DARWINO

Developer's Guide

Table of Contents

Darwino User's Guide

Introduction

Welcome to Darwino! Darwino is an open platform for rapidly developing social business applications targeting primarily Mobile and the Cloud. Importantly, Darwino allows developers to focus on the application, and not on platform specific issues or wiring to other platforms/data sources. Because of this, Darwino greatly reduces the time needed for you to deliver value to your customers. Darwino's full stack of ready-to-run components and connectors allow you to design applications that run locally on any mobile device, connected to any back-end data source, with next to no platform-specific coding. Finally, Darwino leverages the skills of any Java developer - allowing Java developers to move to build enterprise mobile applications with no additional training.

This developer's guide will help you learn the specifics of Darwino, and to build applications in no time. The following is a high-level overview of Darwino that will get you kick-started.

Darwino Editions

There are two editions of Darwino: The Community Edition and the Enterprise Edition.

- Darwino Community Edition

  The Darwino Community Edition is designed for non-commercial developers. Compatible with any free database plan or trial plan, it offers a wide range of features:

  - Extensive database support, including a no-charge RDBMS, IBM DB2-C 10.5+, PostgresSQL 9.4+ Open Source, IBM BlueMix, SQLDB, Compose, and Elephant
  - Enterprise Directory support for JNDI LDAP access, IBM BlueMix, and static files
  - IBM Domino Connector with synchronization of up to three NSFs per instance of each Domino server
  - Access to all major releases, with community-based support

  In addition to these features, Darwino will be extending database support to include Oracle Express Edition, SQL Server Express, and more in the near future.

- Darwino Enterprise Edition

The Darwino Enterprise Edition is intended for commercial developers and others who depend on periods of intense evaluation as they consider deployment of applications of their own. The Darwino Enterprise Edition includes everything available in the Community Edition as well as:

- Support for all enterprise RDBMS
- No limits on the number of database instances
- Encrypted Mobile SQLite
- Unlimited Domino connectors
- Drivers for IBM Connections (on premise and cloud) and VMM (running in IBM WebSphere)
- Optimized applications with Grunt/Gulp/Bower
- Access to all releases, including nightly builds
- Darwino Support

Darwino Architecture

Darwino is specifically designed and oriented around the use of Java, on both the server side and the mobile side. As such, your applications are built in pure Java, using POJOs, while deploying the application to multiple platforms is handled by the Darwino platform. Because of this, many of the components that Java developers are accustomed to remain consistent when developing for Darwino, including:

- The use of a standard RDBMS for data storage (although the Darwino DB JSON Store is much more than a typical RDBMS)
- LDAP directory authentication and authorization
- Extendable platform components using standard Java techniques
- Runs on top of standard JVM Servlet Container
- Add-in for Eclipse developer studio
- Maven compatible/compliant, so you can use whatever development tool you choose
- Write once, run on Web/Mobile/Hybrid, with automatic porting of application to Android and iOS

Required Infrastructure

Supported Database Platforms

- Postgres DB 9.4+
- IBM DB2 10.5 +
- Microsoft SQL Server 2016+
- Oracle Database 12c+
- SQLite (for mobile, provided by Darwino)

Webserver

- Any servlet container that is Servlet 3.0+ compliant (e.g., TOMCAT, IBM Websphere, IBM Websphere Liberty)

Supported Cloud Deployment Platforms

- IBM BlueMix

Supported Mobile OS

- Android 4.4+
- iOS 8.1+

Supported LDAP servers

- Microsoft Active Directory
- Oracle Directory Server
- IBM Domino
- IBM Tivoli Directory Server

