Developing Applications Using the BigIndex API (Draft, December 9, 2014)

Contents

Chapter 1. Introduction to the BigIndex API

Introducing a high-performance, enterprise-ready Application Programming Interface (API) for 360-degree information applications and their infrastructure.

This chapter introducs the Watson™ Explorer BigIndex API. It begins by describing the motivation for developing applications with the API. It then provides a brief overview of the API, as well as the distributed repository it uses to manage its configuration data. It concludes with an overview of the document's organization and the typographical conventions and approach the document takes to describing example code.

Enterprise Requirements for Watson Explorer Applications

The web-based nature of IBM® Watson Explorer Engine and IBM Watson Explorer Application Builder applications liberates them from many of the limitations of traditional enterprise applications, such as local installation and system requirements or of being available only from a single server. The ease of access to web-based applications through any standards-compliant browser makes it even more important that the data resources required by web-based applications are always available. Increased availability is usually facilitated by eliminating single points of failure that can prevent successful application execution and operation, which is typically achieved by replicating application-specific configuration information and data on multiple networked systems.

Replicating information across multiple hosts provides opportunities for both reliability and performance improvements by distributing application load among multiple systems. Replicating configuration data across multiple systems eliminates configuration data as a single point of failure when balancing front-end application load across distributed systems. Replicating application data across multiple systems eliminates application data as a single point of failure when balancing back-end data access requirements across the systems.

Data replication and front- and backend load balancing are common means of ensuring that applications are always available and can scale to satisfy increasing numbers of users. But enterprise applications must also support continually increasing amounts of data and associated storage. Although IBM Watson Explorer indices are typically much smaller than the data repository they index, employing a flexible model for allocating and distributing the backend storage associated with a Watson Explorer Engine or Watson Explorer Application Builder application both eliminates many common storage problems and simplifies the management of that data if it must be redistributed, expanded, or moved to new or additional storage devices or systems.

Thus, being able to replicate and distribute indices across multiple hosts is a fundamental requirement for the reliability and availability expectations of enterprise applications such as those created with Watson Explorer Application Builder. But the data repository indices and the mechanisms used to configure and access them have historically been managed manually by using the Watson Explorer Engine administration tool. Unfortunately, requiring manual intervention is not scalable.

Introducing the BigIndex API

Watson Explorer BigIndex provides an enterprise-caliber API for the programmatic creation, distribution, and management of indices and associated storage. This API hides the complexity that can be associated with creating indices, querying those indices or other content sources, and returning the results of queries against those indices. The BigIndex API also hides the complexity of replicating Watson Explorer Engine indices across multiple servers, providing opportunities for automating load and storage balancing across participating servers, automating fault-tolerance, and migrating indices between servers.

Applications that require these capabilities can simply leverage and configure them via the BigIndex API, eliminating the countless hours otherwise needed to develop and test them for each application. The BigIndex API reduces code complexity and makes working with complex distributed systems as easy as working with a local client API.

To support enterprise application requirements such as high availability and load balancing, indices created with the BigIndex API are typically partitioned into a number of segments known as *shards*, which can be replicated across the Watson Explorer Engine servers associated with a given index. Segmenting indices into shards, tuning their size, and distributing them across servers also provides opportunities for reducing the footprint and memory requirements of an index on any given host. With the BigIndex API, the servers associated with an index are identified in a cluster-collection-store entity in the application's entity model. This entity definition also includes attributes that specify the number of shards that are associated with the index and how the index is used.

Applications built with the BigIndex API can follow either a programmatic or a configuration-driven model to working with entities. The programmatic approach is simpler in terms of the setup required for configuration, but all entities and data stores must be established via the API. The configuration-driven approach relies on external XML-based configuration model files to define its entities and configuration, a more powerful approach that is also more appropriate for a production environment. This document introduces the API and the concepts of indexing and searching via the programmatic model but quickly moves to the more robust configuration model to explain the API.

The BigIndex API was developed in conjunction with Application Builder but makes it easy for other applications to incorporate the data exploration, navigation, and search capabilities that Watson Explorer provides.

Introducing ZooKeeper

BigIndex and Application Builder applications store configuration data such as their entity model in a networked data repository enabled by Apache Zookeeper. Zookeeper is an open source data repository designed to provide a highly reliable, synchronized repository for configuration data such as that required by large-scale distributed systems. Application-specific configuration data is stored in distinct namespaces, enabling a single Zookeeper installation to simultaneously support the configuration requirements of multiple applications.

Zookeeper satisfies its reliability and availability requirements by coordinating its content across multiple systems that are running a Zookeeper server, are associated with each other, and support the same namespace for configuration data. A single

Zookeeper server is often used when developing Application Builder applications, but three or more Zookeeper nodes are commonly used in production.

A set of Zookeeper nodes that interact with each other are known as an *ensemble*. Configuring an odd number of Zookeeper servers enables a subset of related servers to use majority voting to select one member of the ensemble as the authoritative master node. Updates to any member of the ensemble are synchronized with the master, which then updates all other servers appropriately. Production environments often use at least five Zookeeper nodes to prevent system or network maintenance from affecting overall availability or reliability.

Document Organization

This document includes the following chapters:

- Chapter 2, “Installing and Configuring Related Modules,” on page 5 discusses the installation and configuration of software modules required to use the BigIndex API, relying on other documents for some of the steps where possible.
- Chapter 3, “Hello, BigIndex,” on page 7 introduces basic indexing and searching via the first two example applications delivered with the BigIndex API. The chapter describes the examples in detail to provide an overview of the API and its use. The examples in this and the following two chapters are driven programmatically to avoid complexity when first learning the API.
- Chapter 4, “Basic Data Indexing,” on page 23 introduces the basic concept of indexing with the API. It describes the fundamental entity model, records, record sections, and record schemas, and it introducs the concepts of indexers and their options.
- Chapter 5, “Basic Data Searching,” on page 51 builds on the previous chapter to introduce the concept of searching. It describes basic searching and the different types of searches you can perform based on how your data is indexed.
- Chapter 6, “Application Configuration Management,” on page 53 focuses on configuration-driven use of the API via external XML files. Whereas the previous chapters focus on basic capabilities, this chapter introduces the more powerful approach to using the API that you will want to adopt for production deployments.
- Chapter 7, “Deleting Data from an Index,” on page 55 discusses how to delete information from an index. The topic is introduced here to enable the use of external configuration files in the examples that demonstrate deletion.
- Chapter 8, “Advanced Topics,” on page 57 describes indexing and searching topics that are more advanced than those introduced earlier, such as field options, binary-encoded data, tokenization, security, and others. Note that while these topics are more advanced than those discussed in earlier chapters, they still represent common functionality you will want to learn and apply.
- Chapter 9, “Example Code,” on page 59 lists and briefly describes the example applications provided with the API. It also describes how to work with the example code and provides a brief overview of the supporting utilities the examples rely upon.
- Chapter 10, “API Reference Summary,” on page 65 explains some basic aspects of the API and its design. It then lists and briefly describes the various packages delivered with the API. It does not replace the javadoc reference documentation installed with the API.

Note: This document is still under development, so the chapter outline is preliminary and subject to change. For example, the chapter that discusses advanced topics may be further decomposed into separate chapters for more involved concepts, or some material could be merged into earlier chapters.

Documentation Conventions

This document adopts the following conventions for describing the API, example code, and commands:

- The document uses the following typefaces to indicate various elements of code and commands: `classes`, `methods`, `keywords`, `example code`, `output`, and `pathnames` are shown in monospace; *variables* are shown in italics; and **commands** and **elements of graphical user interfaces** are shown in bold.
- With few exceptions, the document does not include the full source code for the example applications. Rather, it describes snippets of the individual examples in detail to help you understand the code in a more fine-grained fashion. Refer to the source files for the examples to see the complete code in the full context of the example applications.
- With few exceptions, the document does not show `import` statements used to include required packages and classes. Refer to the source files for the individual examples to determine the packages and classes on which they rely.
- The document uses line breaks and general formatting that do not always match the structure of the corresponding source files. Differences exist for presentation purposes only. Both the code in the source files and that shown in the documentation is valid and compiles and runs as shown.
- The document frequently uses ellipses to elide aspects of example code not germane to the current discussion. For example, descriptions may omit intervening code that does not pertain directly to the operations being discussed.
- The document often omits `try/catch/finally` clauses because they are standard coding practice that do not contribute to your understanding of the code. In some cases, the text mentions the existence of such clauses to emphasize their importance. Refer to the source files for the individual examples to determine how and where the code uses these clauses.

Chapter 2. Installing and Configuring Related Modules

Applications that use the Watson Explorer BigIndex API use Watson Explorer Engine as the data store for all of the content that is ingested through the API. To use the API, you must install and configure Watson Explorer Engine and have it running on all of the systems in your search cluster. When installed for use by the API, instances of Watson Explorer Engine typically use their embedded web server.

Applications that use the BigIndex API also use Apache Zookeeper to maintain configuration information for each cluster. This includes information about the instances of Watson Explorer Engine installed in the cluster and schema information for the indices used by the applications.

For information about installing Watson Explorer Engine, Zookeeper, and the BigIndex API itself, see the Watson Explorer Engine Installation and Integration Guide. After you have installed the modules, refer to this chapter to learn how to configure the modules for use during application development and when running the applications in production.

Configuring Watson Explorer Engine Instances

This section under development

Configuring Apache Zookeeper

This section under development

Configuring the BigIndex API

This section under development

Chapter 3. Hello, BigIndex

This chapter introduces application development with the BigIndex Application Programming Interface (API). It discusses the first two simple examples provided with the BigIndex API. “The HelloWorld Code” introduces the `HelloWorld.java` example, which indexes and searches for a simple string. “HelloWorld Redux: The ExampleApplication Code” on page 12 then describes the `ExampleApplication.java` code, which indexes and searches for slightly more complex data. The two examples provide a gentle introduction to the programming model used with the BigIndex API. The chapter provides only an introduction to the classes and methods used by the examples, including just enough information to help you understand what the examples are doing; later chapters provide more detailed discussions of the various components of the API.

“Supporting Utilities” on page 19 introduces the example utilities provided with the API to make it easier for you to work with the example applications. Understanding this section is important for using both the examples introduced in this chapter and those described in the remainder of this document.

The examples and utilities described in this chapter, like all examples described in this document, are located in the directory `bigindex/examples` of the BigIndex installation. Chapter 9, “Example Code,” on page 59 lists and briefly describes all of the available example applications. That chapter also provides information about working with the examples, including how to build and run them in the Eclipse Integrated Development Environment.

The HelloWorld Code

The canonical first example application provided with the BigIndex API is named `HelloWorld.java`. The Hello World example application indexes the simple string "Hello World" and verifies that it appears in the index. In addition to providing a very basic introduction to programming with the API, the example lets you ensure that your Watson Explorer Engine and BigIndex installation and configuration are correct.

This section shows the complete source code for the Hello World example application. The following sections describe the example in two logical phases: “Indexing” on page 9 and “Searching” on page 10. The division is somewhat arbitrary, but the separate sections make the code and explanations easier to follow. Note that the discussion of the code remains at a high level; deeper discussions of the interface are reserved for later chapters. (The code that follows does not show the `import` statements that include required packages and classes; refer to the `HelloWorld.java` source file to view the `import` statements.)

Note: The `HelloWorld` class begins by defining three static strings that are used by the Watson Explorer Engine configuration and embedded Zookeeper server relied upon by the example. Similarly, the `main` method of the example begins by preparing this configuration and embedded server. This code is not described in the following sections discussing the example code, an approach followed for all remaining examples in this document. See “Supporting Utilities” on page 19 for a detailed discussion of how these supporting utilities are used with the BigIndex example applications.

```
public class HelloWorld {
    public static final String DATA_EXPLORER_ENDPOINT_URL =
          "http://localhost:9080/vivisimo/cgi-bin/velocity.exe"
          + "?v.app=api-soap&wsdl=1&use-types=true";
    public static final String DATA_EXPLORER_ENDPOINT_USERNAME = "data-explorer-admin";
    public static final String DATA_EXPLORER_ENDPOINT_PASSWORD = "TH1nk1710";
    public static final String HELLO_WORLD_COLLECTION_NAME = "hello-world-test-collection";
    public static final String HELLO_WORLD_ENTITY_TYPE = "hw_entity_type";
    public static final String MESSAGE_FIELD = "message";
    public static final String RECORD_ID = "RECORD_ID";

    public static void main(String[] args) throws IndexerInterruptedException,
        ParsingException, SchemaViolationException {
        Configuration configuration = generateConfiguration();

        ExampleEmbeddedZookeeperServer embeddedZkServer =
            new ExampleEmbeddedZookeeperServer.Builder(configuration).build();
        ZookeeperBigIndexConfigurationProvider configurationProvider = null;
        try {
            embeddedZkServer.startEmbeddedServer();
            ZookeeperConfiguration zkConfig = embeddedZkServer.getZookeeperConfiguration();
            configurationProvider = new ZookeeperBigIndexConfigurationProvider(zkConfig);

            RecordSchema recordSchema =
                new RecordSchemaBuilder().addRecordType(HELLO_WORLD_ENTITY_TYPE)
                .addTextField(MESSAGE_FIELD).retrievable(true).build();
            RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
            IndexerOptions options = new IndexerOptions.Builder().build();
            Indexer indexer = new DataIndexer(configurationProvider, options);
            try {

                RecordBuilder logRecordBuilder =
                    recordBuilderFactory.newRecordBuilder(HELLO_WORLD_ENTITY_TYPE);
                Record logRecord =
                    logRecordBuilder.id(RECORD_ID).addField(MESSAGE_FIELD, "Hello World").build();

                RequestStatus status = indexer.addOrUpdateRecord(logRecord);

                try {
                    indexer.waitForAllIndexingToComplete();
                } catch (IndexerInterruptedException e) {
                    e.printStackTrace();
                    System.exit(1);
                }

                if (status.getState() == State.ERROR) {
                    status.getIndexerError().getIndexerException().printStackTrace();
                } else {
                    System.out.println("success! Done Indexing Data...");
                }
            } finally {
                indexer.close();
            }
            EntityResolver er = null;
            try {
                er = new EntityResolver(configurationProvider, new SearchOptions.Builder().build());

                ResultSet results = er.retrieve(searchFor(entitiesOfType(HELLO_WORLD_ENTITY_TYPE)));
                for (Entity e : results) {
                    System.out.println(e.getEntityKey().getEntityType()
                        + " -> " + e.getEntityKey().getEntityId());
                    System.out.println("\t"
                        + e.getField(new FieldName(MESSAGE_FIELD)).getFieldName()
                        + " --> " + e.getField(new FieldName(MESSAGE_FIELD)).getFieldValues());
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (er != null) {
                    er.close();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (configurationProvider != null) {
                configurationProvider.close();
            }
            if (embeddedZkServer != null) {
                embeddedZkServer.close();
            }
        }
    }

    public static Configuration generateConfiguration() {
        return new ExampleBigIndexConfigurationGenerator.Builder
            (DATA_EXPLORER_ENDPOINT_URL, DATA_EXPLORER_ENDPOINT_USERNAME,
            DATA_EXPLORER_ENDPOINT_PASSWORD, HELLO_WORLD_COLLECTION_NAME,
            HELLO_WORLD_ENTITY_TYPE).build().getGeneratedConfiguration();
    }
}
```

Indexing

The `HelloWorld` class begins by defining a number of static string variables used to define required elements of the example. Some of these static variables are used by the supporting utilities described elsewhere. Among those necessary to understand the indexing and searching functionality of the example are the following:

```
public static final String HELLO_WORLD_COLLECTION_NAME = "hello-world-test-collection";
public static final String HELLO_WORLD_ENTITY_TYPE = "hw_entity_type";
public static final String MESSAGE_FIELD = "message";
public static final String RECORD_ID = "RECORD_ID";
```

The simple Hello World example uses a single Watson Explorer Engine collection named `hello-world-test-collection`, which is configured for the example by the utility class `ExampleBigIndexConfigurationGenerator`. All data for the example resides in this collection. The example uses a single entity type, `hw_entity_type`, to store indexed data in this collection; `Entity` is a core data class of the BigIndex API that defines the type of a data record stored in the index. Like the `hello-world-test-collection`, the `hw_entity_type` is configured for the example by the `ExampleBigIndexConfigurationGenerator` class. The *MESSAGE_FIELD* and *RECORD_ID* variables are described below.

The code to create the actual index begins by creating a record schema for the data to be indexed via a call to the `build` method of the `RecordSchemaBuilder` class. A record schema is comparable to a database schema; it defines the fields and attributes for a type of `Record`, which is another core data class of the BigIndex API. In this case, the schema is created for the record type specified by the *HELLO_WORLD_ENTITY_TYPE* variable. The record includes a single data field, a text field named `message`, as specified by the *MESSAGE_FIELD* variable. The text field is made retrievable so that it is copied into the index and can later be presented to the user. The field is searchable by default, meaning that it can be looked for by a search operation.

