java.programVersion=2.01A ...
```

Alternatively, you can also use the server factory `getServer` method's properties parameter:

```
Properties props = new Properties(System.getProperties());
props.setProperty(PropertyDefs.PROG_NAME_KEY, "ant-test");
props.setProperty(PropertyDefs.PROG_VERSION_KEY, "Alpha 0.9d");

...

server = IOptionServerFactory.getServer(serverUriString, props);
```

- If your application receives a `ConnectionException` from an `IOptionsServer` or `IClient` method while communicating with a Helix server, the only truly safe action is to close the connection and start over with a new connection, rather than continue using the connection.

A `ConnectionException` event typically represents a serious network error (such as the Helix server unexpectedly closing a connection or a bad checksum in a network packet), and there's no guarantee that after receiving such an event the connection is even usable, let alone reliable.

- There is currently no `diff` method on `IFileSpec` interfaces to compare versions of the same Helix server-managed file, but this functionality may be easily implemented with a combination of `IOptionsServer.getFileContents` to retrieve the contents of specific versions to temporary files, and the use of the operating system's `diff` application on these temporary files as shown below:

```
InputStream fspecStream1 = server.getFileContents(
    FileSpecBuilder.makeFileSpecList(
        new String[] {spec1}), false, true);
InputStream fspecStream2 = server.getFileContents(
    FileSpecBuilder.makeFileSpecList(
        new String[] {spec2}), false, true);

File file1 = null;
```

```
File file2 = null;

try {
    file1 = File.createTempFile("p4jdiff", ".tmp");
    file2 = File.createTempFile("p4jdiff", ".tmp");
    FileOutputStream outputStream1 = new FileOutputStream(file1);
    FileOutputStream outputStream2 = new FileOutputStream(file2);
    byte[] bytes = new byte[1024];
    int bytesRead = 0;
    while (bytesRead = fspecStream1.read(bytes > 0) > 0) {
        outputStream1.write(bytes, 0, bytesRead);
    }
    fspecStream1.close();
    outputStream1.close();
    while (bytesRead = fspecStream2.read(bytes > 0) > 0) {
        outputStream2.write(bytes, 0, bytesRead);
    }
    fspecStream2.close();
    outputStream2.close();
    Process diffProc = Runtime.getRuntime().exec(new String[] {
        "/usr/bin/diff", file1.getPath(), file2.getPath() });
    diffProc.waitFor();
    if (diffProc != null) {
        InputStream inputStream = diffProc.getInputStream();
        byte[] inBytes = new byte[1024];
        int inBytesRead = 0;
        while (inBytesRead = inputStream.read(inBytes > 0) > 0) {
            System.out.write(inBytes, 0, inBytesRead);
        }
    }
} catch (Exception exc) {
    error("diff error: " + exc.getLocalizedMessage());
    return;
} finally {
```

```
if (file1 != null) file1.delete();  
if (file2 != null) file2.delete();  
}
```

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (versioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

For files: The file revision, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision. For checked out streams: The public have version from which the checked out version is derived.

binary file type

A Helix server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix server and is able to run scripts on the commands before sending them to the Helix server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction and changelist number.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix server depot.

check out

To designate one or more files, or a stream, for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix server files that is not streams-based. Uses the Perforce file revision model, not the graph model. The default depot name is depot. See also default depot, stream depot, and graph depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix server, a file with its head revision marked as deleted. Older revisions of the file are still available. In Helix server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Core server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix server syntax for specifying the location of files in the depot. Depot syntax begins with: `//depot/`

diff

(noun) A set of lines that do not match when two files, or stream versions, are compared. A conflict is a pair of unequal diffs between each of two files and a base, or between two versions of a stream.
(verb) To compare the contents of files or file revisions, or of stream versions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

extension

Similar to a trigger, but more modern. See "Helix Core Server Administrator Guide" on "Extensions".

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicate servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix server. See also Helix4Git and classic depot.

group

A feature in Helix server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

heartbeat

A process that allows one server to monitor another server, such as a standby server monitoring the master server (see the p4 heartbeat command).

Helix server

The Helix server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core server.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix server depots.

hybrid workspace

A workspace that maps to files stored in a depot of the classic Perforce file revision model as well as to files stored in a repo of the graph model associated with git.

I**iconv**

A PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J**job**

A user-defined unit of work tracked by Helix server. The job template determines what information is tracked. The template can be modified by the Helix server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job daemon

A program that checks the Helix server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

A form describing the fields and possible values for each job stored in the Helix server machine.

job view

A syntax used for searching Helix server jobs.

journal

A file containing a record of every change made to the Helix server's metadata since the time of the last checkpoint. This file grows as each Helix server transaction is logged. The file should be automatically truncated and renamed into a numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix server's metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix server from two conflicting file revisions.

metadata

The data stored by the Helix server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata is stored in the Perforce database and is separate from the archive files that users submit.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix server has assigned a number.

O

opened file

A file you have checked out in your client workspace as a result of a Helix Core server operation (such as an edit, add, delete, integrate). Opening a file from your operating system file browser is not tracked by Helix Core server.

owner

The Helix server user who created a particular client, branch, or label.

P

p4

1. The Helix Core server command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix server; p4d manages depot files and metadata.

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

Perforce

Perforce Software, Inc., a leading provider of enterprise-scale software solutions to technology developers and development operations (“DevOps”) teams requiring productivity, visibility, and scale during all phases of the development lifecycle.

project

In Helix Swarm, a group of Helix server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix server’s protections table.

proxy server

A Helix server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix server but not make any changes.

remote depot

A depot located on another Helix server accessed by the current Helix server.

replica

A Helix server that contains a full or partial copy of metadata from a master Helix server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file, or two versions of a stream. You can choose to resolve file conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes. To resolve stream conflicts, you can choose to accept the public source, accept the checked out target, manually accept changes, or combine path fields of the public and checked out version while accepting all other changes made in the checked out version.

reverse delta storage

The method that Helix server uses to store revisions of text files. Helix server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

review daemon

A program that periodically checks the Helix server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files included in those changelists, informing them of changes in files they are interested in.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

RPM

RPM Package Manager. A tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

The combination of server metadata (the Helix server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Core server, the shared versioning service that responds to requests from Helix server client applications. The Helix server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

storage record

An entry within the db.storage table to track references to an archive file.

stream

A "branch" with built-in rules that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients. Has structured branching, unlike the free-form branching of a "classic" depot. Uses the Perforce file revision model, not the graph model. See also classic depot and graph depot.

submit

To send a pending changelist into the Helix server depot for processing.

super access

An access level that gives the user permission to run every Helix server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script that is automatically invoked by Helix server when various conditions are met. (See "Helix Core Server Administrator Guide" on "Triggers".)

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix server in which you assign file types to files.

U

user

The identifier that Helix server uses to determine who is performing an operation.

V

versioned file

Source files stored in the Helix server depot, including one or more revisions. Also known as an archive file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

Perforce programs include software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.openssl.org/>).