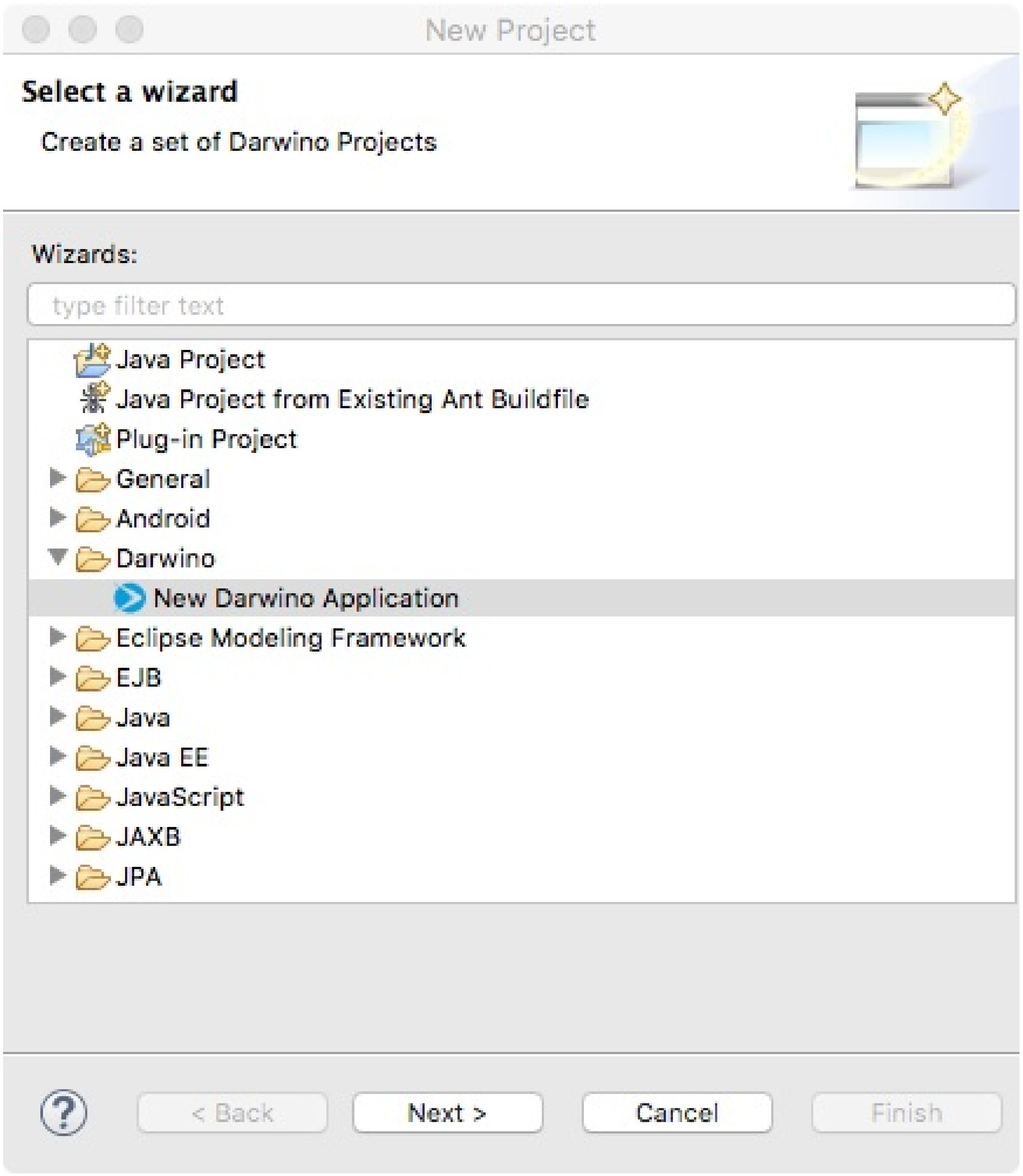
Installing a development environment

Installing a development environment is decribed in the Installation guide, available here: https://playground.darwino.com/playground.nsf/Doc_InstallationGuide.xsp

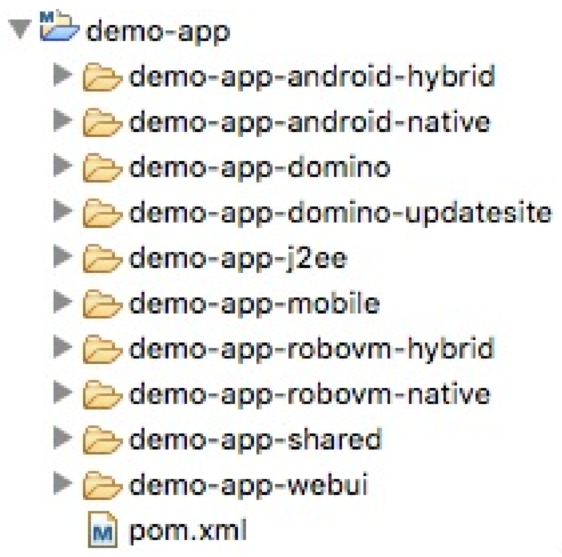
Using the Studio - Creating your first Darwino application

The Darwino Application Wizard

The Darwino application wizard is the first step in creating a Darwino application.

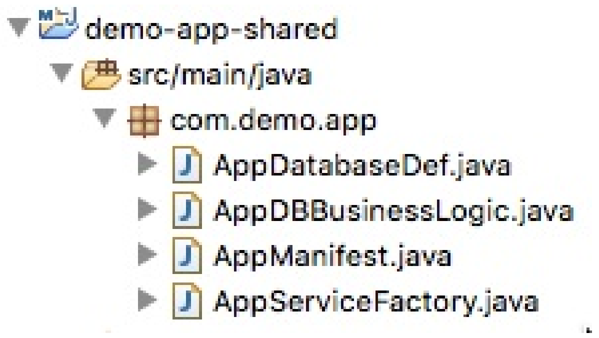


The Wizard generates a set of Maven projects. The top project is the container for the other projects.