A `RecordBuilderFactory`, which produces `RecordBuilder` objects that create the actual records stored in the index for the application, is instantiated by a call to that class's constructor. The *recordSchema* instance is passed to the constructor to indicate the schema to adhere to when creating record builders.

```
RecordSchema recordSchema = new RecordSchemaBuilder()
     .addRecordType(HELLO_WORLD_ENTITY_TYPE)
     .addTextField(MESSAGE_FIELD).retrievable(true).build();
RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
```

Actual indexing of data for records is performed by an instance of the `DataIndexer` class named *indexer*, which implements the `Indexer` interface. The constructor used to create the data indexer in this example requires two arguments. The first identifies the configuration provider, which is supplied by the embedded Zookeeper server (see "Supporting Utilities" on page 19). The second indicates the options to be used by the indexer; these are defined by the call to the `build` method of the `IndexerOptions.Builder` class. For this simple example, the resulting instance of the `IndexerOptions` class uses only default options.

```
IndexerOptions options = new IndexerOptions.Builder().build();
Indexer indexer = new DataIndexer(configurationProvider, options);
```

Within a nested `try` block, the code now completes final preparation and proceeds with the actual indexing. The instance of a `RecordBuilderFactory` created previously is used to create a `RecordBuilder` named *logRecordBuilder* that in turn creates a record for the *HELLO_WORLD_ENTITY_TYPE*, `hw_entity_type`. The `build` method of the new `RecordBuilder` creates a single record, *logRecord*. This record has an ID with the simple value `RECORD_ID` (as specified by the variable *RECORD_ID*) and a text field named `message` (as specified by the variable *MESSAGE_FIELD*) that contains the simple string `Hello World`.

```
RecordBuilder logRecordBuilder =
     recordBuilderFactory.newRecordBuilder(HELLO_WORLD_ENTITY_TYPE);
Record logRecord = logRecordBuilder.id(RECORD_ID)
     .addField(MESSAGE_FIELD, "Hello World").build();
```

The `addOrUpdateRecord` method of the `DataIndexer` created previously is then called to index the record and add it to the collection. The method adds the record to the collection because the record does not already exist; otherwise, it would update an existing record. The asynchronous return status of the method is captured by the *status* object of the `RequestStatus` type, which is later checked to determine whether the operation succeeded.

The `waitForAllIndexingToComplete` method of the `DataIndexer` object is called to block indefinitely until the indexing operation completes. (A second version of the overloaded method lets you specify a timeout for the operation.) The method is called within a `try/catch` block to respond appropriately if an `IndexerInterruptedException` is raised while waiting.

```
RequestStatus status = indexer.addOrUpdateRecord(logRecord);

try {
     indexer.waitForAllIndexingToComplete();
} catch (IndexerInterruptedException e) {
     e.printStackTrace();
     System.exit(1);
}
```

The `getState` method of the `RequestStatus` object is now called to check the state of the indexing operation. If the returned state equals `State.ERROR`, the `getIndexError` method of the `RequestStatus` object is called to obtain the actual `IndexerError` for the operation, the `getIndexerException` method is called on the error object, and a stack trace is displayed. If the `getState` method does not indicate that an error occurred, the code displays a simple success message.

```
if (status.getState() == State.ERROR) {
     status.getIndexerError().getIndexerException().printStackTrace();
} else {
     System.out.println("success! Done Indexing Data...");
}
```

The indexing operation is concluded by a call to the `close` method of the `DataIndexer` object, which shuts down the `DataIndexer` and all associated threads. The call is enclosed in a `finally` clause, since applications must ensure that it is always called to guarantee that all resources are freed.

```
indexer.close();
```

Searching

The code to search for and retrieve the indexed data begins with creation of an `EntityResolver`, which retrieves one or more indexed entities from a collection. The *er* variable is initialized and then instantiated by a call to the constructor of the

EntityResolver class, which requires two arguments. The first identifies the configuration provider, which for this example application is supplied by the embedded Zookeeper server (see “Supporting Utilities” on page 19). The second is an instance of the SearchOptions class, which is the configuration container for all search options for an EntityResolver. The anonymous SearchOptions instance is instantiated via a call to the build method of the SearchOptions.Builder class.

```
EntityResolver er = null;
...
er = new EntityResolver (configurationProvider, new SearchOptions.Builder().build());
```

The EntityResolver can now be used to search for entities in the index. The retrieve method of the class is used to search for indexed entities of the desired type. The version of the overloaded retrieve method used in this example returns an instance of a SearchResultSet. The argument to the method calls the static searchFor method of the RequestBuilders class, which supplies a rich collection of methods to enable search requests of many different types. The argument to that method is in turn a call to another static method of the RequestBuilders class, entitiesOfType, which searches for entities only of the type specified. The method accepts multiple entity types, but only a single type, *HELLO_WORLD_ENTITY_TYPE*, is needed for this simple example.

The entitiesOfType method returns an instance of an EntityTypeRequest object, which allows for EntityType granularity when searching for types via a search request. This instance is the argument type for the searchFor method, which returns a SearchRequest object. And this is the argument type for the retrieve method, which returns a SearchResultSet object to the ultimate caller. This result is captured in the *results* object, which is of type ResultSet, the interface implemented by the SearchResultSet class.

```
ResultSet results = er.retrieve(searchFor(entitiesOfType(HELLO_WORLD_ENTITY_TYPE)));
```

Now that all entities of the desired type have been retrieved (in the case of this example, just one entity is retrieved), the code iterates through the set of search results in *results* to display them to the end user by various calls to methods of the Entity class. The first line of code gets the entity key for the returned entity and, from that, the type of the entity; it then gets the ID of the entity, again based on the entity key. The second line of code gets the field of the entity and from that the field name; it then gets the value of the field. For the field-related values, the code needs to instantiate an object of type FieldName to pass to the getField method of the Entity class, after which the getFieldName and getFieldValues methods of the Field class are called to get the actual values to be displayed.

```
for (Entity e : results) {
     System.out.println(e.getEntityKey().getEntityType()
          + " -> " + e.getEntityKey().getEntityId());
     System.out.println("\t"
          + e.getField(new FieldName(MESSAGE_FIELD)).getFieldName()
          + " --> " + e.getField(new FieldName(MESSAGE_FIELD)).getFieldValues());
}
```

The close method of the EntityResolver object is then called to close any services used in the search operation. The call is made from a finally clause to ensure that resources are properly freed. The test to ensure that *er* is not null ensures that the code closes the EntityResolver only if it was successfully created.

```
if (er != null) {
     er.close();
}
```

The following example output shows the results displayed when the application is run. The success message is displayed at conclusion of the index operation, as described in the previous section; the details about the retrieved entity and its field are produced by the search operation. The full internal names and hexadecimal identifiers of the various elements are shown, along with the string identifiers specified by the code, such as the entity type (`hw_entity_type`), record ID (`RECORD_ID`), field name (`message`), and field value (`Hello World`). For this example, the field includes no bold ranges, a concept explained later in this document.

```
success! Done Indexing Data...
com.ibm.dataexplorer.bigindex.search.model.EntityType@3d4745f3[entityType=hw_entity_type]
-> com.ibm.dataexplorer.bigindex.search.model.EntityId@74219071
   [entityId=9b70d0aa90a98bbb3ee7f4aab8905f96,bigIndexRecordId=RECORD_ID]
        com.ibm.dataexplorer.bigindex.search.model.FieldName@e9434690[fieldName=message]
        --> [com.ibm.dataexplorer.bigindex.search.model.FieldValue@bd843f78
            [fieldValue=Hello World,boldRanges=[]]]
```

HelloWorld Redux: The ExampleApplication Code

The second example application provided with the BigIndex API is named `ExampleApplication.java`. Like the Hello World example, this example indexes some data and verifies that it appears in the index. However, this example indexes slightly more complex data to demonstrate additional aspects of the API, and it uses different techniques for working with the API. A more granular approach is therefore taken to describing this example code.

This section begins by showing the complete source code for the Example Application. The following sections then describe the example in more detail: “Preparing to Index” on page 14, “Indexing” on page 16, “Handling Indexing Errors” on page 17, and “Searching” on page 18. Because the Example Application is similar to the Hello World example in many ways, the text does not generally duplicate explanations of basic operations described with the preceding example; refer to “The HelloWorld Code” on page 7 for more information about code that is common to both examples.

Note: Once again, the example code begins by defining three static variables and the `main` method begins by preparing the Watson Explorer Engine configuration and embedded Zookeeper server used with the example. This code is not discussed in the following sections that describe the example code; see “Supporting Utilities” on page 19 for more information about how these supporting utilities are used with the BigIndex examples.

```
public class ExampleApplication {
    public static final String DATA_EXPLORER_ENDPOINT_URL =
        "http://localhost:9080/vivisimo/cgi-bin/velocity.exe"
        + "?v.app=api-soap&wsdl=1&use-types=true";
    public static final String DATA_EXPLORER_ENDPOINT_USERNAME = "data-explorer-admin";
    public static final String DATA_EXPLORER_ENDPOINT_PASSWORD = "TH1nk1710";
    public static final String EXAMPLE_APPLICATION_COLLECTION_NAME =
        "example-application-test-collection";
    public static final String EXAMPLE_APPLICATION_ENTITY_TYPE = "ex_entity_type";
    public static final String ID_FIELD = "id";
    public static final String FLAG_FIELD = "flag";
    public static final String TYPE_FIELD = "type";
    public static final String MESSAGE_FIELD = "message";
    public static final String DATE_FIELD = "date";
    private static Calendar calendar = Calendar.getInstance();
    private static SimpleDateFormat dateFormatter = new SimpleDateFormat("MM/dd/yyyy");

    public static void main(String[] args) {
        Configuration configuration = generateConfiguration();

        ExampleEmbeddedZookeeperServer embeddedZkServer =
            new ExampleEmbeddedZookeeperServer.Builder(configuration).build();
        ZookeeperBigIndexConfigurationProvider configurationProvider = null;
        try {
            embeddedZkServer.startEmbeddedServer();
            ZookeeperConfiguration zkConfig = embeddedZkServer.getZookeeperConfiguration();
            configurationProvider = new ZookeeperBigIndexConfigurationProvider(
                zkConfig);

        RecordSchema recordSchema = setupRecordSchema();
        RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
        IndexerOptions options = new IndexerOptions.Builder().build();
        Indexer indexer = new DataIndexer(configurationProvider, options);

        try {
            RecordBuilder logRecordBuilder =
                recordBuilderFactory.newRecordBuilder(EXAMPLE_APPLICATION_ENTITY_TYPE);

            List<RequestStatus> statuses = new ArrayList<RequestStatus>();
            List<Map<String, String>> records = new ArrayList<Map<String, String>>();
            for (int recordId = 0; recordId < 100; recordId++) {
                records.add(generateKeyValueHash(Integer.toString(recordId)));
            }

            for (Map<String, String> record : records) {
                logRecordBuilder = logRecordBuilder.id(record.get(ID_FIELD));
                for (String fieldName : record.keySet()) {
                    if (!fieldName.equals(ID_FIELD)) {
                        logRecordBuilder =
                            logRecordBuilder.addField(fieldName, record.get(fieldName));
                    }
                }

                RequestStatus status = indexer.addOrUpdateRecord(logRecordBuilder.build());
                statuses.add(status);
            }

            try {
                indexer.waitForAllIndexingToComplete();
            } catch (IndexerInterruptedException e) {
                e.printStackTrace();
                System.exit(1);
            }

            int numErrors = 0;
            for (RequestStatus s : statuses) {
                if (s.getState() == State.ERROR) {
                    handleError(s);
                    numErrors++;
                }
            }

            if (numErrors == 0) {
                System.out.println("success");
            } else {
                System.out.println("Found " + numErrors + " errors");
            }
        } finally {
            indexer.close();
        }

        EntityResolver er = null;
        try {
            er = new EntityResolver(configurationProvider, new SearchOptions.Builder().build());
            ResultSet results =
                er.retrieve(searchFor(entitiesOfType(EXAMPLE_APPLICATION_ENTITY_TYPE)));
            for (Entity e : results) {
                System.out.println(e.getEntityKey().getEntityType()
                    + " -> " + e.getEntityKey().getEntityId());
                System.out.println("\t"
                    + e.getField(new FieldName(DATE_FIELD)).getFieldName()
                    + " --> " + e.getField(new FieldName(DATE_FIELD)).getFieldValues());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (er != null) {
                er.close();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (configurationProvider != null) {
            configurationProvider.close();
        }
        if (embeddedZkServer != null) {
                embeddedZkServer.close();
            }
        }
    }

    private static RecordSchema setupRecordSchema() {
        RecordSchemaBuilder schemaBuilder =
            new RecordSchemaBuilder().addRecordType(EXAMPLE_APPLICATION_ENTITY_TYPE);
        schemaBuilder = schemaBuilder.addDateField
            (DATE_FIELD, new DateParser(dateFormatter))
            .retrievable(true).filterable(true).sortable(true);
        schemaBuilder = schemaBuilder.addTextField(MESSAGE_FIELD);
        schemaBuilder = schemaBuilder.addTextField(TYPE_FIELD);
        schemaBuilder = schemaBuilder.addLongField(FLAG_FIELD)
            .searchable(false).sortable(true).filterable(true);
        return schemaBuilder.build();
    }

    private static Map<String, String> generateKeyValueHash(String recordId) {
        Map<String, String> nameValueHash = new HashMap<String, String>();
        nameValueHash.put(DATE_FIELD, generateRandomDate());
        nameValueHash.put(MESSAGE_FIELD, "sample log message");
        nameValueHash.put(TYPE_FIELD, "info");
        nameValueHash.put(FLAG_FIELD, "1");
        nameValueHash.put(ID_FIELD, recordId);
        return nameValueHash;
    }

    private static String generateRandomDate() {
        calendar.add(Calendar.DAY_OF_YEAR, 1);
        return dateFormatter.format(calendar.getTime());
    }

    public static Configuration generateConfiguration() {
        return new ExampleBigIndexConfigurationGenerator.Builder
            (DATA_EXPLORER_ENDPOINT_URL, DATA_EXPLORER_ENDPOINT_USERNAME,
            DATA_EXPLORER_ENDPOINT_PASSWORD, EXAMPLE_APPLICATION_COLLECTION_NAME,
            EXAMPLE_APPLICATION_ENTITY_TYPE).build().getGeneratedConfiguration();
    }

    private static void handleError(RequestStatus s) {
        try {
            s.getRecordId();
            s.getRecordType();
            s.getSectionId();
            throw s.getIndexerError().getIndexerException();
        } catch (IndexerException e) {
            // Catching exception just for example purposes
        }
    }
}
```

Preparing to Index

The `ExampleApplication` class begins by defining static string variables to define required elements of the example, some of which are similar to those defined for the Hello World example. For example, the *EXAMPLE_APPLICATION_COLLECTION_NAME* and *EXAMPLE_APPLICATION_ENTITY_TYPE* specify the same basic information. But because this example indexes more complex data, it defines a number of additional variables to specify further aspects of the index; these are described below.

```
public static final String EXAMPLE_APPLICATION_COLLECTION_NAME =
     "example-application-test-collection";
public static final String EXAMPLE_APPLICATION_ENTITY_TYPE = "ex_entity_type";
public static final String ID_FIELD = "id";
public static final String FLAG_FIELD = "flag";
public static final String TYPE_FIELD = "type";
public static final String MESSAGE_FIELD = "message";
public static final String DATE_FIELD = "date";
```

The code also instantiates two additional objects. The first is a `Calendar` object used to define random date information for specific records; the instance is initialized with the current date and time by the `getInstance` method of the `Calendar` class.

The second is a SimpleDateFormat object used to specify the date format for a record. The example's use of these methods is also described further below.

```
private static Calendar calendar = Calendar.getInstance();
private static SimpleDateFormat dateFormatter = new SimpleDateFormat("MM/dd/yyyy");
```

The Hello World example defined its record schema inline because the entity type defined for its record type had only a single retrievable text field. The Example Application, on the other hand, relies on a separate method to define its more complex schema. Because the entity type for its record type has multiple fields with a variety of attributes, the code relies on a local method, setupRecordSchema, to create its RecordSchema.

The setupRecordSchema method begins by instantiating a new RecordSchemaBuilder named *recordSchema*, to which it adds the record type defined by the variable *EXAMPLE_APPLICATION_ENTITY_TYPE*, ex_entity_type. Methods of the RecordSchemaBuilder class are then called to augment the schema with the fields used by the example, all of which are identified by the variables defined previously.

First, the addDateField method is called to add a date field based on the definition of the *DATE_FIELD* variable that uses a new DateParser based on the SimpleDateFormat instance created earlier. The parser is specified to enable fine-grained control when moving from the custom DATE_FIELD object to a proper Java object format supported by the API, in this case from a date represented as a string to a proper Java Date object. In addition to being searchable by default, this field is defined as retrievable, meaning that a copy is stored in the index from which it can be retrieved and presented to the user; filterable, meaning that it can be used in equality queries; and sortable, meaning that the results of a search can be sorted based on its value. The addTextField method is then called to add text fields based on the *MESSAGE_FIELD* and *TYPE_FIELD* variables, both of whuch are searchable by default. The addLongField method is then called to add a long field for the *FLAG_FIELD* variable that is not searchable but is sortable and filterable.

Finally, the build method of the RecordSchemaBuilder class is called to return the new instance of the RecordSchema.

```
private static RecordSchema setupRecordSchema() {

     RecordSchemaBuilder schemaBuilder =
          new RecordSchemaBuilder().addRecordType(EXAMPLE_APPLICATION_ENTITY_TYPE);

     schemaBuilder = schemaBuilder
          .addDateField(DATE_FIELD, new DateParser(dateFormatter))
          .retrievable(true).filterable(true).sortable(true);
     schemaBuilder = schemaBuilder.addTextField(MESSAGE_FIELD);
     schemaBuilder = schemaBuilder.addTextField(TYPE_FIELD);
     schemaBuilder = schemaBuilder.addLongField(FLAG_FIELD)
          .searchable(false).sortable(true).filterable(true);

     return schemaBuilder.build();

}
```

As described, the code calls the application's setupRecordSchema method to create its RecordSchema. It then instantiates a RecordBuilderFactory by a call to that class's constructor, to which it passes the *recordSchema* object to indicate the schema to adhere to when creating record builders to produce the actual records stored in the index. An instance of the `DataIndexer` class is then created, specifying the configuration provider and the indexer options as arguments, as in the Hello World example.

```
RecordSchema recordSchema = setupRecordSchema();
RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
IndexerOptions options = new IndexerOptions.Builder().build();
Indexer indexer = new DataIndexer(configurationProvider, options);
```

Indexing

Within a nested `try` block, the example code now proceeds with the actual indexing of the data. The *recordBuilderFactory* object is used to create a `RecordBuilder` named *logRecordBuilder* that creates records for the *EXAMPLE_APPLICATION_ENTITY_TYPE*, `ex_entity_type`. Because this example creates many more records of greater complexity, the code to perform the indexing is noticeably more complex than that of the Hello World example. It also employs a much more robust model for error handling to ensure that data is successfully indexed. To this end, it initializes an `ArrayList` of `RequestStatus` objects to capture the status of each indexing operation; see "Handling Indexing Errors" on page 17 for more information about how this collection is used.

```
RecordBuilder logRecordBuilder =
     recordBuilderFactory.newRecordBuilder(EXAMPLE_APPLICATION_ENTITY_TYPE);

List<RequestStatus> statuses = new ArrayList<RequestStatus>();
```

The code then creates a second `ArrayList`, this time of `Map` objects to represent name-value pairs of data converted to strings. It then calls the local `generateKeyValueHash` method to populate the *records* `Map` with one hundred elements that represent fields and their values according to the example's entity type; this method is described later in this section. The code then iterates over each element of *records*, calling methods of the *logRecordBuilder* object to populate a new index record with the field values from each element. It first uses the `id` method to set the ID of the record based on the element's *ID_FIELD*. Then, for each remaining key in the element, it uses the `addField` method to add the key and its value as the *fieldName* and *fieldValue* (the two parameters of the method) of the new record. The types of the fields were defined when the schema was established by the `setupRecordSchema` method.

After each record has been created by the *logRecordBuilder*, the `addOrUpdateRecord` method of the `DataIndexer` object, *indexer*, is called to add the new record to the collection. The `build` method of the *logRecordBuilder* object creates each record to be added. The *indexer* object returns the status of each operation as a `RequestStatus` object, and each status is added to the list of *statuses* to enable the error handling described in the following section.

```
List<Map<String, String>> records = new ArrayList<Map<String, String>>();

for (int recordId = 0; recordId < 100; recordId++) {
     records.add(generateKeyValueHash(Integer.toString(recordId)));
}

for (Map<String, String> record : records) {
     logRecordBuilder = logRecordBuilder.id(record.get(ID_FIELD));
     for (String fieldName : record.keySet()) {
          if (!fieldName.equals(ID_FIELD)) {
               logRecordBuilder =
                    logRecordBuilder.addField(fieldName, record.get(fieldName));
          }
     }
     RequestStatus status = indexer.addOrUpdateRecord(logRecordBuilder.build());
     statuses.add(status);
}
```

As with the Hello World example, the waitForAllIndexingToComplete method of the DataIndexer object is called to block indefinitely until the indexing operations complete, with a try/catch block employed to capture a possible IndexerInterruptedException.

```
try {
     indexer.waitForAllIndexingToComplete();
} catch (IndexerInterruptedException e) {
     e.printStackTrace();
     System.exit(1);
}
```

The indexing operation concludes with a call to the close method of the DataIndexer class. The call is enclosed within a finally clause to ensure that it is called to free all resources used by the indexing operation.

```
indexer.close();
```

The supporting generateKeyValueHash method mentioned previously creates the elements added to the *records* Map. This function simply adds example values for each field of a record. For the *DATE_FIELD*, it calls the generateRandomDate method to increment the current date represented by the Calendar object by one and returns that value as a string in the format MM/dd/yyyy specified by the SimpleDateFormat instance named *dateFormatter*. The values of the remaining fields are simple and obvious from the code. The name-value pairs for the record are all returned via the *nameValueHash* HashMap to which they are added.

```
private static Map<String, String> generateKeyValueHash(String recordId) {
     Map<String, String> nameValueHash = new HashMap<String, String>();
     nameValueHash.put(DATE_FIELD, generateRandomDate());
     nameValueHash.put(MESSAGE_FIELD, "sample log message");
     nameValueHash.put(TYPE_FIELD, "info");
     nameValueHash.put(FLAG_FIELD, "1");
     nameValueHash.put(ID_FIELD, recordId);
     return nameValueHash;
}

private static String generateRandomDate() {
     calendar.add(Calendar.DAY_OF_YEAR, 1);
     return dateFormatter.format(calendar.getTime());
}
```

Handling Indexing Errors

As described in "Indexing" on page 16, the Example Application takes a more refined approach to error handling than the simple Hello World example employed. With only a single record added to the index, that earlier example could adopt a simpler approach to determine whether the indexing operation succeeded. Because the current example adds an array of records, it relies on the ArrayList of RequstStatus objects represented by *statuses*, to which the result of each addOrUpdateRecord call was added, to determine the success of the cumulative indexing operations.

When the call to waitForAllIndexingToComplete unblocks, the code initializes a *numErrors* counter and checks each element in the *statuses* List. If the getState method of the RequestStatus interface indicates that the current status equals State.ERROR, the code calls the local handleError method and increments the *numErrors* counter by one. If no errors were encountered, the code displays a simple success message; otherwise, it reports the number of errors found.

```
int numErrors = 0;
for (RequestStatus s : statuses) {
     if (s.getState() == State.ERROR) {
          handleError(s);
          numErrors++;
     }
}

if (numErrors == 0) {
     System.out.println("success");
} else {
     System.out.println("Found " + numErrors + " errors");
}
```

The `handleError` method offers an example to illustrate the operations available with the `RequestStatus` object and how to reach an exception if one exists. The `getRecordID`, `getRecordType`, and `getSectionId` methods can be used to obtain the ID, type, and section ID of the record whose addition generated an error from its status. The `getIndexerError` method returns the error associated with a request, if one exists, and the `getIndexerException` method returns the `IndexerException` associated with that error. The following code throws and catches such an exception but does not respond to it at present. It provides a template for how you might do so in your own code:

```
private static void handleError(RequestStatus s) {
     try {
          s.getRecordId();
          s.getRecordType();
          s.getSectionId();
          throw s.getIndexerError().getIndexerException();
     } catch (IndexerException e) {
          // Catching exception just for example purposes
     }
}
```

Another approach to checking the status of an indexing operation is to use the `CallbackDataIndexer` class instead of the `DataIndexer` class to index data. The two classes are functionally equivalent, but the former uses an implementation of the `IndexerCallback` interface, which provides methods that parallel the states represented by the `RequestStatus.State` type, to evaluate the status of different stages of an indexing operation. A later example demonstrates the use of this approach.

Searching

The code to search for and retrieve the indexed data for the Example Application is very similar to the code for the Hello World example. An `EntityResolver` named *er* is initialized and instantiated. The call to the constructor specifies the configuration provider, which is the embedded Zookeeper server for this example (see "Supporting Utilities" on page 19), and an instance of the `SearchOptions` class, which is created via a call to the `build` method of the `SearchOptions.Builder` class.

The `retrieve` method of the `EntityResolver` class is then used to search for entities of type `ex_entity_type`, as specified by the *HELLO_WORLD_ENTITY_TYPE* variable, the sole entity type for this example. The retrieval is accomplished via the static `searchFor` and `entitiesOfType` methods of the `RequestBuilder` class. The `retrieve` method returns an instance of the `SearchResultSet` class named *results*, which implements the `ResultSet` interface.

```
EntityResolver er = null;
...
er = new EntityResolver(configurationProvider, new SearchOptions.Builder().build());
ResultSet results =
     er.retrieve(searchFor(entitiesOfType(EXAMPLE_APPLICATION_ENTITY_TYPE)));
```

The code then iterates over the *results* to display them for the user via calls to methods of the `Entity` class. The first line of code shows the type and ID of the entity based on its entity key; the second shows the name and value of the date field. For the field-related values, the code creates an object of type `FieldName` to pass to the `getField` method of the `Entity` class, after which it uses the `getFieldName` and `getFieldValues` methods to obtain the actual strings to be displayed. Note that the *MESSAGE_FIELD* is not included in the output; since that field was indicated as searchable but not retrievable for this example, the results of the search return nothing for that field.

```
for (Entity e : results) {
     System.out.println(e.getEntityKey().getEntityType()
          + " -> " + e.getEntityKey().getEntityId());
     System.out.println("\t" + e.getField(new FieldName(DATE_FIELD)).getFieldName()
          + " --> " + e.getField(new FieldName(DATE_FIELD)).getFieldValues());
}
```

Finally, the `close` method of the `EntityResolver` object is called to close any services used in the search operation. The code makes the call from a `finally` clause, in which it ensures that *er* is not null.

```
if (er != null) {
     er.close();
}
```

The following shows a portion of the results displayed when the example application is run. The success message is displayed at conclusion of the index operation; the details about the retrieved entity and its date field are produced by the search operation. The full internal names and identifiers of the various elements are shown, along with the string identifiers specified by the code.

```
success
com.ibm.dataexplorer.bigindex.search.model.EntityType@b4240b92[entityType=ex_entity_type]
-> com.ibm.dataexplorer.bigindex.search.model.EntityId@46d14b7
   [entityId=4285fbbedee53e485485f2e90efd736e,bigIndexRecordId=12]
        com.ibm.dataexplorer.bigindex.search.model.FieldName@66964fbe[fieldName=date]
        --> [com.ibm.dataexplorer.bigindex.search.model.FieldValue@33e83921
            [fieldValue=2014-11-27T00:00:00-05:00,boldRanges=[]]]
com.ibm.dataexplorer.bigindex.search.model.EntityType@25e8c807[entityType=ex_entity_type]
-> com.ibm.dataexplorer.bigindex.search.model.EntityId@717175b9
   [entityId=7943cc64d05a595d0467399ca7009c26,bigIndexRecordId=80]
        com.ibm.dataexplorer.bigindex.search.model.FieldName@a9fb383d[fieldName=date]
        --> [com.ibm.dataexplorer.bigindex.search.model.FieldValue@c2c6afa8
            [fieldValue=2015-02-03T00:00:00-05:00,boldRanges=[]]]
com.ibm.dataexplorer.bigindex.search.model.EntityType@390eef5d[entityType=ex_entity_type]
-> com.ibm.dataexplorer.bigindex.search.model.EntityId@d7fbd579
   [entityId=f309c71fe7f8fd1f0cde5eee1ebb198e,bigIndexRecordId=36]
        com.ibm.dataexplorer.bigindex.search.model.FieldName@46a28b5d[fieldName=date]
        --> [com.ibm.dataexplorer.bigindex.search.model.FieldValue@58164272
            [fieldValue=2014-12-21T00:00:00-05:00,boldRanges=[]]]
...
```

Supporting Utilities

This section describes the additional supporting utilities provided with the BigIndex example code. “Watson Explorer Engine Configuration” describes the Watson Explorer Engine configuration used with the example code; “Embedded Zookeeper Server” on page 21 describes the embedded Zookeeper server used with the example code. The information that follows pertains to all examples provided with the BigIndex API.

Watson Explorer Engine Configuration

Each example provided with the BigIndex API takes one of two approaches to specify Watson Explorer Engine configuration data. The first approach is used for simple example applications, which rely on a single collection and a single shard as defined in the file `ExampleBigIndexConfigurationGenerator.java` in the `bigindex/examples/exampleutils` directory. The source files for these simple examples have the following `import` statement to include the `ExampleBigIndexConifgurationGenerator` member of the `bigindex.exampleutils` package:

```
import bigindex.exampleutils.ExampleBigIndexConfigurationGenerator;
```

The code for each simple example defines a *configuration* variable of type `Configuration`, which is an interface that serves as the container for all `EntityModel`, `DataStores`, and `DataSources` classes. Each example includes a `generateConfiguration` method that specifies the data source (Watson Explorer Engine instance), data store (collection), and entity types used with the example; the examples call this method to define the *configuration* variable.

```
Configuration configuration = generateConfiguration();
```

For instance, the `HelloWorld.java` and `ExampleApplication.java` examples include the following `generateConfiguration` method. This method relies on the `build` method of the `Builder` class of the `ExampleBigIndexConifgurationGenerator` class to generate the configuration. It calls the `getGeneratedConfiguration` method of the `ExampleBigIndexConifgurationGenerator` class to return the new configuration.

```
public static Configuration generateConfiguration() {
     return new ExampleBigIndexConfigurationGenerator.Builder(DATA_EXPLORER_ENDPOINT_URL,
          DATA_EXPLORER_ENDPOINT_USERNAME, DATA_EXPLORER_ENDPOINT_PASSWORD,
          HELLO_WORLD_COLLECTION_NAME, HELLO_WORLD_ENTITY_TYPE).build()
          .getGeneratedConfiguration();
}
```

The resulting *configuration* is passed to the `build` method of the `Builder` class of the `ExampleEmbeddedZookeeperServer` to create an instance of the embedded server. To use the simple examples, you must update the following lines of the source code to change the static string *DATA_EXPLORER_ENDPOINT_URL* to identify the URL of your Watson Explorer Engine instance and the static strings *DATA_EXPLORER_ENDPOINT_USERNAME* and *DATA_EXPLORER_ENDPOINT_PASSWORD* to specify your username and password for that instance. For example, depending on how you configured your Watson Explorer Engine, the three placeholders shown in italics in the following code snippet may take the values **localhost:9080/vivisimo**, **data-explorer-admin**, and **TH1nk1710**, respectively.

```
public static final String DATA_EXPLORER_ENDPOINT_URL =
     "http://path-to-data-explorer-engine-instance-1/cgi-bin/velocity.exe"
     + "?v.app=api-soap&wsdl=1&use-types=true";
public static final String DATA_EXPLORER_ENDPOINT_USERNAME = "your-username";
public static final String DATA_EXPLORER_ENDPOINT_PASSWORD = "your-password";
```

The second approach, which is used for more complex example applications, does not use the `ExampleBigIndexConifgurationGenerator` class. Instead, these examples rely on external configuration model files to specify the data source, data store, and entity types used with the examples. In this case, the static string *CONFIGURATION_FILE_LOCATION* identifies the configuration file to be used with the examples, which is passed to the new instance of the embedded Zookeeper server.

```
ExampleEmbeddedZookeeperServer embeddedZkServer =
     new ExampleEmbeddedZookeeperServer.Builder(CONFIGURATION_FILE_LOCATION).build();
```

To use the complex examples, you must update the external configuration model file associated with the example to change each occurrence of the *url*, *username*, and *password* attributes to indicate the URL of your Watson Explorer Engine instance and the username and password for that instance, as described previously. (Note that the value of the *url* attribute must be a single line, with no embedded line break; the value in the following example is broken across two lines for presentational purposes only.)

```
<data-sources>
     <data-explorer-engine-instance username="your-username"
          url="http://path-to-data-explorer-engine-instance-1/cgi-bin/velocity.exe
          ?v.app=api-soap&wsdl=1&use-types=true&"
          password="your-password"
          id="instance-1" />
</data-sources>
```

In some cases, you need to identify multiple instances, usernames, and passwords. More information about the individual external configuration files is provided with the examples with which they are associated.