. Which projects are generated depends on the options that were selected in the wizard.

- -shared: This project contains the Java code that is shared by all the platforms.



-- AppDatabaseDef.java defines the metadata of the JSON store. This metadata is used when the database is initially created, and then anytime you need to make a change.

The first time replication runs, the tables will optionally be created automatically. It can also check to ensure that the tables are at the required level. If the database and the DATABASE_VERSION are equal, it will proceed. If the table version is higher than expected, an error will be raised. If the table version is lower, you can upgrade the tables (if autodeploy was selected), or raise an error.

In the enterprise, it is often the case that the database developer does not have the authority to create and modify tables on the J2EE server; that task is restricted to the database administrator. To accomodate this, Darwino provides the AppDDLGenerator, a class that will create the DDL text file that can then be given to the administrator for processing. The DDL file is created considering the database definition, any customizations, and the database provider. The developer will run the AppDDLGenerator in Eclipse, and it will produce the DDL file in the console.

-- AppDBBusinessLogic.java provides the means to handle database events. Examples include the Document events (create, edit, delete) and database replication events.

-- AppManifest.java defines the options for the Darwino application layer. The options defined here are shared among all platforms. For example, the getDatabases() method returns the database names used by the application, and getLabel() returns the database label for J2EE.

-- AppServiceFactory.java - In a Darwino app, all of the business logic is isolated into services that can be exposed as web services. By default it provides you with a set of services, such as the JSON store, but it also allows you to create your own services. It is in AppServiceFactory where custom application services are defined.

The wizard generates a very basic, example skeleton service that is ready to use.

There is also the RestServiceBinder which maps URLs to different services, in a platform/library independent manner.

- -webui: This project is generated when the wizard is instructed to create a J2EE or hybrid app. It contains the web artifacts that are consumed by the web application. It’s not under “shared” because we can have apps that are not web apps, for example pure backend apps on the server, or native apps for mobile devices.

  It is a basic skeleton using web technologies and AngularJS. AngularJS is not a requirement, but it is recommended. The index.html file that’s generated includes AngularJS code, but you can delete it and use whatever framework you like.

  The web resources are packaged under a folder called DARWINO-INF under src/main/resources (a Maven convention for where to put resources).

  Darwino comes with a custom service that is able to read the resources inside that directory and act as though they were part of your J2EE project and part of your mobile assets.

- -j2ee: This is a J2EE wrapper to this project. It includes the Java in the Java runtime and it includes the web resources in the runtime.

  The pom.xml defines the dependencies of the project. In our case, it shows that we’re including the demo-app-shared and demo-app-webui, thus all the content in both projects will be included.

In the web.xml, we can point to Darwino artifacts, which are either servlets or filters. Several of these are of particular importance. The DarwinoJ2EEFilter handles on-the-fly transformation of url requests, allowing the folder structure to vary without requiring code modifications to accommodate the changes, and making urls platform-independent).

Defined in web.xml:

- DarwinoAppResourcesServlet
- DarwinoJ2EEFilter: When a request comes in, this is the first filter to be activated. It retrieves the current context and puts it on the stack so it becomes available to the application. It does this for every URL.
- DarwinoGlobalPathRewriterFilter translates all of the URLs containing $darwino-xxx/... into /.darwino-xxx/... paths. All the predefined Darwino services are now mapped to /.darwino-. This makes the url completely platform-independent.
- Darwino services: for example, built-in services like the JSON store, or custom (user-created) services served by the DarwinoServiceDispatcher filter.

In order for the highest level, the Darwino application, to have the context it requires, at application initialization the com.demo.app.AppContextListener is triggered before anything else. It provides the application with access to all of its environment information. As a listener, it cannot pass parameters, but the application can have global parameters, so the listener uses global parameters to pass context information

Authentication

The wizard generates code allowing the use of J2EE authentication, but this form of authentication requires the J2EE server to be connected to your directory of users, and every web application server has its own mechanism for doing that; this approach does not lend itself to true portability. Also, J2EE authentication lacks granularity.

Darwino comes with its own authentication filter than you can choose to use. This allows us to use our own user directory regardless of the application server.

There are two other files in WEB-INF that are critical in solving the problem of configuring application parameters such as database connections and logging. Rather than storing this configuration within the application itself, we can externalize the configuration. This allows configuration changes to be made without recompilation. It's up to the developer to decide where these files are stored, but, in keeping with J2EE convention, Darwino stores these files in the WEB-INF directory.

- darwino-beans.xml These are Java objects, and, as managed beans, the platform will automatically create instances of these objects when needed. Contains sections for:
    - Database access
    - IBM Connections endpoint
    - HttpTracer – allows tracing of all aspects of the communication between client and server
    - Static directory of users

  Each bean has a type and a name, and can have an alias list, which can include the alias "default".

- darwino.properties There is an API in Darwino to get these property values.

- -mobile: This project doesn't run anywhere. It is a container for resources that are common to the mobile platforms (currently iOS and Android). AppMobileManifest.java is like AppManifest, but specifically for mobile devices. It adds a set of options that are very specific to mobile devices. The wizard provides only the basic skeleton. The developer puts the common mobile code and resources here.

- -android-hybrid: Depending on wizard choices, you can have an Android-hybrid and/or and Android native. Typically you'll choose one and not both. We are not showing Android native in the demos. This project depends on all the others except the J2EE app. It includes all the Android assets, such as the AndroidManifest.xml. It is exactly like a project generated with the Android wizard, except it includes a set of dependencies on the Darwino project.

  AndroidApplication.java is a core Android SDK object that initializes the Darwino Application object, as well as the context.

  DarwinoApplication.java contains two very important objects:

    - DarwinoApplication is a singleton that acts as the entry point for all of the objects of the Darwino application, including the manifest, and it enables the triggering of Darwino application actions such as replication.
    - DarwinoContext, which is generated by the Darwino runtime. It has a different behavior depending on the platform, but gives access to the same information.

A mobile environment is single-user, while the server environment is multi-user. In both cases, there will be one Darwino application which is exactly the same regardless of the user, however the DarwinoContext on the J2EE server is the context of the current request (“Who is the user for this request? What is the contextual environment–the execution environment–of this request?”). On the server, there is one new DarwinoContext created per request. On the mobile device, none of this applies.

Of course, if you want to just use the JSON API to get this context info, you can, but these objects make the job much easier.

DarwinoServiceDispatcher: this is the class that is used by the mobile local HTTP Server to dispatch the services. By default, all services are enabled, but with DarwinoServiceDispatcher we can selectively enable and disable services.

MainActivity: This is an Android SDK object. This implementation creates an embedded Android browser.

SplashScreenActivity provides the splash screen.

- -moe-hybrid: This is the iOS version of the app. We’re writing Java for iOS, but Java is not supported natively by iOS, so we rely on 3rd-party frameworks. We currently support Multi-OS Engine (MOE). As of Darwino 1.5, RoboVM is no longer supported as the product as been acquired and ceased by Microsoft.

  Just as we generate an Android wrapper, we generate a MOE one. The contained classes match those of the android-hybrid, with MainViewController equating to MainActivity.

Important Concepts

Platform agnosticism is central to the Darwino philosophy. In this section we will look at several of the techniques Darwino employs to ensure that your applications will work across a variety of server and mobile device platforms with a minimum of programmer effort.

The Platform Object

Darwino applications execute on multiple platforms, including iOS, Android, JVM, and a Web Container. Running code must handle differences in these platforms. The Darwino Platform Object manages these platform differences so that you do not have to worry about them.

Darwino is built to be platform agnostic, and exposes everything in the architecture as pluggable. The foundation of this capability is the Darwino Platform object, which encapsulates all platform-specific functionality. When working with multiple platform development, even simple tasks such as logging are managed differently depending on which platform the application is currently running. The Darwino Platform object allows you to interact with platform capabilities such as logging in an abstract manner, without having to determine the execution platform.

> NOTE: The Darwino Platform (com.darwino.commons.Platform) is instantiated as a singleton object.

Services

The Platform object provides several default services. See Services and extensions for details.

Developers can extend the Platform object using custom services and extensions.

> The Platform object is the entry point for all services, and the Platform object makes all services available from anywhere in your application. Further, any library can contribute services, as Darwino services are POJOs, and do not need to extend any particular interface.

Platform Configuration

In order to maintain platform agnosticism, Darwino does not depend on the use of fixed configuration files; instead, it is the Platform object that provides access to configuration properties.

Services and extensions

There are two elements in the platform: services and extensions. Services and extensions are created lazy, when they are needed for the first time.

Services and extensions are defined by plugins, or by the Darwino Application object. A plugin is a class that can be provided either by libraries or by your application directly, and they can register services and extensions to the platform.

Services

A service is a singleton. For a particular class of service, there will be ONE implementation. If there are multiple implementations, you will get a runtime error.

Extensions

Services and extensions differ in that an extension can have multiple implementations and they can all work together at the same time, like, for example, a database connection. You can have a connection to Postgres and a connection to DB2 and use both of them within the same app. An extension is like a service but it can have multiple instances that are non-exclusive.

The Darwino wizard will generate a skeleton plugin for your application that you can use to override or add to the default platform implementation.