Embedded Zookeeper Server

Each of the example applications provided with the BigIndex API relies on an embedded Zookeeper server for centralized configuration management. The embedded Zookeeper server runs as part of the example application code rather than as a separate server. It persists for the duration of the application, exiting and deleting the data it manages when the application exits.

Important: The embedded Zookeeper server is a standalone, non-clustered server that is provided only as a convenience for working with the example applications; it is not intended for use with a production deployment. In production, you need to use distributed, persistent instances of the server to manage your configuration data reliably.

The source code for the embedded Zookeeper server is provided in the file `ExampleEmbeddedZookeeperServer.java` in the `bigindex/examples/exampleutils` directory. Each source file has the following `import` statement to include the `ExampleEmbeddedZookeeperServer` member of the `bigindex.exampleutils` package:

```
import bigindex.exampleutils.ExampleEmbeddedZookeeperServer;
```

For instance, the following line of code from the `HelloWorld.java` file instantiates the embedded Zookeeper server with a configuration based on a single collection and a single shard, as used with the simple example applications that rely on the `ExampleBigIndexConifgurationGenerator` class. The *configuration* variable identifies the name of the `Configuration` instance to be used with the example.

```
ExampleEmbeddedZookeeperServer embeddedZkServer =
     new ExampleEmbeddedZookeeperServer.Builder(configuration).build();
```

Alternatively, the following line of code instantiates the embedded server based on an external configuration model file, as used with the more complex example applications. In this case, the static string *CONFIGURATION_FILE_LOCATION* identifies the configuration file to be used with the example.

```
ExampleEmbeddedZookeeperServer embeddedZkServer =
     new ExampleEmbeddedZookeeperServer.Builder(CONFIGURATION_FILE_LOCATION).build();
```

In both cases, the following lines of code first initialize the `ZookeeperBigIndexConfigurationProvider`, which loads a BigIndex configuration from a Zookeeper server, and then start the embedded Zookeeper server (the latter line is enclosed in a `try/catch` clause in the actual code):

```
ZookeeperBigIndexConfigurationProvider configurationProvider = null;
embeddedZkServer.startEmbeddedServer();
```

Subsequent lines of code create the instance of the `ZookeeperBigIndexConfigurationProvider` named *configurationProvider* based on an instance of a `ZookeeperConfiguration`, which encapsulates the URLs to the endpoints for the Zookeeper server and the base namepace for the data used in the example. For simple examples, the call to the `getZookeeperConfiguration` method of the `ExampleEmbeddedZookeeperServer` class instantiates the Zookeeper configuration as *zkConfig*, which is then passed to the constructor for the `ZookeeperBigIndexConfigurationProvider` class.

```
ZookeeperConfiguration zkConfig = embeddedZkServer.getZookeeperConfiguration();
configurationProvider = new ZookeeperBigIndexConfigurationProvider(zkConfig);
```

For complex examples, the instance of `ZookeeperConfiguration` is created anonymously as part of the call to the constructor. The two approaches are functionally equivalent.

```
configurationProvider = new ZookeeperBigIndexConfigurationProvider(
     embeddedZkServer.getZookeeperConfiguration());
```

All example applications include a `finally` clause that closes the configuration provider and the embedded Zookeeper server when the example exits. The following code is from the `HelloWorld.java` example, but all examples include the same or obviously similar lines of code:

```
} finally {
     if (bigIndexConfigurationProvider != null) {
          bigIndexConfigurationProvider.close();
     }
     if (embeddedZkServer != null) {
          embeddedZkServer.close();
     }
}
```

Chapter 4. Basic Data Indexing

This chapter describes basic indexing, also known as data ingestion, with the Watson Explorer BigIndex Application Programming Interface (API). The API provides the capabilities you need to build a client application for indexing and searching data. The BigIndex API enables the programmtic creation, distribution, and management of indices via instances of the Watson Explorer Engine. You can push potentially vary large amounts of data through the API to make the data searchable.

You put into the index only that data you plan to explore in some fashion later, either via the BigIndex API or with an Watson Explorer Application Builder application. Indexing and searching are therefore closely related topics: you can search for data using only the structure and attributes with which you index it. When creating your index, you therefore need to consider what you will search for and how. Such questions help you determine the types of records you will create, the number and types of fields your records will contain, the properties of those fields, how your records will be connected via associations, and more. As this chapter explains, your answers to these questions have direct implications for how you can search your data, a concept that is reinforced by the many example applications provided with the API. As the code and explanations make clear, the indexing and searching components of the examples complement each other, with each index designed to accommodate its corresponding search.

When you index data, you do not physically move the data into the index; the data remains at its current location, and the index contains information about the data. The size of the index, however, can vary greatly depending on the parameters you choose during the indexing operation. You decide exactly what parts of the data to index and how much of the original data is to be contained in the index, effecting your choices either by using the BigIndex API or by relying on the built-in and highly configurable conversion framework of Watson Explorer Engine. Once you have an index, you can use the BigIndex API to keep it current with respect to the source data. You can update an existing index by adding new information to it, by deleting existing information from it, or by modifying existing information by sending updates to reflect changes to the source data.

The BigIndex API supports the indexing of both structued and unstructured data. Structured data consists of fields that contain meta information such as name, price, and other characteristics relevant to your domain. Unstructured data includes sections and paragraphs of information, tweets, logs, and so on from which structured data can be extracted. For example, you may want to index all mentions of a specific person or product in a collection of tweets. These examples, however, are all types of textual data provided in a human-readable format.

You can also index and search binary-encoded data by using converters to perform entity extraction, transforming the data into field-value pairs for use in the index. An indexed record can contain binary-encoded data in such non-textual formats as Adobe PDF, Microsoft Word, JPEG, and more. Each binary format is assigned a content type that allows the Watson Explorer Engine to select the converter pipeline to apply to transform the data into fields and values. Content types can also be used with regular text fields to enable application-specific entity extraction based on custom converters. Most default base configurations have pedefined converter pipelines for well-known content types, and you can also develop and apply custom converters for a data store configuration. For example, you can index a PDF file along with text fields so that the resulting record is searchable using terms in the PDF document not otherwise found in the text fields. This example also touches on the concept of record sections, which let you index metadata about a document independently of the document itself, as described in “Record Sections” on page 36.

The following sections describe the classes, interfaces, and methods of the BigIndex API associated with basic indexing. The text includes example code drawn from the files `SearchExamples.java`, `RecordSectionsExample.java`, and `CallbackDataIndexerExample.java`, all of which are example applications delivered with the API. The example code in this chapter prepares indices for use by the examples shown in the following chapter, Chapter 5, “Basic Data Searching,” on page 51. The examples in this chapter use only a single collection and shard, and they are driven programmatically via the API; Chapter 6, “Application Configuration Management,” on page 53 describes application configuration that relies on external configuration model files to use multiple collections and shards.

The Entity Model

Entities are the core data elements of a BigIndex application. They define and connect the data represented in your index. The entity model is comparable to a database schema. A database consists of tables, attributes, keys that link the tables, and indexes into the tables. An entity model specifies similar concepts for the index to be built with a BigIndex application.

For example, consider an application to index basic information for the automobile insurance domain. Such an application might define a model with entities for customers, policies, and vehicles. The customer entity type would contain fields with information about a customer, such as name, address, contact information, driver's license, and so on. The policy entity type would contain information about an insurance policy, such as cost, coverage, and liability. And the vehicle entity type would contain information about vehicles, such as identification number, make, model, and license plate number. Each of these entity types would include fields of various types to capture information about specific entities.

These entity types would be linked in the entity model via associations among their fields. In the insurance example, for instance, the customer type would have a field that identifies the policy associated with a customer. The policy type, in turn, would have a field that identifies each of the vehicles covered by that policy. When the index is searched, these associations can be leveraged to extract related data for customers, policies, and vehicles for more meaningful results.

In a BigIndex application, the entity model is defined by a record schema. The basic unit of indexing is a record, an indexable unit that conceptually serves the same role as a document in Watson Explorer Engine. With the BigIndex API, you define a record schema that captures the types, fields, and attributes of your entity model. You can also use record sections to further compartmentalize your indexed data based, for example, on its likelihood to change over time. When you push data into your index, the API creates records and record sections that conform to the entity model defined by a record schema. You can specify a record schema programmatically in Java code or by using configuration model files. The following sections describe the classes, interfaces, and methods of the BigIndex API used to work with entities, fields, and records programmatically; see Chapter 6, “Application Configuration Management,” on page 53 for information about configuration-driven indexing.

You can also build your own configuration provider, in which case you would work directly with the `EntityDefinition` and `EntityModel` classes. The former class defines an entity and its fields and properties; the latter is a container for the configuration of entities. These classes are described in a later chapter of the document.

Entities

Entities are represented in the BigIndex API by the `Entity` class. The `Entity` class is a core data class that serves as a data object for the API. The `Entity` class, like all data objects of the BigIndex API, conforms to the Builder design pattern (see "General Information about the API" on page 65), so it has a nested `Builder` class that you can use to create entities, something you may want to do for testing purposes. But the example applications provided with the API do not use the `Entity.Builder` class. In the examples, entities are instead created internally and returned by search operations, which is how they are usually created in production applications.

Associated with each entity is a type, an ID unique along all entities of that type, and a key, which is a combination of the entity's type and ID and is guaranteed to be unique among all entities. Finally, each entity has a collection of fields, each of which can have any number of values. An entity's fields capture the name-value data pairs asociated with the entity. "Fields" on page 27 describes the `Field` class.

Table 1 describes the methods of the `Entity` class. The following sections then describe entity types, IDs, keys, and creation of entities via the `Entity.Builder` class.

Table 1. Methods of the `Entity` Class

Method	Description
`getEntityKey`	Returns the `EntityKey` that uniquely identifies the entity.
`getEntityType`	Returns the `EntityType` for the entity.
`getEntityId`	Returns the `EntityId` for the entity.
`getFields`	Returns the `Collection` of `Field` objects defined for the entity, which may be empty.
`getField`	Returns the `Field` object associated with a specified field name, which may be specified by a `FieldName` object or by a string.
`getEntityMetadata`	Returns an `EntityMetadata` object that contains metadata about the entity. The metadata describes how the entity was matched in a search and its order in the search results.

Entity Types

Each entity has a single type, which is defined by an instance of the `EntityType` class. The type is specified by the user when the record schema for the entity is created. The examples in Chapter 3, "Hello, BigIndex," on page 7 define a single entity type. The examples described later in this chapter define multiple entity types with different fields and properties to enable different kinds of searches.

Table 2 on page 26 describes the most significant method of the `EntityType` class. (As with all classes of the API, this class includes additional methods, as described in "Common Methods of Classes" on page 65.)

Table 2. Methods of the `EntityType` Class

Method	Description
`getType`	Returns a string representation of the entity type.

Entity IDs

Each entity has an ID, represented by an instance of the `EntityId` class, which provides a unique identifier for the entity within its type. The ID is assigned to the entity automatically when it is created as the result of a search operation. Entities of different types may have the same ID, since they reside in different namespaces.

Table 3 describes the most significant method of the `EntityId` class.

Table 3. Methods of the `EntityId` Class

Method	Description
`getId`	Returns a string representation of the entity ID.
`getBigIndexRecordId`	If the `Entity` was indexed through the BigIndex API, returns a string representation of the entity ID used for indexing; otherwise, returns null.

Entity Keys

Each entity also has a unique key, represented by an instance of the `EntityKey` class, which uniquely identifies the entity among all entities. The key consists of a combination of the entity's type and ID, making it unique across all entities regardless of type. Like its `EntityId`, an entity's `EntityKey` is assigned automatically when it is created by a search operation. The examples in Chapter 3, "Hello, BigIndex," on page 7 used methods of the `EntityKey` class to obtain information about the entities returned from a search.

Table 4 describes the methods of the `EntityKey` class.

Table 4. Methods of the `EntityKey` Class

Method	Description
`getEntityId`	Returns the `EntityId` associated with the entity key.
`getEntityType`	Returns the `EntityType` associated with the entity key.

Building Entities

Instances of the `Entity` class are typically returned by search requests, which is the approach shown in the examples provided with the BigIndex API. Nonetheless, the class does include a nested `Builder` class that allows you to create entities directly, which you might want to do for testing purposes. The `Entity.Builder` class has two constructors. One requires that you pass both an `EntityType` and an `EntityId`; the second allows you to specify the same values via a single `EntityKey` argument. Additional methods of the class let you specify fields and metadata for the new entity and to build the instance of the entity.

Table 5 describes the methods of the `Entity.Builder` class.

Table 5. Methods of the `Entity.Builder` Class

Method	Description
`addField`	Returns an `Entity.Builder` that creates an entity with the specified `Field`.

Table 5. Methods of the `Entity.Builder` Class (continued)

Method	Description
`addFields`	Returns an `Entity.Builder` that creates an entity with the specified `Fields`, which can be passed to the method as individual arguments or in the form of a collection of `Field` objects.
`entityMetadata`	Returns an `Entity.Builder` that creates an entity with the specified `EntityMetadata`.
`build`	Returns an instance of the `Entity` class that has the entity type and ID specified by the constructor for the `Builder` class and any fields and metadata specified by the previous methods of the class.

Fields

A field refers to a single data element of an entity. Each field is uniquely identified within an entity by its name. For example, an entity of type customer might have fields for the customer's name, address, email address, phone number, and so on. Each field would have one or more values providing the associated information about the entity. For example, the name field for a customer entity would have a single value, the customer's name, while the email address and phone number fields could have multiple values for a customer who has more than one email addess or phone number. Fields can be explicitly referenced when retrieving entities, such as in search requests or to sort the entities returned from a search by field value. Fields can also enable associations between different entities for a richer data hierarchy.

Each field is given a type when it is added to an entity definition to indicate the sort of data its values provide. Each field can also be assigned properties to indicate the kinds of search operations supported on its values. The following subsections describe field types and properties in more detail.

Different versions of the `Field` class exist in multiple packages. All are similar and manipulate conceptually equivalent objects. But the three classes are each derived from the Java `Object` class and are used in different contexts. The version discussed in this section is included in the `com.ibm.dataexplorer.bigindex.ingestion.record` package. Additional versions of the class exist in the `com.ibm.dataexplorer.bigindex.search.model` package, which is the version used to access search results (see Chapter 5, "Basic Data Searching," on page 51), and in the `com.ibm.dataexplorer.bigindex.configuration` package, which is used when developing configuration providers.

The version of the `Field` class discussed in this chapter has a nested `Builder` class that you can use to create instances of the `Field` class. It also has a number of methods that you can call on instances of the class. The example applications provided with the API do not use the nested `Builder` class to create fields; the examples instead add fields to the schema for an entity type via the `RecordSchemaBuilder` class and create records that contain the fields via an object of type `RecordBuilder`, as described later in this chapter. The examples also do not demonstrate the methods of the `Field` class, which may prove useful for testing purposes.

Note that this chapter discusses only programmatic definition of entities and fields by a record schema builder. Like the entities with which they are associated, fields can also be defined by an external configuration model file. Discussions of configuration-driven indexing and additional topics such as bolding, tokenization, and access control are reserved for later chapters.

Table 6 describes the methods of the `Field` class. The following sections describe field types and properties, and creation of fields via the `Field.Builder` class.

Table 6. Methods of the `Field` Class

Method	Description
`getName`	Returns a string that represents the field's name.
`getValue`	Returns a string that represents the field's value
`getType`	Returns the `FieldType` for the field.
`isSearchable`	Returns a boolean value indicating whether the field is searchable.
`isRetrievable`	Returns a boolean value indicating whether the field is retrievable.
`isSortable`	Returns a boolean value indicating whether the field is sortable.
`isFacetable`	Returns a boolean value indicating whether the field is facetable.
`isFilterable`	Returns a boolean value indicating whether the field is filterable.
`getWeight`	Returns a double value indicating the field's weight.
`getAclEntries`	Returns a collection of `AclEntry` objects for the field if any exist; otherwise, returns an empty collection; see *later chapter*.
`getFieldOptions`	Returns the `FieldOptions` for the field; see *later chapter*.
`getTokenizationOptions`	Returns the `TokenizationOptions` for the field; see *later chapter*.

Field Types

A field is assigned a type when it is added to an entity definition. Its type indicates the values that can be assigned to the field. Acceptable field types are defined by the `FieldType` enumerated type. Programmatically, a field's type is specified when it is added to an entity by a record schema builder. A field's type has important implications for its properties, since the type can enable more advanced features such as proper numeric sorting, range facets and queries, and so on.

Table 7 lists the supported field types, describing the types and the methods of the `RecordSchemaBuilder` class that are used to add them to an enity.