Logging

Darwino provides its own logging library for each platform, and dynamically chooses the appropriate one for the current platform. This makes it completely transparent to the developer; your logging code will be fully cross-platform.

```
// General logging
Platform.log("Logging {0} unconditionally", "My Message");
Platform.log(new Exception(),"Logging an Exception, {0}", "Here it is");
```

The log() method will print unconditionally to the console when running on the web application server, or to the Android or iOS logging mechanism when running on the mobile device.

Log groups

While Platfom.log is the basis for logging in Darwino, we can also use the more powerful concept of log groups. Log groups in Darwino are an abstraction of the various platforms' built-in logging mechanisms.

A group is a named object, typically with a hierarchical name of the form "a.b.c". You can log different classes of messages, such as information, warnings, errors, and debug information to a group. Then you can enable a particular a logging level for a specific group.

For example, you could create a log group named "archive". You could then refer to "archive.info" and pass some strings or content. You could enable this group for a particular level of logging, for example warnings, or errors plus warnings, or errors plus warnings plus information, or everything including debug information.

Generally, you define hierarchical groups; this creates the possibility of functional logging. You could then, for example, enable logging only for "archive.permanent" or "archive.permanent.*".

Properties

It's good to be able to externalize some application properties. You can pass properties to an application and then access these properties through a service. The property service, getProperty() method, and putProperty() method are the means for working with these externally-defined properties.

The properties are defined in a manner similar to how managed beans are defined, except instead of being in the darwino-beans.xml file, properties are in the darwino.properties file. The properties file can be at the same file system paths as the darwino-beans.xml, and can be available through a JNDI call.

Property references in web.xml

It's possible to reference Darwino-related properties, defined within the web.xml file, via the Darwino Application Listener.

Here, we see a paramater value being extracted from a named property:

```
<context-param>
    <param-name>dwo-sync-trace</param-name>
    <param-value>${discdb.sync-trace=false}</param-value>
</context-param>
```

Darwino will check the platform for a property called "discdb.sync-trace" and return its value. If it is not found, it will use the default value, which is "false" in this example.

Managed Beans

A key component of the Darwino Architecture is the concept of managed beans. This concept is borrowed from, although different from, Spring and JSF. Managed beans are manageable resources – Java objects that are instantiated by the platform when needed, and destroyed when they go out of scope.

Common managed beans include database connections, user directory connectors, and several services, such as the mail service.

Darwino uses managed beans primarily as a generic way to configure the platform.

Accessing a managed bean

A managed bean is defined by a type, a name, and, eventually, aliases. For example, when the JSON store runtime looks for a database connection, it searches for a bean of type 'darwino/jsondb', with a specific name. The name generally comes from the Application object which defines the names to use by this application. When multiple names are used, then the runtime searches for a bean with the first name and, if it is not found, it tries the other names in order until it finds a bean that matches.

Configuring Managed Beans

Managed beans are provided by extension points. You can define your managed beans' location as a custom extension. Where managed means are loaded from depends on the DefaultWebBeanExtension class. The class looks for beans in various places.

It first looks using JNDI. It looks for an entry in the path java:/comp/env/darwino-beans. If it finds one, it will either be a text file or a url pointing to a file. Either way, it will interpret the XML found there and load the bean accordingly.

Next it looks to the web application server, following the conventions for the various application servers.

After that, it looks in the classpath. It looks for a file called darwino-beans.xml within the current classpath.

Then it looks in WEB-INF for a darwino-beans.xml file. In addition, it will search there for a darwino-beans file with a name determined by the application's configuration files suffix, such as "bluemix'. The resulting file that Darwino will look for would then, in this case, be "darwino-beans.bluemix.xml". Thus, specifying the suffix determines which darwino-beans files will or will not be loaded.

After that, if there is a system property called "darwino-beans" containing either XML or a URL, then Darwino will load the beans specified within.

Finally, Darwino will look for an environment variable called "darwino-beans", and it wil process it just as it does the similarly-named system variable mentioned above.

Managed beans are configured using the following xml structure:

```
<bean type="[defined bean type]" name="[unique bean name]" class="[full class name
]"
    alias "[optional alias names, separated by comma"/>
    <property name='[property name]'>[property value]</property> //list of propert
ies
</bean>
```

Managed bean configurations can instantiate multiple sets of bean objects lists simply by using a list tag as follows:

```
<bean type="[defined bean type]" name="[unique bean name]" class="[full class name
]"
    alias "[optional alias names, separated by comma"/>
    <list name = "[list name]">
        <bean class='[class name]'>
            //property list for instance
        </bean>
        <bean class='[class name]'>
            //property list for instance
        </bean>
        <bean class='[class name]'>
            //property list for instance
        </bean>
        //etc.
    </list>
</bean>
```

For nested beans, and if the bean class is an inner class of the main bean, the class name can simply be ".InnerClassName". In the example below...

```
<bean  class="com.acme.business.Finance" ....>
<property name="account">
  <bean  class="com.acme.business.Finance.Account" ....>
```

...the last statement can be shortened to:

```
  <bean  class=".Account" ....>
```

A list can also match a Java array, and Maps can be used as well:

```
<map name="properties">
    <entry key='....'>...value...</entry>
    <entry key='....'>...value...</entry>
</map>
```

As you can see, this structure is intentionally generic. While the bean type must be defined based upon a definition, the remainder of the definition is completely generic. This allows for any type of object to be defined as a bean and managed by the platform.

If a call is made to a managed bean, the platform checks to see if the object exists and, if not, it instantiates the object according to this definition file, using the class name and configured property definitions. This leads to Darwino applications being defined in a very flexible and generic manner.

Property references in managed bean definitions

When the Darwino code that parses managed bean definitions encounters a property name, it looks first to see if it is defined in the file as a local property. If it is not defined locally, it will then look to the platform for a property of that name. Finding it, it will use its value in the managed bean definition.

This allows you to drive the creation of the managed beans from outside the application.

Property references are of the form:

```
${propertyname[=default value]}
```

The goal with all of this is to have a WAR file that is customizable from outside the application, without having to repackage the application.

Scope of Managed Beans

Managed beans have a defined scope. By default, if you do not provide a scope, the scope will be considered global, which means that the bean will be a singleton object, with a single instance for the entire application. Other scope choices are:

- **None:** A new instance of the bean is created on every call to the bean. The bean object is discarded after every call.
- **Request:** A new instance of the bean is created on every request, meaning the developer can call the bean, then call multiple methods on the same instance of the bean. Once the bean object is out of scope, it will be discarded,
- **Session:** A new instance of the bean is created and persisted for each session. Once the session is discarded, the bean object is discarded.
- **Application:** A new instance of the bean is created for each calling application in a particular class loader.

JSON library and data binding

A lot of Darwino is based on top of JSON, and particularly the JSON store. There is no standard JSON library in Java, and the ones that are available can be inconsistent and incompatable with one-another; Darwino deals with this in two ways:

- The Darwino library provides a JSON Factory, which sits atop your libraries of choice and encapsulates all the features you need to manipulate JSON. It provides a uniform view of JSON - a common API on top of disparate API implementations.
- Darwino also provides its own library for JSON. While other libraries can be used via the factory described above, these libraries tend to be poor performers, cumbersome, or just heavy-weight. Their dependencies make for heavy mobile app code. Also, it is not always convenient to have to work though the extra layer provided by the Darwino-provided adaptor.

The Darwino library is based on top of one written at IBM. It is 100% compatible with the JSON standard, and has been optimized for JSON parsing, and it includes JSON extensions. This library is used all over the Darwino product, and it makes it easy to deal with and consume JSON objects and arrays.

Because of this, best practice when manipulating JSON in your Darwino application is to use the default JsonJavaFactory instance and its JsonObject and JsonArray classes.

The JsonObject is a Map of string/object; the keys are strings and the values are objects. This means that if you have a function that is expecting a Map as a parameter, you can pass it a JsonObject.

The JsonArray object is a list of values implemented as a Java List. It is a List of objects. It has all of the methods you'd expect from a JSON array, but it is also a List, so if you have a function expecting a List of objects, you can pass it a JsonArray as a parameter.

To make JSON values easily consumable in Java, a string inside a JsonObject is a Java String, a number is a Java Number, and a boolean is a Java Boolean.

In standard JSON, we have key/value pairs. Keys must be wrapped in double quotes. JavaScript, though, allows single quotes or key names with no quotes at all. The Darwino JSON parser is permissive and allows those. When it serializes, it does so according to JSON standard, but when it reads, it is permissive.

```
 // A JSON parser is available through a Factory
JsonFactory f = JsonJavaFactory.instance;
JsonObject jo = (JsonObject)f.fromJson("{a:11, b:12, c: 13}");

// Json objects/arrays have easy-to-use methods
_formatText("JSON Object, compact: {0}",jo.toJson());
_formatText("JSON Object, pretty: {0}",jo.toJson(false));

// Or this can be done through a factory
_formatText("JSON Object, compact: {0}",f.toJson(jo));
_formatText("JSON Object, pretty: {0}",f.toJson(jo,false));
```

Command Insertion

Darwino takes advantage of JSON comments to provide a facility to insert commands in JSON. Comments in JSON is an extension to the JSON standard, and is supported by Darwino. Similarly to JavaScript, the parser supports single-line (//) and multi-line (/ *... /*)
comments.