Table 7. `FieldType` Constants

Type	Description
DATE	A field whose value is a Java `Date` object. A field of this type is created with the `addDateField` method.
DOUBLE	A field whose value is of Java primitive data type `double`. A field of this type is created with the `addDoubleField` method.
LONG	A field whose value is of Java primitive data type `long`. A field of this type is created with the `addLongField` method.
TEXT	A field whose value is a Java `String` object. A field of this type is created with the `addTextField` method. (Additional classes have a generic `addField` method; if not otherwise specified, such methods create fields of type TEXT.)

Two versions exist for each of the methods cited in the table. The first takes a single string argument that specifies the name of the field to be added. The second takes an additional argument that names an object that implements the `Parser` interface. Objects of this type allow greater flexibility when specifying field values by supporting the use of custom domain objects. When a value is specified for the field, the parser automatically converts it into the appropriate type for the field. The API provides `DateParser`, `DoubleParser`, and `LongParser` classes to convert strings to the respective types.

The `FieldType` class has two static methods that you can use to obtain additional information about this enumerated type. The `valueOf` method returns the enum constant of a field type passed as a string, and the `valuesOf` method returns an array that contains the constants for all of the field types in the order in which they are declared.

Field Properties

Fields have properties that specify the types of search operations they support. A field's properties dictate how it can be used and the type of information it can make available in the results of a search. Table 8 describes the different properties that can be assigned to fields. The table also indicates the methods of the `RecordSchemaBuilder` class that are used to specify the property when defining a field.

Table 8. Field Properties

Property	Description
Searchable	Defines whether the field can be searched by the end user; the default is true. The property is set by the `searchable` method.
Retrievable	Defines whether Watson Explorer Engine stores a copy of the indexed content; the default is false. The property is set by the `retrievable` method.
Sortable	Defines whether the field can be used to sort the results of a search; the default is false. The property is set by the `sortable` method.
Facetable	Defines whether the field can be used to generate facets; the default is false. The property is set by the `facetable` method.
Filterable	Defines whether the field can be used in equality and range queries; the default is false. The property is set by the `filterable` method.
Weight	Defines the relevance of the field when its contents are matched by a search query. The larger a field's weight, the greater its impact on relevance. The default is a weight of 1.0. The property is set by the `weight` method.

Some of the properties are useful only in specific contexts. For example, you might set the searchable property to false to avoid spurious results when searching on the field. In this case, you may still want to make the contents of the indexed data retrievable for presentation to the user. Many of the properties also have implications for other properties. For example, making a field sortable, facetable, or filterable automatically makes it retrievable, regardless of whether the retrievable property is set to false, since these capabilities depend on access to a copy of the indexed data.

In addition, the properties also have implications for the backing Watson Explorer Engine. Because making a field retrievable causes a copy of the indexed content to be stored on the Watson Explorer Engine, this can greatly increase the amount of disk space required to store the index. Also, making a field sortable, facetable, or filterable increases the memory requirements of the Watson Explorer Engine.

Building Fields

The `Field.Builder` class lets you instantiate `Field` objects directly, rather than via the `RecordSchemaBuilder` class as shown in the example applications. For example, you might want to create fields directly for testing purposes that are beyond the scope of the examples. The examples demonstrate adding fields to the scehma for an entity type with the `RecordSchemaBuilder` class and specifying their values when you create records by using objects of type `RecordBuilder`.

The constructor for the `Field.Builder` class takes three arguments: a string indicating the name of the field, a string indicating the value of the field, and the `FieldType` of the field. Table 9 describes the methods of the `Field.Builder` class, which are used to specify the properties of the new field and to build the instance of the field.

Table 9. Methods of the `Field.Builder` Class

Method	**Description**
`searchable`	Returns a `Field.Builder` that creates a field whose searchable property is specified by the single boolean argument of the method.
`retrievable`	Returns a `Field.Builder` that creates a field whose retrievable property is specified by the single boolean argument of the method.
`sortable`	Returns a `Field.Builder` that creates a field whose sortable property is specified by the single boolean argument of the method.
`facetable`	Returns a `Field.Builder` that creates a field whose facetable property is specified by the single boolean arguments of the method.
`filterable`	Returns a `Field.Builder` that creates a field whose filterable property is specified by the single boolean argument of the method.
`weight`	Returns a `Field.Builder` that creates a field that has the indicated weight, which is specified by a single argument of type double.
`addAclEntry`	Returns a `Field.Builder` that creates a field that that has the specified `AclEntry`.
`fieldOptions`	Returns a `Field.Builder` that creates a field that that has the specified `FieldOptions`.
`tokenizationOptions`	Returns a `Field.Builder` that creates a field that that has the specified `TokenizationOptions.`
`build`	Returns an instance of the `Field` class that has the name, value, and type specified by the constructor for the `Builder` class and any characteristics specified by the previous methods of the class.

Field options and tokenization are described in a later chapter of this document.

Records

In the BigIndex API, a record is the basic unit of indexing. The `Record` class is a core data class that represents an indexable unit of information. Records conform to a schema for an entity type that includes that type's fields and their properties. Each record also includes an identifier that uniquely identifies the record. It is actual instances of the Record class, populated with the data to be indexed, that are pushed into an index for later retrieval by search operations. Records can also have record sections, which further decompose your index into more granular units, as described in "Record Sections" on page 36. See "Indexing Data" on page 40 for information about indexing records and record sections.

The example applications provided with the BigIndex API rely on pairs of classes and interfaces to generate new records for an index. The RecordSchemaBuilder class creates a new object of type RecordSchema. RecordSchema is an interface that defines the fields to be associated with an entity type (or record type) along with the properties of those fields. The RecordBuilderFactory class represents a factory object for a specified record schema; it generates objects of type RecordBuilder, which is an interface that produces actual records with data for the fields of the entity (record) type defined by the record schema associated with the factory. Together, these classes and interfaces encapsulate the definition of the entity model and the generation of records for an index.

The Record class includes a nested Builder class that, like the corresponding classes for entities and fields, you may want to use for testing purposes. The example applications forego demonstration of the Record.Builder class and rely instead on the approach described in the preceding paragraph. The following sections describe the classes and interfaces used in the examples.

Table 10 describes the methods of the Record class. A later section describes the Record.Builder class.

Table 10. Methods of the Record Class

Method	Description
getType	Returns a string representation of the entity type of the record.
getContentType	Returns a string representation of the content type of the record as indexed by Watson Explorer Engine; the default content type for all records is application/vxml, as defined by the constant Record.DEFAULT_CONTENT_TYPE.
getId	Returns a string representation of the record's ID.
getFields	Returns a collection of all Fields defined for the record, if any exist; otherwise, returns an empty collection.
getBinaryEncodedData	Returns a collection of all BinaryEncodedData indexed as part of the record, if any exists; otherwise, returns an empty collection. See *later chapter*.

Record Schemas and Builders

A record schema captures the definition of a type of record and its allowable fields. Objects of this type are represented by the RecordSchema interface. These objects are created by instances of the RecordSchemaBuilder class. This class includes all of the methods necessary to specify the record type and its fields.

Table 11 on page 32 describes the many methods of the RecordSchemaBuilder class. The constructor for the class takes no arguments. The methods of the class are used to specify fields of the schema and their properties, and to build the record schema.

Table 11. Methods of the `RecordSchemaBuilder` Class

Method	**Description**
`addRecodType`	Returns a `RecordSchemaBuilder` that adds a new record type; the record type is specified as a string that identifies an entity type.
`addDateField`	Returns a `RecordSchemaBuilder` that adds a new date `Field` to the record type; the field is added to the previous record type added via the `addRecordType` method. The overloaded method accepts just a string identifying the field name or both a string for the field name and an additional `Parser` to convert values to the `DATE` type.
`addDoubleField`	Returns a `RecordSchemaBuilder` that adds a new double `Field` to the record type; the field is added to the previous record type added via the `addRecordType` method. The overloaded method accepts just a string identifying the field name or both a string for the field name and an additional `Parser` to convert values to the `DOUBLE` type.
`addLongField`	Returns a `RecordSchemaBuilder` that adds a new long `Field` to the record type; the field is added to the previous record type added via the `addRecordType` method. The overloaded method accepts just a string identifying the field name or both a string for the field name and an additional `Parser` to convert values to the `LONG` type.
`addTextField`	Returns a `RecordSchemaBuilder` that adds a new text `Field` to the record type; the field is added to the previous record type added via the `addRecordType` method. The overloaded method accepts just a string identifying the field name or both a string for the field name and an additional `Parser` to convert values to the `TEXT` type.
`searchable`	Returns a `RecordSchemaBuilder` that sets the previously added `Field` as searchable; the method takes a single boolean argument.
`retrievable`	Returns a `RecordSchemaBuilder` that sets the previously added `Field` as retrievable; the method takes a single boolean argument.
`sortable`	Returns a `RecordSchemaBuilder` that sets the previously added `Field` as sortable; the method takes a single boolean argument.
`facetable`	Returns a `RecordSchemaBuilder` that sets the previously added `Field` as facetable; the method takes a single boolean argument.
`filterable`	Returns a `RecordSchemaBuilder` that sets the previously added `Field` as filterable; the method takes a single boolean argument.
`weight`	Returns a `RecordSchemaBuilder` that sets the specified weight of the previously added `Field`; the method takes a single argument of type double.
`fieldOptionsDefinition`	Returns a `RecordSchemaBuilder` that sets the specified `FieldOptionsDefinition` of the previously added `Field`.
`tokenizationDefinition`	Returns a `RecordSchemaBuilder` that sets the specified `tokenizationDefinition` of the previously added `Field`.
`build`	Returns an instance of the `RecordSchema` type with the record type, fields, and properties specified by the previous methods of the class.

You must specify a record type before adding fields and properties to it; you may set the type when you instantiate the RecordSchemaBuilder. You may also set multiple record types for a single record schema. Once a record type is set, you can add fields and their properties when you initially call the constructor or in subsequent calls from the new instance of the class; all subsequent calls pertain to the previously added record type. See “Field Types” on page 28 for more information about associating a Parser with a field; see “Field Properties” on page 29 for more information about field properties. Field options and tokenization are described later in this document.

The RecordSchema object returned by a call to the build method of the RecordSchemaBuilder class is passed to an object of type RecordBuilderFactory to generate record builders for that record type. Table 12 describes the methods available with an object of type RecordSchema. These methods provide information about the record types defined for a record schema.

Table 12. Methods of the RecordSchema Interface

Method	Description
getRecordDefinition	Returns the RecordDefinition for the specified record type associated with this RecordSchema; the record type is specified as a string.
getRecordDefinitions	Returns a collection of all RecordDefinitions associated with this RecordSchema; the method has no parameters.

“Record and Field Definitions” describes interfaces that you can use to obtain more detailed information about records and fields.

Record and Field Definitions

Methods of the RecordSchema interface return objects of type RecordDefinition, some of which return objects of type FieldDefinition. These interfaces let you obtain more detailed information about records and their fields.

The RecordDefinition interface serves as the type for objects returned by methods of the RecordSchema interface. You can use methods of the interface to learn additional information about the schema definition for a type of record and its allowable fields. Table 13 describes the methods of the RecordDefinition interface.

Table 13. Methods of the RecordDefinition Interface

Method	Description
getRecordType	Returns a string representing the type of the Record for this record definition.
getFieldDefinition	Returns the FieldDefinition for a specified field name for this record definition; the field name is specified as a string.
getFieldDefinitions	Returns a collection of all FieldDefinitions for this record definition; the method has no parameters.

The FieldDefinition interface serves as the type for objects returned by the field-related methods of the RecordDefinition interface. You can use methods of the interface to learn more about about the fields associated with a record definition. Table 14 on page 34 describes the methods of the FieldDefinition interface.

Table 14. Methods of the `FieldDefinition` Interface

Method	Description
`getName`	Returns a string that represents the name of the field for this definition.
`getType`	Returns the `FieldType` of the field for this definition.
`isSearchable`	Returns a boolean value indicating whether the field for this definition is searchable.
`isRetrievable`	Returns a boolean value indicating whether the field for this definition is retrievable.
`isSortable`	Returns a boolean value indicating whether the field for this definition is sortable.
`isFacetable`	Returns a boolean value indicating whether the field for this definition is facetable.
`isFilterable`	Returns a boolean value indicating whether the field for this definition is filterable.
`getWeight`	Returns a double value indicating the weight of the field for this definition.
`getParser`	Returns the `Parser` to be used with the field for this definition, if one exists.
`getTokenizationDefinitions`	Returns the `TokenizationDefinition` of the field for this definition.
`getFieldOptionsDefinition`	Returns the `FieldOptionsDefinition` of the field for this definition.

See “Field Types” on page 28 for more information about field types and parsers; see “Field Properties” on page 29 for more information about field properties. Field options and tokenization are described later in this document.

Record Builders and Factories

A record builder creates a record to be added to an index. It specifies the identifier for the record and the values of the record's fields. It can also specify additional aspects of the record, such as the content type of the record as a whole, Access Control List (ACL) entries for fields of the record, and binary-encoded data associated with the record. (Many of these concepts are described in a later chapter of this document, after you have been introduced to basic indexing, searching, and configuration management.)

Objects of type `RecordBuilder` are created by record builder factories. You first create a new record builder factory for a specified schema and then use that new factory to create a new record builder. The `RecordBuilderFactory` class has two constructors. The first creates a new record builder factory based on a specified record schema; this is the approach described in this chapter. The second creates a new factory based on a record schema defined by a configuration provider; this approach is described in Chapter 6, “Application Configuration Management,” on page 53.

Table 15 on page 35 describes the methods of the `RecordBuilderFactory` class, both of which create new record builders.

Table 15. Methods of the `RecordBuilderFactory` Class

Method	Description
`newRecordBuilder`	Returns a new `RecordBuilder` that builds records of the specified type; the specified record type must exist in the record schema associated with the `RecordBuilderFactory`.
`newRecordSectionBuilder`	Returns a new `RecordSectionBuilder` that builds record sections of the specified type; the specified record type must exist in the record schema associated with the `RecordBuilderFactory`. (Record sections are described in "Record Sections" on page 36.)

The `newRecordBuilder` method returns an object of type `RecordBuilder` that implements the methods of that interface. These methods allow you to specify fields and other information for a new record and to build the actual record. Table 16 describes the methods of the `RecordBuilder` interface.

Table 16. Methods of the `RecordBuilder` Interface

Method	Description
`contentType`	Sets the content type of the new `Record` to the specified string. The content type applies to all fields of the `Record` and is used optionally to leverage converters in Watson Explorer Engine. Calling this method multiple times overwrites the previously set content type for the record. The default content type for all records is `application/vxml`, as defined by the constant `Record.DEFAULT_CONTENT_TYPE`.
`id`	Sets the ID of the new `Record` to the specified string. Calling this method multiple times overwrites the previously set ID for the record.
`addField`	Adds a `Field` and its value to the new `Record`. Five versions of the overloaded method exist; see the text that follows the table for more information.
`withFieldAclEntry`	Adds the specified `AclEntry` to the previously added `Field` of the new `Record`. Calling this method multiple times adds multiple ACL entries to the `Field`.
`addBinaryEncodedData`	Adds the specified `BinaryEncodedData` to the new `Record`. Calling this method multiple times adds multiple instancs of binary-encoded data to the record.
`build`	Returns an instance of the `Record` class with the content type, ID, fields and values, and any additional elements specified by the previous methods of the interface.

Five versions of the overloaded `addField` method exist for the `RecordBuilder` interface. The first argument of each method is a string that specifies the name of the `Field` to be added to the record. The second argument specifies the value for the indicated field and depends on the field's type: a Java `Date` object, a double value, a long value, a string, or a generic object. With the first four versions of the method, if no parsing rules are defined for the specified field, the method adds a field with the specified value; if parsing rules are defined for the field, the method adds a field whose value is parsed according to those rules. In the fifth version of the method, the object specified as the value is parsed according to the rules defined for the field name in the record schema, which must have a valid `Parser`.

For all five methods, the named `Field` must be defined for the record schema associated with the builder and must have the appropriate type for its value.

Calling the `addField` method multiple times for the same field name adds distinct instances of the `Field` to the record. Each resulting instance of the field contains a separate value for the field, which is how multi-value fields are represented in an index.

Building Records

The `Record.Builder` class lets you create instances of the `Record` object directly, rather than via objects of type `RecordBuilder` and their associated classes, as shown in the example applications. You may want to create records directly, for instance, for testing purposes, which is beyond the scope of the examples.

The constructor for the `Record.Builder` class takes two string arguments: the type of record (entity) to be created and an identifier for the new record. Table 17 describes the methods of the `Record.Builder` class, which are used to specify additional information about the new record and to build the instance of the record.