Standard JSON has no provision for inserting commands. Darwino's JSON implementation supports JavaScript-style comments as an extension, and the parser can read inside these comments. This feature enables reading and writing commands embedded in the JSON.

When a comment in JSON starts with "/*%=" the Darwino JSON interpreter will interpret the JSON object that follows as a command. It will parse the contents and return the result to whatever called the parser. The parser will always be looking for inersted commands, but if you haven't registered a callback to handle the commands it will do nothing.

A typical use case for this is a progress bar. Imagine that a very large JSON file is being returned to the client via REST services. When the client tells the server that it supports commands, the server can emit comments in the JSON that describe activity progress. The client can use those progress comments to display a progress bar.

A command is generally a notification from the code that generated the JSON payload, as in this example to track the progress of a file. It takes the form of a JSON value, containing data that can be interpreted by the consumer. For example, it can be the current progress in the whole file, like:

```
/*%={label:'Updating table TTT', progress: '56%'} */
```

It is up to the consumer to interpret its content appropriately.

The Darwino JSON parser can interpret commands when reading, but does not, by default, emit them. To enable insertion of commands in JSON output, specify OPTION_PARTIALPROGRESS when creating the JsonWriter. The HttpServiceContext, when processing an HTTP request, will then look for a header with a value of "x-dwo-json-progress" from the caller, and only if it sees that header will it insert the commands in its output.

JSON Compression

We can also emit compressed, binary JSON in place of text. Darwino does not use this externally, as when writing to a file or to a database, but it can be used when communicating via HTTP. For example, when the client is replicating with the server, it uses a REST API that is based on JSON. If the client sends the server a header saying that the client understands the binary form of JSON, then it can compress the data. Values are compressed, and names can be sent once and subsequently only pointed to. Also, this removes the need for parsing the data.

JSON Query Language

There is a query language for JSON, allowing you to quickly and easily query JSON data. The query language is actually a JSON document itself and is super-subset of the MongoDB one. This language is used throughout Darwino.

Here is an example showing the query language being used to populate a cursor based on JSON content:

```
  Cursor c = store.openCursor()
     .query("{$or: [{state:'MI'},{state:'TX'}]}");
```

JSON Data Extractor

The JSON Data Extractor facility is similar to the query language, but used to extract fields and values from a JSON document:

```
Cursor c = store.openCursor()
    .extract("{first:'firstName',last:'lastName'}").range(0,5);
```

See Appendix 3. The Query Language for details on the Query Language and JSON Data Extractor.

HttpClient

Because a lot of activities in Darwino are based on REST services, we need a means to connect to those services. We need to connect to them from Java, and not only from the browser. The Darwino HttpClient is very easy to use. It has classes for authentication and for handling JSON. For example, to call a REST service, pass some parameters, and get a result back, all that is required is to call getAsJson(). It will handle the job of composing the proper URL and processing the result. In Java, the HTTP calls are executed synchronously.

```
String url = "http://localhost/playground.nsf/playground/$darwino-jstore";
HttpClient c = ((HttpClientService)Platform.getService(HttpClientService.class))
                      .createHttpClient(url);

// Call the information service and interpret the result as JSON
//    <base-url>
Object r = c.getAsJson(StringArray.EMPTY_ARRAY);
_formatText("JSON Store information: {0}",r);

// Call the user service and interpret the result as JSON
//    <base-url>/user
Object r2 = c.getAsJson(new String[]{"user"});
_formatText("JSON Store User: {0}",r2);
```

The Darwino HttpClient is built on top of the platform's own client; the developer doesn’t have to worry about the specifics of the underlying HTTP client.

The HttpClient also handles GZIP, ChunkedPost, and multi-part MIME.

Darwino application objects

Darwino runtime objects exist for every Darwino application and are created appropriately by the runtime. They are organized in a class hierarchy based on the runtime platforms. The highest class in the hierarchy represents the features common to all platforms, while the deepest ones are specific to the runtime platform.

Typically, the hierarchy looks like:

- Darwino
    - DarwinoJ2EE
    - DarwinoMobile
        - DarwinoIOS
        - DarwinoAndroid

Application

The Darwino application is a singleton. There is one–and only one–running instance. The Application instance is the entry point for all of the application options, including the manifest (the Application's getManifest() method will return the manifest object).

It is through the Application object that we can perform application-wide actions such as triggering replication.

At any time, the developer can get access to the current application by calling DarwinoApplication.get(). If you know the platform you're running in, and if you want to get access to platform-specific features, then you can access the specific application by casting the application object to the platform-specific class, or by calling get() on this class (ex: DarwinoJ2EEApplication.get()).