Table 17. Methods of the `Record.Builder` Class

Method	Description
`contentType`	Returns a `Record.Builder` that creates a record with the specified content type in Watson Explorer Engine; the default content type for all records is `application/vxml`, as defined by the constant `Record.DEFAULT_CONTENT_TYPE`.
`addField`	Returns a `Record.Builder` that creates a record with the specified `Field`.
`addFields`	Returns a `Record.Builder` that creates a record with the specified collection of `Fields`.
`addBinaryEncodedData`	Returns a `Record.Builder` that creates a record with the specified `BinaryEncodedData`; the overloaded method accepts a single instance or a collection of `BinaryEncodedData`.
`build`	Returns an instance of the `Record` class that has the type and identifier specified by the constructor for the `Builder` class and any characteristics specified by the previous methods of the class.

Record Sections

Record sections are a powerful feature of the BigIndex API that let you update only parts, or sections, of the information indexed for a document without needing to re-index the entire document. For example, a user might add a tag, comment, or rating to an existing document. By using a record section for such tags, you could update your index to include this additional metadata without having to index the whole document anew.

Many documents have metadata of this sort, such as a PDF document stored in a distribution management system such as SharePoint, or are purely metadata driven, such as a database table, for example. Within any given document, you can typically group the pieces that make up the document into two general categories: static information that remains constant once created, and dynamic information that can change or evolve over time. You can leverage this distinction to organize the data in your index. The BigIndex API lets you reflect the distinction by allowing you to add all of the static information when you first create a record and then put all of the dynamic information into a record section that you can add and update independently over time and after the fact.

Moreover, you can further decompose such dynamic data for a more fine-grained approach to indexing. For example, perhaps your dynamic data includes logical subsets of information that are always updated together; in this case, it might be more efficient to group these subsets together into a single section. Or perhaps a specific field, such as a tag, is often updated by itself, in isolation from other metadata; in this case, you could choose to represent that tag in its own record section.

You might conclude that all data can be broken into its own section. But such divisions are less efficient in terms of storage and retrieval, so you should choose your record sections wisely. Also consider that when updating a section, all prior data associated with that section is replaced with the new data you provide; you cannot update only certain fields of a section. If a single section has five fields, for instance, but only one of them changes, you still need to provide information for all five fields when you update the section.

Record sections are represented in the BigIndex API by the `RecordSection` class. Each record section is associated with a specific record ID and record type, but each record section also has its own section ID. From a search perspective, record sections are completely transparent. To the end-user, records and their sections always appear to have been indexed together at the same time, so search syntax and behavior are identical for records with and without sections.

Working with recond sections is very similar to working with records. As with building records, the example applications delivered with the BigIndex API use pairs of classes and interfaces to create new record sections for an index. As described previously, the `RecordSchemaBuilder` class creates a new object of type `RecordSchema`, which is an interface that defines the fields to be associated with an entity type and the properties of those fields. The `RecordBuilderFactory` class represents a factory object for a specified record schema; in addition to generating objects of type `RecordBuilder`, this factory class also produces objects of type `RecordSectionBuilder`, which is an interface that creates the actual record sections that contain data for the fields of the record schema associated with the factory.

The model for creating new record sections parallels exactly the model followed for creating new records. See “Record Schemas and Builders” on page 31 for information about record schemas; see “Record Builders and Factories” on page 34 for information about the `RecordBuilderFactory` class; and see the following section for information about the `RecordSectionBuilder` interface. Note that the `RecordSection` class includes a nested builder class that you may use for testing purposes; this nested `Builder` class is described in “Building Records” on page 36.

Table 18 describes the methods of the `RecordSection` class, which you can use to obtain information about a record section.

Table 18. Methods of the `RecordSection` Class

Method	**Description**
`getContentType`	Returns a string representation of the content type of the record section as indexed by Watson Explorer Engine; the default content type for all record sections is `application/vxml`, as defined by the constant `RecordSection.DEFAULT_CONTENT_TYPE`.
`getSectionId`	Returns a string representation of the record section's ID.
`getRecordId`	Returns a string representation of the ID of the record with which the record section is associated.

Table 18. Methods of the `RecordSection` Class (continued)

Method	Description
`getRecordType`	Returns a string representation of the ID of the entity type of the record with which the record section is associated.
`getFields`	Returns a collection of all `Fields` defined for the record section, if any exist; otherwise, returns an empty collection.
`getBinaryEncodedData`	Returns a collection of all `BinaryEncodedData` indexed as part of the record section, if any exists; otherwise, returns an empty collection. See *later chapter*.

Record Section Builders

A record section builder creates a record section to be added to an index. It specifies the identifier for the record section and for the record with which it is associated. It also specifies the type of the record with which the record section is associated and the values of the record section's fields. It can also specify additional aspects of the record section, including the content type of the record section as a whole, ACL entries for fields of the record section, and binary-encoded data for the record section. (As mentioned earlier, the discussions of many of these concepts are reserved for a later chapter.)

Like `RecordBuilders`, objects of type `RecordSectionBuilder` are created by record builder factories. You first create a new record builder factory for a specified schema and then use the `newRecordSectionBuilder` method of the new factory to create a new record section builder; see "Record Builders and Factories" on page 34 for more information about the `RecordBuilderFactory` class.

The `newRecordSectionBuilder` method returns an object of type `RecordSectionBuilder` that implements the methods of that interface. These methods allow you to specify fields and other aspects of the new record section and to build the actual section. Table 19 describes the methods of the `RecordSectionBuilder` interface.

Table 19. Methods of the `RecordSectionBuilder` Interface

Method	Description
`contentType`	Sets the content type of the new `RecordSection` to the specified string. The content type applies to all fields of the `RecordSection` and is used optionally to leverage converters in Watson Explorer Engine. Calling this method multiple times overwrites the previously set content type for the record. The default content type for all record sections is `application/vxml`, as defined by the constant `RecordSection.DEFAULT_CONTENT_TYPE`.
`recordId`	Sets the ID of the record with which the new `RecordSection` is to be associated to the specified string. Calling this method multiple times overwrites the previously set record ID for the record section.
`sectionId`	Sets the record section ID of the new `RecordSection` to the specified string. Calling this method multiple times overwrites the previously set ID for the record section.
`addField`	Adds a `Field` and its value to the new `RecordSection`. Five versions of the overloaded method exist; see the text that follows the table for more information.

Table 19. Methods of the `RecordSectionBuilder` Interface (continued)

Method	Description
`withFieldAclEntry`	Adds the specified `AclEntry` to the previously added `Field` of the new `RecordSection`. Calling this method multiple times adds multiple ACL entries to the `Field`.
`addBinaryEncodedData`	Adds the specified `BinaryEncodedData` to the new `RecordSection`. Calling this method multiple times adds multiple instances of binary-encoded data to the record section.
`build`	Returns an instance of the `RecordSection` class with the content type, record ID, section ID, fields and values, and any additional elements specified by the previous methods of the interface.

Five versions of the overloaded `addField` method exist for the `RecordSecionBuilder` interface. The first argument of each method is a string that specifies the name of the `Field` to be added to the record section, and the second argument specifies the value for the indicated field. The value depends on the field's type: a Java `Date` object, a double value, a long value, a string, or a generic object. With the first four versions of the method, if no parsing rules are defined for the specified field, the method adds a field with the specified value; if parsing rules are defined for the field, the method adds a field whose value is parsed according to those rules. In the fifth version of the method, the object specified as the value is parsed according to the rules defined for the field in the record schema, which must have a valid `Parser`.

For all five methods, the named `Field` must be defined for the record schema associated with the builder and must have the appropriate type for its value. Calling the `addField` method multiple times for the same field name adds distinct instances of the `Field` to the record section. Each resulting instance of the field contains a separate value, resulting in a multi-value field in the index.

Building Record Sections

The `RecordSection.Builder` class lets you create instances of the `RecordSection` object directly, rather than via objects of type `RecordSectionBuilder` and the methods associated with that interface, as demonstrated in the example applications. You may want to create record sections directly, for example, for testing purposes.

The constructor for the `RecordSection.Builder` class accepts three string arguments: the type and identifier of the record with which the record section to be created is associated, and an identifier for the new record section itself. Table 20 describes the methods of the `RecordSection.Builder` class, which are used to specify additional information about the new record section and to build the instance of the section.

Table 20. Methods of the `RecordSection.Builder` Class

Method	Description
`contentType`	Returns a `RecordSection.Builder` that creates a record section with the specified content type in Watson Explorer Engine; the default content type for all records and record sections is `application/vxml`, as defined by the constant `RecordSection.DEFAULT_CONTENT_TYPE`.
`addField`	Returns a `RecordSection.Builder` that creates a record section with the specified `Field`.

Table 20. Methods of the `RecordSection.Builder` Class (continued)

Method	Description
`addFields`	Returns a `RecordSection.Builder` that creates a record section with the specified collection of `Fields`.
`addBinaryEncodedData`	Returns a `RecordSection.Builder` that creates a record section with the specified `BinaryEncodedData`; the overloaded method accepts a single instance or a collection of `BinaryEncodedData`.
`build`	Returns an instance of the `RecordSection` class that has the record type, record identifier, and record section identifier specified by the constructor for the `Builder` class and any characteristics specified by the previous methods of the class.

Indexing Data

Once you have defined your entity model, you can push information into your index that conforms to that model. As described in the previous section, you first create individual records that represent actual instances of entities and fields based on your record schema. You then add these records to the index on your backend Watson Explorer Engine instances so that they can be retrieved by search operations.

The BigIndex API includes a number of classes and interfaces for specifying aspects of the indexing operation. You begin by specifying the options to be applied during indexing. These options control such aspects as batching by number of indexing requests or by the amount of data to be indexed, the maximum number of requests or amount of data held in memory by the client, and the frequency with which batches of indexing requests are sent to the backend server, among other things. You can choose either to tune your indexing operation or to rely on the default values for all options.

You then create an indexer object to insert the records into your index, passing to the indexer the options defined previously to specify attributes of the operation. Indexers are the conduit for all updates to the records in an index. All indexing is asynchronous; as data transitions through the indexing operation, the API notifies you of the status of the indexing request. The status lets you learn whether the indexer has enqueued each record to the backend server, successfully indexed the record, partially indexed the recod with errors, or failed to index the record. You choose how to access the status of your request based on the needs of your application.

You can adopt one of two functionally equivalent approaches to tracking the status of your indexing operation. The first, based on the `DataIndexer` class, uses polling based on a request status object to determine the status of the operation. This approach can be more costly in terms of memory and computing cycles. The second, based on the `CallbackDataIndexer` class, lets you use callback routines to track the state of the operation. This approach enables a simpler and more efficient implementation, letting you focus more on how to respond when the status of an operation changes and less on how to track that status. The greater the number of records to be indexed, the greater the potential gains in efficiency. Both of these approaches can be used with either programmatic or configuration-driven indexing.

The following sections describe the classes, interfaces, and methods associated with indexer options, data indexers, and callback data indexers.

Indexer Options

Indexer options let you control the attributes of your indexing operation. The options define high-level characteristics of the operation in terms of how the client application is configured and how it communicates with backend instances of Watson Explorer Engine. For simple indexing operations such as those demonstrated by the example applications described in this document, the default values are sufficient. For more complex indexing operations associated with production deployments that involve large amounts of data, the options can help you control the operations in meaningful ways.

Indexer options are encapsulated by the class `IndexerOptions`, a container for all of the mandatory and optional configuration options required by an indexer. Instances of the class are created by objects of type `IndexerOptions.Builder`. Table 21 describes the methods of the `IndexerOptions` class. Later sections define the options in more detail and describe the `IndexerOptions.Builder` class, which you use to specify values for the options.

Table 21. Methods of the `IndexerOptions` Class

Method	Description
`getFlushBatchSize`	Returns an integer representing the number of outstanding indexing requests that can exist before the requests are flushed.
`getFlushBatchDataSize`	Returns an object of type `IndexerDataSizeOptions` representing the size of outstanding indexing requests that can exist before the requests are flushed.
`getMaximumInProgressRequests`	Returns an integer representing the number of outstanding indexing requests that the client can hold in memory.
`getMaximumInProgressDataSize`	Returns an object of type `IndexerDataSizeOptions` representing the size of outstanding requests that the client can hold in memory.
`getIndexVerificationBatchSize`	Returns an integer representing the number of indexing requests about which status is to be sought at one time.
`getIndexVerificationIntervalInMS`	Returns a long value representing the maximum length of time in milliseconds the client waits before seeking status about indexing requests.
`getFlushIntervalInMS`	Returns a long value representing the maximum length of time in milliseconds the client waits before sending a batch of indexing requests.
`getRequestTimeoutInMS`	Returns a long value representing the maximum length of time in milliseconds an indexing request can remain in-progress before it times out.
`getIndexingThreadCount`	Returns an integer representing the number of threads maintained for indexing operations.
`getThreadFactory`	Returns the `ThreadFactory` used to create threads internally for the indexing operation.

Indexer Option Definitions

A variety of indexer options are available to provide fine-grained control of an indexing operation. Table 22 on page 42 describes the available indexer options and their default values. The table also indicates the methods of the `IndexerOptions.Builder` class that are used to define the options. The text that follows the table provides additional information about how options related to data size are calculated and about pairs of related options whose values override each other if both are set.

Table 22. Indexing Option Definitions

Option	Description
Flush batch size	The number of outstanding indexing requests that can exist before the requests are flushed. The size is specified by the `flushBatchSize` method. The default is defined by the constant `DEFAULT_FLUSH_BATCH_SIZE`.
Flush batch data size	The size of outstanding indexing requests that can exist before the requests are flushed. The data size is specified by the `flushBatchDataSize` method. The default is defined by the constant `DEFAULT_FLUSH_BATCH_SIZE_MEGABYTES`. Information about how data sizes are specified and calculated follows this table.
Maximum in-progress requests	The number of outstanding indexing requests that the client can hold in memory. If the client is currently processing this number of requests, future requests block until the existing requests finish. The number is specified by the `maximumInProgressRequests` method. The default is defined by the constant `DEFAULT_MAXIMUM_IN_PROGRESS_REQUESTS`. A default minimum computed as the value of the flush batch size multipled by the index thread count is also applied, if applicable.
Maximum in-progress data size	The size of outstanding indexing requests that the client can hold in memory. If the client is currently processing this amount of data, future requests block until the existing requests finish. The data size is specified by the `maximumInProgressDataSize` method. The default is defined by the constant `DEFAULT_MAXIMUM_IN_PROGRESS_REQUESTS_MEGABYTES`. Information about how data sizes are specified and calculated follows this table.
Index verification batch size	The number of indexing requests about which status is to be sought at one time. The operations are asynchronous; this option controls how the status of the operations is obtained. The size is specified by the `indexVerificationBatchSize` method. The default is defined by the constant `DEFAULT_INDEX_VERIFICATION_BATCH_SIZE`.
Index verification interval	The maximum length of time the client waits before seeking status about indexing requests. The operations are asynchronous; this option controls the frequency with which the status of the operations is obtained. The interval is specified by the `indexVerificationInterval` method. The default is defined by the constant `DEFAULT_INDEX_VERIFICATION_INTERVAL_IN_MS`.
Flush interval	The maximum length of time the client waits before sending a batch of indexing requests if the flush batch size or flush batch data size is not reached. The interval is specified by the `flushInterval` method. The default is defined by the constant `DEFAULT_FLUSH_INTERVAL_IN_MS`.
Request timeout	The maximum length of time an indexing request can remain in-progress before it times out. Requests that remain outstanding for longer than the specified timeout are marked as having failed. The timeout is specified by the `requestTimeout` method. The default is defined by the constant `DEFAULT_REQUEST_TIMEOUT_IN_MS`.

Table 22. Indexing Option Definitions (continued)

Option	Description
Index thread count	The number of threads to maintain for indexing operations even if the threads are idle. The number is specified by the `indexThreadCount` method. The default is defined by the constant `DEFAULT_INDEXING_THREAD_COUNT`.

The flush batch size and maximum in-progress data size are specified as objects of type `IndexerDataSizeOptions` (see the following section). For indexing requests, the values to be compared against these thresholds are computed as follows:

- For an *add* or *update* operation, the size is computed based on the sizes of the records and fields being indexed. For fields, the size is calculated as the lengths of the field name and field value strings multiplied by two (assuming UTF-16 encoding) plus the length of the byte array for any binary-encoded data, which is converted into a base-64 encoded string.
- For a *delete* operation, the size is computed as the sum of the record key ID and type length multiplied by two.
- For requests involving *record sections*, the sum also includes the length of the section ID multiplied by two.

Note that two pairs of related options override each other if values are specified for both options. The flush batch size and flush batch data size are one such pair; the maximum in progres requests and maximum in progress data size are the other. In both cases, setting a value for one option of the pair overrides a value already set for its counterpart.

Indexer Data Size Options

Values for the flush batch size and maximum in-progress data size options are specified as objects of type `IndexerDataSizeOptions`. Objects of this class are configuration containers for the data size and its unit. They are created by objects of type `IndexerDataSizeOptions.Builder`.

Table 23 describes the methods of the `IndexerDataSizeOptions` class.

Table 23. Methods of the `IndexerDataSizeOptions` Class

Method	Description
getDataSize	Returns a long value indicating the data size for the `IndexerDataSizeOptions` object, which is specified in the units associated with the object.
getDataSizeUnit	Returns an `IndexerDataSizeOptions.DataSizeUnit` enumerated data type that indicates the unit of measure for the `IndexerDataSizeOptions` object. The data type is one of the constants `BYTES`, `KILOBYTES`, `MEGABYTES`, or `GIGABYTES`.

Table 24 on page 44 describes the lone method of the `IndexerDataSizeOptions.Builder` class. The constructor for the class accepts two arguments: a long value specifying the data size and `IndexerDataSizeOptions.DataSizeUnit` enumerated data type indicating the unit of measure for the specified size.

Table 24. Methods of the `IndexerDataSizeOptions.Builder` Class

Method	Description
build	Returns an instance of the `IndexerDataSizeOptions` class that has the size and units specified by the constructor for the `Builder` class.

Building Indexer Options

The `IndexerOptions.Builder` class lets you create instances of `IndexerOptions` to be used with an indexer. The constructors for the indexer classes require that you specify an `IndexerOptions` object, even if the indexer uses the default options, as is the case with the example applications described in this document.

The constructor for the `IndexerOptions.Builder` class takes no arguments. Table 25 describes the methods of the `IndexerOptions.Builder` class, which are used to specify the options associated with the new `IndexerOptions` object and to build the instance.

Table 25. Methods of the `IndexerOptions.Builder` Class

Method	Description
`flushBatchSize`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified flush batch size. The size is specified as an integer.
`flushBatchDataSize`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified flush batch data size. The data size is specified as an object of type `IndexerDataSizeOptions`.
`maximumInProgressRequests`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified maximum in-progress requests. The number is specified as an integer.
`maximumInProgressDataSize`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified maximum in-progress data size. The data size is specified as an object of type `IndexerDataSizeOptions`.
`indexVerificationBatchSize`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified indexer verification batch size. The size is specified as an integer.
`indexVerificationInterval`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified indexer verification interval. The interval is specified by two parameters: a long value and a unit in the form `java.util.concurrent.TimeUnit`.
`flushInterval`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified flush interval. The interval is specified by two parameters: a long value and a unit in the form `java.util.concurrent.TimeUnit`.
`requestTimeout`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified request timeout. The timeout is specified by two parameters: a long value and a unit in the form `java.util.concurrent.TimeUnit`.
`indexThreadCount`	Returns an `IndexerOptions.Builder` that creates indexer options with the specified index thread count. The number is specified as an integer.

Table 25. Methods of the `IndexerOptions.Builder` Class (continued)

Method	Description
`threadFactory`	Returns an `IndexerOptions.Builder` that creates indexer options that use the specified thread factory to create threads internally. The factory is specified as an object of type `java.util.concurrent.ThreadFactory`.
`build`	Returns an instance of the `IndexerOptions` class that uses any options specified by the previous methods of the `Builder` class.

Indexers

Indexers are the objects that push records into your index. As mentioned previously, the BigIndex API provides two approaches to indexing. Based on two distinct classes, the approaches are functionally equivalent and produce identical results, and both are asynchronous. But each provides a different mechanism for tracking the status of their operations:

- The `DataIndexer` class, which implements the `Indexer` interface, allows you to use polling to determine the status of its indexing operations. Polling can be costly in terms of resource usage and lost efficiency, especially for operations that involve large numbers of records. When using this indexer, application code to monitor the success or failure of the indexing operation must track the status of each individual record. The code thus requires additional infrastructure to manage each request status and to poll until the operation is completion.
- The `CallbackDataIndexer` class, which implements the `IndexerCallback` interface, lets you use callback routines to track the status of its operations. This approach lends itself to a more efficient implementation. You provide an `IndexerCallback` object for each operation, which is then called automatically as requests move through different stages of the indexing lifecycle. Rather than poll and block waiting for each record to complete, the application is notified asynchronously via the callback as each request passes through the states of the operation.

Both the `DataIndexer` and `CallbackDataIndexer` classes provide two constructors. The first constructor takes two arguments: a configuration provider of type `BigIndexConfigurationProvider`, which identifies the Watson Explorer Engine configuration to be used by the indexer, and an indexer options object of type `IndexerOptions`, which defines the attributes of the indexing operation. The second constructor takes an additional third argument, a unique indexer ID prefix, which uniquely identifies the instance of the indexer among all indexers. The prefix can prove useful for identifying an indexer in a human-readable fashion when examining log files, for example; by default, each indexer is identified only by a unique hash value.

Table 26 on page 46 describes the methods of the `DataIndexer` and `CallbackDataIndexer` classes, which are identical in name and function; the differences are discussed after the table. Other than the `close` method, each of these methods can throw an `IndexerInterruptedException` to indicate that its operation has been interrupted. (Deletion of records and record sections is discussed in Chapter 7, “Deleting Data from an Index,” on page 55.)

Table 26. Methods of the `DataIndexer` and `CallbackDataIndexer` Classes

Method	Description
`addOrUpdateRecord`	Adds the specified record to the index if a record with that record ID does not already exist; updates the record by overwriting it in the index if a record with that ID already exists.
`addOrUpdateRecordSection`	Adds the specified record section to the corresponding record if a record section with that section ID does not already exist; updates the record section for the corresponding record by replacing it if a record section with that ID already exists. If the record section is appended to a nonexistent record, the record section is not searchable.
`deleteRecord`	Deletes the specified record from the index, along with any record sections that contribute to it. The record is specified by its type and ID. The operation is considered successful if the specified record does not exist when the operation completes, regardless of whether it existed prior to the call.
`deleteRecordSection`	Deletes the specified record section from the index. The record section is specified by the type and ID of the record with which it is associated and by the ID of the section. The operation is considered successful if the specified record section does not exist when the operation completes, regardless of whether it existed prior to the call.
`waitForAllIndexingToComplete`	Causes the application to block until all indexing operations are complete. Operations submitted after this method is called will potentially block until the indexer completes all of its operations. Two forms of this overloaded method exist. The first takes no arguments and blocks indefinitely; this version of the method returns no value. The second lets you specify the maximum amount of time for which the method is to block; this version of the method returns a boolean value of false if the method times out and true if it does not. The timeout is specified via two arguments: a long value and a unit in the form `java.util.concurrent.TimeUnit`.
`close`	Shuts down the indexer and all of its threads, freeing all resources associated with the indexer. You should construct your applications such that this method is always called, for example, by invoking it from a `finally` clause.

What differentiates the `DataIndexer` and `CallbackDataIndexer` classes is how they report the status for operations that add, update, and delete records and record sections:

- When called, these methods of the `DataIndexer` class return an object of type `RequestStatus`, which you can then interrogate to determine the status of the operations.
- Rather than return a status object, the corresponding methods of the `CallbackDataIndexer` class return no value but require an additional paremeter to specify an `IndexerCallback`, which allows you to respond appropriately when the status of an indexing request changes. The methods of the callback interface

include a RequestStatus parameter; when the callback is executed, you can query the request status object to learn more about the operation that triggered the callback.

Otherwise, the methods of the two classes are identical. "Indexer Request Status" describes the RequqstStatus interface and methods, and "Indexer States and Types" on page 48 describes states and types associated with the class. "Indexer Callbacks" on page 49 describes the IndexerCallback interface and its methods.

Indexer Request Status

Objects of type RequestStatus allow you to obtain information about an indexing operation. In addition to tracking the status of the operation, the RequestStatus object encapsulates information about the record or record section impacted by the operation. Request status objects are returned by calls to the addOrUpdateRecord, addOrUpdateRecordSection, deleteRecord, and deleteRecordSection methods of the DataIndexer class; these methods return the object when first called and then asynchronously update it with the current status as the requested operation proceeds, providing a means for you to query the status of the operation as it progresses. The corresponding methods of the CallbackDataIndexer class require an object of type IndexerCallback; the methods of this interface require a single parameter of type RequestStatus, which you can query to obtain additional information about the indexing operation that triggered the callback.

Table 27 describes the methods of the RequestStatus interface. Note that two versions of the interface are available, one for indexing operations and one for search operations. The version described in this section pertains to indexing operations and is found in the package com.ibm.dataexplorer.bigindex.ingestion.

Table 27. Methods of the RequestStatus Interface

Method	Description
getState	Returns the current RequestStatus.State for the operation that generated this RequestStatus.
getType	Retuns the RequestStatus.Type for the operation that generated this RequestStatus.
getRecordType	Returns a string that represents the type of the record involved in the operation that generated this RequestStatus.
getRecordId	Returns a string that represents the ID of the record involved in the operation that generated this RequestStatus.
getSectionId	Returns a string that represents the ID of the record section involved in the operation that generated this RequestStatus; returns the record ID for the operation if no section ID is associated with the operation.
getIndexerError	Returns the IndexerError for the operation that generated this RequestStatus; returns null if no error occurred.

Table 27. Methods of the `RequestStatus` *Interface (continued)*

Method	Description
`waitForIndexingToComplete`	Causes the application to potentially block until the operation that generated this `RequestStatus` completes. Two forms of this overloaded method exist. The first takes no arguments and blocks indefinitely; this version of the method returns no value. The second lets you specify the maximum amount of time for which the method is to block; this version of the method returns a boolean value of false if the method times out and true if it does not. The timeout is specified via two arguments: a long value and a unit in the form `java.util.concurrent.TimeUnit`.

Calling the `getState` method when using the `CallbackDataIndexer` class is superfluous, since the callback method invoked by the indexer already indicates the state of the operation. Calling either version of the `waitForIndexingToComplete` method makes sense only for operations whose `RequestStatus.State` is `SUBMITTED` or `ENQUEUED`; otherwise, the operation is already complete. (The following section describes request states and types.) Both versions of the `waitForIndexingToComplete` method can throw an `IndexerInterruptedException` if the wait is interrupted.

The `getIndexerError` method returns an `IndexerError` if the state of the operation succeeds only partially or fails. This object encapsulates any error processing that occurred during the operation. You can use the `getIndexerException` method of the `IndexerError` class to obtain the exception that generated the error, which you can then examine to learn more about the failure.

Indexer States and Types

The `getState` and `getType` methods of the `RequestStatus` interface return constants defined by the `RequestStatus.State` and `RequestStatus.Type` enumerated types, respectively. The `State` constants represent the possible states of a request as it passes through an indexing operation. The `Type` constants indicate the nature of the indexing operation associated with the request.

Table 28 briefly describes the possible request states returned by the `getState` method.

Table 28. `RequestStatus.State` *Constants*

State	Description
`SUBMITTED`	The request has been submitted to be indexed but is not yet enqueued.
`ENQUEUED`	The request has been sent to be indexed but completion of the operation has not yet been verified.
`INDEXED`	The request was successfully indexed.
`INDEXED_WITH_ERRORS`	The request was partially successful but experienced some errors.
`ERROR`	The request failed with errors.

Table 29 on page 49 briefly describes the possible request types returned by the `getType` method.

Table 29. `RequestStatus.Type` Constants

Type	Description
ADD_OR_UPDATE_RECORD	The request was initiated with the `addOrUpdateRecord` method.
ADD_OR_UPDATE_RECORD_SECTION	The request was initiated with the `addOrUpdateRecordSection` method.
DELETE_RECORD	The request was initiated with the `deleteRecord` method.
DELETE_RECORD_SECTION	The request was initiated with the `deleteRecordSection` method.

The `State` and `Type` classes that define these constants are implemented as nested static subclasses of the `RequestStatus` interface. Each class has two static methods that you can use to obtain additional information about the enumerated type. The `valueOf` method returns the enum constant of a state or type passed as a string, and the `valuesOf` method returns an array that contains the constants for all of the states or types in the order in which they are declared.

Indexer Callbacks

The methods of the `CallbackDataIndexer` class that add, update, and delete records or record sections require you to pass an indexer callback object. The object you pass must implement the `IndexerCallback` interface. This interface includes four callback methods that are invoked when the operation enters appropriate stages of the indexing operation. The callback methods parallel the states represented by the `RequestStatus.State` constant, with the exception of the `RequestStatus.State.SUBMITTED` state, which has no corresponding callback method: because `SUBMITTED` represents the initial state of an operation, a callback method makes no sense.

Table 30 describes the methods of the `IndexerCallback` interface. Each of these methods requires a parameter of type `RequestStatus`. When the method is called asynchronously during the indexing operation, this parameter provides a means for you to learn more about the specific operation that generated the callback; you can call any methods of the `RequestStatus` interface on the object returned, as described in "Indexer Request Status" on page 47.

Table 30. Methods of the `IndexerCallback` Interface

Method	Description
enqueued	Called when the indexing operation has been queued to be indexed but completion of the operation has not yet been verified. This is equivalent to the state `RequestStatus.State.ENQUEUED`.
indexed	Called when the indexing operation has completed successfully. This is equivalent to the state `RequestStatus.State.INDEXED`.
indexedWithErrors	Called when the indexing operation has completed but was only partially successful, experiencing some errors. This is equivalent to the state `RequestStatus.State.INDEXED_WITH_ERRORS`.
error	Called when the indexing operation has failed with errors. This is equivalent to the state `RequestStatus.State.ERROR`.

Callback methods are called across multiple threads; your implementation must therefore be thread safe. It should also be relatively efficient since an implementation that makes expensive calls or blocks for substantial amounts of time can significantly slow down internal processing of indexing requests.

In addition to these four mandatory methods, you are of course free to define additional methods when you implement the `IndexerCallback` interface. For example, you can declare variables to record the cumulative results of multiple indexing operations and methods to modify and display the results.

Chapter 5. Basic Data Searching

This chapter describes searching for information previously indexed with the Watson Explorer BigIndex Application Programming Interface (API). You can search only for information you have previously indexed, and the entity model and properties you use when indexing directly affect how you search and what you can retrieve. For that reason, the concepts presented in this chapter complement indexing as presented in the previous chapter.

Searching depends on a variety of classes and methods. Some, such as entities, fields, and request status, are the same or very similar to those presented in the previous chapter about indexing. Others, such as entity resolvers, search options, search requests, and search results are logical counterparts of objects discussed in the previous chapter. Like the previous chapter, the descriptions in this chapter rely on programmtic rather than configuration-driven indexing; the following chapter introduces the configuration-driven approach to working with the BigIndex API.

The BigIndex API provides a powerful collection of features for searching an index in a variety of meaningful ways. Depending on how you create your entity model, you can search individual or multiple entities, keywords, fields, bold ranges, dates and date ranges, and you can use regular expressions to parse the information in your index for more complex searches. Depending on your fields and their properties, you can also sort and filter your results, and you can use faceted and weighted searching. The examples that follow demonstrate all of these concepts to increase your comfort with the basic approach to retrieving data from your index.

Chapter 6. Application Configuration Management

This chapter introduces configuration-driven indexing. All previous chapters explored programmatic configuration, where the entity model is based on record schemas established via methods of the Watson Explorer BigIndex Application Programming Interface (API). The examples discussed in the previous chapters also relied on a simple single-collection, single-shard configuration and an embedded Apache Zookeeper server based on supporting utilities provided with the examples. These approaches are sufficient to introduce the API and the basic concepts of indexing and searching.

In a production environment, however, you will want to use a more powerful and robust approach to establishing your entity model and maintaining your configuration. The configuration-driven approach takes advantage of external configuration model files. This XML-based model promotes the use of configuration providers based on multiple Watson Explorer Engine instances. All of the concepts from the previous chapters apply to either programmatic or configuration-driven indexing, but with a configuration-driven application, updates are replicated and requests are distributed across multiple servers for fault tolerance, high availability, data persistence, and better response times.

The chapter begins by describing the use of distributed Zookeeper servers rather than a simple embedded Zookeeper instance. It continues by describing data stores, collections, and clusters. The descriptions include the use of shards and rebalancing data across multiple servers in a growing cluster, all concepts you will employ in an actual deployment. The chapter also describes a health monitor example that can help keep clusters available and route requests to available shards.

Chapter 7. Deleting Data from an Index

This chapter describes how to delete information from a Watson Explorer Engine index by using the Watson Explorer BigIndex API. You delete records and record sections with the methods of the `DataIndexer` and `CallbackDataIndexer` classes introduced previously in "Indexers" on page 45. Such operations are straightforward and not significantly more complex than those described previously.

In addition, the BigIndex API also provides a class that allows you to delete the data contained in a collection store or cluster collection store. The API also includes a second class that lets you delete a subset of data from an entity type based on a matching query. These operations are more complex and thus are the primary focus of this chapter.

Chapter 8. Advanced Topics

This chapter discusses more advanced topics of the Watson Explorer BigIndex Application Programming Interface (API) that are beyond the scope of previous chapters. Earlier chapters focused on basic concepts related to indexing, searching, and configuration management. This chapter covers a broad spectrum of material that builds upon the previous concepts to complete your understanding of the API. While these topics are best presented at this point in the document, they are no less important to your effective use of the API in a production environment.

Among the topics presented in this chapter are the indexing of binary-encoded data; establishing relationships among entity types to enable association queries; and using tokenization definitions in a configuration. Additional topics include security, which is achieved by associating Access Control Lists (ACLs) with fields of a record, and a more detailed discussion of using indexer options to improve the performance of indexing operations.

Note: This document is a work-in-progress and still under development, so the content of many of the chapters is still very much subject to change. This is especially true of this chapter, which could be restructured or split into separate chapters to better suit the material as the content unfolds.

Chapter 9. Example Code

This chapter provides information about the Java example applications delivered with the Watson Explorer BigIndex Application Programming Interface (API). The chapter begins by listing and briefly describing the example code delivered with the API. The descriptions include pointers to the previous chapters in which the individual examples are described in more detail. The chapter continues with a brief discussion of the supporting utilities made available to simplify your use of the examples. It concludes with an overview of how to work with the examples.

Summary of Example Code

Table 31 desribes the Java example code and, where appropriate, its supporting configuration and data files. The table includes references to those earlier chapters of the document that describe the example code in greater detail; refer to the earlier chapters for a more detailed discussion of the examples. All example code is located in the directory `bigindex/examples/bigindex` of the BigIndex installation.

Table 31. BigIndex Example Code

`HelloWorld.java`	
Indexes the simple string "Hello World" and verifies that it appears in the index.	“The HelloWorld Code” on page 7
`ExampleApplication.java`	
Indexes more complex data and verifies that it appears in the index.	“HelloWorld Redux: The ExampleApplication Code” on page 12
`SearchExamples.java`	
Demonstrates different ways of indexing and searching more complex types of various data.	Chapter 4, “Basic Data Indexing,” on page 23 and Chapter 5, “Basic Data Searching,” on page 51
`CallbackDataIndexerExample.java`	
Demonstrates use of the `CallbackDataIndexer` class, an indexing entry point that supports a callback for obtaining the indexing status.	Chapter 4, “Basic Data Indexing,” on page 23
`RecordSectionsExamples.java`	
Shows how record sections may be used to break a record into pieces.	Chapter 4, “Basic Data Indexing,” on page 23
`ZookeeperExample.java`	
Loads a configuration into a set of Apache Zookeeper nodes. Uses the configuration file `ZookeeperExampleModel.xml`.	
`ConfigurationDrivenIndexingExample.java`	
Leverages an entity model stored in an XML configuration file. Uses the configuration file `ConfigurationDrivenIndexingModel.xml`.	
`ClusterExample.java`	
Configures the index for clustering based on field values and retrieves those clusters. Uses the configuration file `ClusterExampleModel.xml`.	

Table 31. BigIndex Example Code (continued)

`storeconfiguration/StoreConfigurationExample.java`	
Manages a Watson Explorer Engine configuration for use with a BigIndex cluster. Uses the configuration files `ApplicationConfiguration.xml` and `StoreConfigurationWithCustomConverter.xml`.	
`ShardExample.java`	
Uses shards across multiple Watson Explorer instances to index and search for data. Uses the configuration file `ShardExampleModel.xml`.	
`RebalanceExamples.java`	
Grows a BigIndex cluster dynamically by adding new Watson Explorer instances to the cluster. Uses the configuration files `RebalanceTwoShardsOneServerModel.xml` and `RebalanceMultipleReplicatedShardsTwoServers.xml`.	
`HealthMonitor.java`	
Uses the HealthMonitor to keep BigIndex clusters available and routes requests only to available shards. Uses the configuration file `HealthMonitorExampleModel.xml`.	
`AssociationExamples.java`	
Shows different ways of performing association queries that leverage relationships among different entity types. Uses the configuration file `AssociationExamples.xml`.	
`SecurityExamples.java`	
Indexes data with Access Control Lists (ACLs) and then securely retrieves the data via BigIndex search APIs.	
`TokenizationDefinitionsExamples.java`	
Shows how to use tokenization definitions in a configuration via the BigIndex API.	
`binarydata/BinaryDataExample.java`	
Indexes binary-encoded data and uses the Watson Explorer Engine converter pipeline. Indexes the example file `sample.pdf`.	
`DeleteStoreDataExample.java`	
Deletes all data from a cluster collection store or from a collection store. Uses the configuration file `DeleteStoreDataExample.xml`.	
`DeleteByQueryExample.java`	
Deletes a subset of a store's data based on the results of a query. Uses the configuration file `DeleteByQueryModel.xml`.	

The directory `bigindex/examples/exampleutils` includes two additional source files, `ExampleBigIndexConfigurationGenerator.java` and `ExampleEmbeddedZookeeperServer.java`, that provide sample utilities that make working with the example applications more convenient. The following section provides more information about these utilities.

Supporting Utilities

Each of the BigIndex example applications summarized in the previous section includes an indexing component and a search component. The complexity of the operations performed varies with the examples, as follows:

- *Simple examples* are configured to use only a single Watson Explorer Engine collection and a single shard, which are provided by the `ExampleBigIndexConfigurationGenerator.java` code in the `bigindex/examples/exampleutils` directory. Such examples are driven completely through their code. Simple examples include `HelloWorld.java`, `ExampleApplication.java`, and `SearchExamples.java`, among others.
- *Complex examples* that involve multiple collections or shards require an external configuration model file that is included in the same directory as the examples. The summary information in the previous section lists the additional configuration, and in some cases data, files provided for use with these more complex examples.

All example applications also rely on an embedded Apache Zookeeper server provided by the `ExampleEmbeddedZookeeperServer.java` code in the `bigindex/examples/exampleutils` directory. This embedded server is provided only as a convenience for working with the examples; it is not intended for use with a production deployment.

See "Supporting Utilities" on page 19 for more information about using the basic sample Watson Explorer Engine configuration and embedded Zookeeper server provided with the examples.

Using the Example Code

All example applications are prepared to run out of the box, with no additional configuration beyond one or more running instances (typically just one) of a Watson Explorer Engine. See Chapter 2, "Installing and Configuring Related Modules," on page 5 for information about installing and configuring Watson Explorer Engine, the Apache Zookeeper server, and the BigIndex API. Once you have completed the necessary installation and configuration steps, follow the instructions in this section to begin working with the example applications.

The simplest way to work with the BigIndex example applications is to use the Eclipse Integrated Development Environment (IDE). The example code in the zip file is configured to work with Eclipse in a seamless fashion that lets you build and run the examples and their dependencies on any supported platform. The root `bigindex` directory of the zip file includes two files, `.project` and `.classpath`, that specify Eclipse project-related configuration information and set up your Java `CLASSPATH` environment variable, freeing you to work with the example code with minimal additional preparation.

Do the following to load the contents of the zip file into Eclipse:

1. Start Eclipse. If you need to download the Eclipse IDE, visit eclipse.org to download Eclipse for your platform and follow the instructions to install and configure the IDE.
2. Select **File->Import**. The **Import** dialog box appears.
3. From the dialog box, select **Existing Projects into Workspace**, then click **Next**.
4. Select the **Select archive file** radio button, then browse to the location of the `bigindex.zip` file in your Watson Explorer Engine installation and select it. The

entry **bigindex-examples (bigindex)** appears next to a selected checkbox under the **Projects** heading in the **Import** dialog box.
5. Click **Finish**. The contents of the `bigindex.zip` file are loaded into your Eclipse workspace under the project name **bigindex-examples**, and a project of that name appears in your **Package Explorer** tab in the Eclipse editor.

You can now use Eclipse to work with any of the example applications delivered with the BigIndex installation. The steps you need to take to prepare an example depend on whether the example is simple or complex, as described in the previous section:
- To use the simple examples, you must update the source code to change *DATA_EXPLORER_ENDPOINT_URL* to identify the URL of your Watson Explorer Engine instance and *DATA_EXPLORER_ENDPOINT_USERNAME* and *DATA_EXPLORER_ENDPOINT_PASSWORD* to specify your username and password for that instance.
- To use the complex examples, you must update the external configuration model file associated with the example to change each occurrence of *url*, *username*, and *password* to identify the URL of your Watson Explorer Engine instance and to specify your username and password for that instance.

Note that some examples, such as `HealthMonitor.java`, do not require these values; such exceptions are noted in the documentation.

For example, to build and run the `HelloWorld.java` example application, do the following:
1. In the Eclipse **Package Explorer** tab, expand **bigindex-examples**, then **examples**, then **bigindex**. The list of Java examples and XML configuration files appears.
2. Double-click `HelloWorld.java` to open the Java source file in the Eclipse editor pane.
3. Edit the source file to change the values of the static string variables *DATA_EXPLORER_ENDPOINT_URL*, *DATA_EXPLORER_ENDPOINT_USERNAME* and *DATA_EXPLORER_ENDPOINT_PASSWORD* as described above, and save your changes to the file.
```
public static final String DATA_EXPLORER_ENDPOINT_URL =
     "http://path-to-data-explorer-engine-instance-1/cgi-bin/velocity.exe?"
          + "v.app=api-soap&wsdl=1&use-types=true";
public static final String DATA_EXPLORER_ENDPOINT_USERNAME = "your-username";
public static final String DATA_EXPLORER_ENDPOINT_PASSWORD = "your-password";
```
4. Right-click `HelloWorld.java` in the **Package Editor** tab, and select **Run As->Java Application** from the pop-up menu to build and run the application.

The application displays output similar to the following showing the results of its simple indexing and searching operations. See "The HelloWorld Code" on page 7 for more information about the `HellowWorld.java` example application.
```
success! Done Indexing Data...
com.ibm.dataexplorer.bigindex.search.model.EntityType@3d4745f3[entityType=hw_entity_type]
-> com.ibm.dataexplorer.bigindex.search.model.EntityId@74219071
   [entityId=9b70d0aa90a98bbb3ee7f4aab8905f96,bigIndexRecordId=RECORD_ID]
        com.ibm.dataexplorer.bigindex.search.model.FieldName@e9434690[fieldName=message]
        --> [com.ibm.dataexplorer.bigindex.search.model.FieldValue@bd843f78
            [fieldValue=Hello World,boldRanges=[]]]
```

These same instructions work for any example application supplied with the API. Recall, however, that for complex examples you need to modify the external configuration model files associated with the examples, as described previously, rather than the Java source files.

Chapter 10. API Reference Summary

This chapter provides information about the packages, classes, interfaces, and methods of the Watson Explorer BigIndex Application Programming Interface (API). The chapter begins by providing general information about the API to help you understand some basic high-level aspects of the interface. It then introduces the components of the API, with pointers to earlier chapters of the document that discuss the components in more detail. At a minimum, you should familiarize yourself with the contents of the first section so that you are aware of aspects of the interface not documented in the previous chapters.

The complete BigIndex API is documented in javadoc reference information installed with the interface. The javadoc is located in the directory `bigindex/javadoc` of the BigIndex installation. The file named `index.html` in that directory is the entry point into the reference documentation. Note that you must unzip the contents of the `javadoc` directory from the file `bigindex.zip` to access the reference documentation in a browser.

General Information about the API

The following sections document some standard interface design practices adopted by the BigIndex API. Because these approaches span the interface as a whole and transcend the basic usage model, they are not documented in the previous chapters. Although they are common Java practices, they are mentioned here to ensure that you are aware of their use by the API.

Common Methods of Classes

All classes of the BigIndex API override a few methods of the Java `Object` class to provide basic functionality across the interface. Table 32 lists and briefly describes these common methods. You are unlikely to call these methods directly on any given class; they are documented here for the sake of completeness.

Table 32. Methods of the `Object` Class

Method	**Description**
`equals`	Returns a boolean value indicating whether the object is equivalent to another specified Java object.
`hashCode`	Returns a hash code value for the object.
`toString`	Returns a string representation of the object.

In addition, many classes of the interface implement the `Comparable` interface. Table 33 on page 66 lists and briefly describes the additional method made available with such classes. As above, you are unlikely to call this method directly, but it is used internally.

Table 33. Method of the `Comparable` *Interface*

Method	**Description**
`compareTo`	Compares the object to another specified instance of the same class as a means of fully ordering the objects. Returns an integer value indicating whether the object is lexicographically less than (negative), equal to (zero), or greater than (positive) the specified instance.

Each class also includes additional methods inherited from the Java `Object` class and any other classes that it extends. Unless otherwise indicated, such methods are not overridden by the inheriting class.

The Builder Design Pattern

The BigIndex API adheres to the *Builder Design Pattern* for object creation. With this technqiue, rather than rely on constructors to instantiate instances of a given class, the parent or outer class is developed with a nested `Builder` class that is used to instantiate an object of that type. The nested class is static to avoid the need to create an instance of the enclosing class. To create an instance of the outer class, you call the `build` method of the nested `Builder` class. This approach avoids a proliferation of constructors for the outer class by including additional methods on the inner class that can be called to specify characteristics of the new instance. Note that many classes that now have nested `Builder` classes continue to have traditional constructors; such constructors persist for backward compatibility but are deprecated to discourage their use.

Examples of classes with nested `Builder` classes include the `SearchOptions` and `SearchOptions.Builder` classes and the `IndexerOptions` and `IndexerOptions.Builder` classes. Examples of their use can be found in the simple `HelloWorld.java` and `ExampleApplication.java` applications.

A related implementation of this model concerns the API's use of a combination of `Builder` and `Factory` classes. In this case, `Builder` classes create instances of objects that implement interfaces. These objects are then passed as arguments to `Factory` classes to create instances of multiple other possible interfaces. This level of abstraction avoids the need for additional classes to create specific objects that implement the interfaces. Examples of this approach include the `RecordSchemaBuilder`, `RecordBuilderFactory`, and `RecordBuilder` classes.

In the first example that follows, from the `HelloWorld.java` code, the `build` method of an instance of the `RecordSchemaBuilder` class creates an object of type `RecordSchema` that uses the entity type *HELLO_WORLD_ENTITY_TYPE* with the specified field and properties. This `RecordSchema` object is passed to the constructor of the `RecordBuilderFactory` class, and the `build` method of the instance of the `RecordBuilderFactory` class is then used to create an object of type `RecordBuilder` for the entity type. The instance of the `RecordBuilder` object then creates a new record with the specified ID and value.

```
RecordSchema recordSchema =
     new RecordSchemaBuilder().addRecordType(HELLO_WORLD_ENTITY_TYPE)
     .addTextField(MESSAGE_FIELD).retrievable(true).build();
RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
...
RecordBuilder logRecordBuilder =
     recordBuilderFactory.newRecordBuilder(HELLO_WORLD_ENTITY_TYPE);
Record logRecord =
     logRecordBuilder.id(RECORD_ID).addField(MESSAGE_FIELD, "Hello World").build();
```

In the second example, from the ConfigurationDrivenIndexingExample.java code, the constructor for the RecordBuilderfactory class is passed the required record schema provided by the instance of the ZookeeperBigIndexConfigurationProvider named *configurationProvider*. After that, the same approach is followed, this time to create records for the entity type *ENTITY_TYPE_TWEET*.

```
configurationProvider = new ZookeeperBigIndexConfigurationProvider(
     embeddedZkServer.getZookeeperConfiguration());
RecordBuilderFactory recordBuilderFactory =
     new RecordBuilderFactory(configurationProvider);
...
RecordBuilder recordBuilder = recordBuilderFactory.newRecordBuilder(ENTITY_TYPE_TWEET);
Record record = recordBuilder.id(RECORD_ID).addField(MESSAGE_FIELD, "Hello World!")
     .addField(DATE_FIELD, "Wed Apr 24 15:50:49 PDT 2013").build();
```

And in the third example, from the RecordSectionsExample.java code, an instance of a RecordSchema named *recordSchema* is created and passed to the constructor for the RecordBuilderFactory class to create both records and record sections for the entity type *RECORD_SECTIONS_ENTITY_TYPE*. The same RecordBuilderFactory object creates objects first of type RecordBuilder and then of type RecordSectionBuilder, which generate records and record sections for the same specified entity type.

```
RecordBuilderFactory recordBuilderFactory = new RecordBuilderFactory(recordSchema);
...
RecordBuilder logRecordBuilder =
     recordBuilderFactory.newRecordBuilder(RECORD_SECTIONS_ENTITY_TYPE);
logRecordBuilder = logRecordBuilder.id(record.get(ID_FIELD));
logRecordBuilder = logRecordBuilder.addField(fieldName, record.get(fieldName));
...
RecordSectionBuilder sectionBuilder = recordBuilderFactory
     .newRecordSectionBuilder(RECORD_SECTIONS_ENTITY_TYPE);
sectionBuilder = sectionBuilder.recordId(record.get(ID_FIELD)).sectionId("sectionId");
```

These examples demonstrate the flexibility of the combined Builder and Factory model to produce objects of different types from a compact set of classes.