Darwino application objects

Darwino runtime objects exist for every Darwino application and are created appropriatly by the runtime. They are organized in a class hierarchy based on the runtime platforms. The highest class in the hierarchy represents the features commons to all platforms, while the deepest ones are specific to the runtime platform.

Typically, the hierarchy looks like:

- Darwino
    - DarwinoJ2EE
    - DarwinoMobile
        - DarwinoIOS
        - DarwinoAndroid

Application

The Darwino application is a singleton. There is one and only one running instance. The Application instance is the entry point for all of the application options, including the manifest (the Application's getManifest() method will return the manifest object).

It is through the Application object that we can perform application-wide actions such as triggering replication.

At any time, the developer can get access to the current application by calling DarwinoApplication.get(). If you know the platform you're running in, and if you want to get access to platform specific features, then you can access the specific application by casting the application object to the platform specific class, or by calling get() on this class (ex: DarwinoJ2EEApplication.get())

Manifest

The Application Manifest generated by the Darwino wizard defines the options for the Darwino application layer. The options defined here - such as the database names used in replication, the label and description of the application, and the application's url - are shared among all platforms.

```
DarwinoApplication app = DarwinoApplication.get();
DarwinoManifest mf = app.getManifest();

_formatText("Label: {0}",mf.getLabel());
_formatText("Description: {0}",mf.getDescription());
_formatText("Main Database: {0}",mf.getMainDatabase());
```

In addition to the global manifest, common to all of the platforms, there are per-platform manifests (J2EE, Mobile). These manifests hold the options specific to their particular platform.

Context

The Darwino Context provides the application with access to all of its environment information, including the user identity and the properties of the execution environment.

The Context has a different behavior depending on the platform, but it provides access to the same information.

```
DarwinoContext ctx = DarwinoContext.get();

_formatText("User: {0}",ctx.getUser());
_formatText("JsonStore Session: {0}",ctx.getSession()!=null);
```

In a web environment, there is one context object created per request. On a mobile environment, it is a singleton.

At any time, the developer can get access to the current context by calling DarwinoContext.get(). If you know the platform you're running in, and if you want to get access to platform-specific features, then you can access the specific context by casting the context object to the platform-specific class, or by calling get() on this class (ex: DarwinoJ2EEContext.get()).

Darwino DB API

The Darwino database is a NoSQL store for storing JSON documents and related attachments. It is actually a database of JSON documents, and you can attach binary pieces to any document.

Built on top of existing relational databases, it provides the benefits of a mature, well-established technology. Is it fully transactional, and it scales to very large datasets. It also inserts well into the existing infrastructure, leveraging existing processes such as backup procedures and security.

Darwino takes advantage of the latest NoSQL additions, such as native JSON and data sharding.

Another benefit of using relational databases as the groundwork is that you can use any existing tool that can connect to a relational database to access the data. You can use BI tools, reporting tools, or other big-data analysis tools, like IBM Watson, and they will directly understand the database format.

One of the key points of the JSON store is that there is a single implementation that works everywhere: servers (Windows, Linux, OS X, etc…), and mobile devices. The same code runs in both environments. Moreover, these disparate environments can replicate data.

Darwino DB API Concepts

Server

Even on a mobile device, you have the Server. The server is the entry point to the database. All database operations start from there. The Server is, in effect, the database itself; the Server encapsulates the connection to a physical RDBMS. It points to your DB2, your SQL Server, your PostgreSQL, or to your SQL Lite. When you connect to an RDBMS on the server, you will specify the JDBC path and user password that the runtime can use to access the data.

In a multi-user environment such as on the web server, the same physical JDBC connections will be used across users. The data access security is then managed by the Darwino code.

Session

The Session object allows different users to share the same server, as in a web application supporting multiple simultaneous users. In a web environment, there is one Session object created per request, and discarded when the request is completed. The Session object carries the user credentials, and is available through the DarwinoContext object.

The app itself will use one database server but multiple Sessions. The Session, created from the user's ID, is what will be used by the runtime to apply security to the data. This is an important concept because the server connects to the physical relational database with one single user and password, which means that this user and password can access all of the data. The security is then applied by the Darwino runtime through the Session object.

Database

Then comes the organization of the data. The data are organized into databases; but we should not confuse that with the relational database file. A database is a logical structure physically backed by a set of relational tables in the RDBMS. In Darwino, it's a document database. A Darwino database is a set of documents with common characteristics.

Stores

The documents inside a database are organized in stores. The store is a container for JSON documents. This allows you to put documents into different “buckets” that have different characteristics, such as different indexing strategies. For example, in a CRM application, you may have one document that is a Customer, and another that is a Product. You would not want to apply the same indexes to these documents. You can specify further customization for a store, such as whether full-text search is enabled.

Pre-defined stores

In the database definition are four pre-defined stores. These stores are created automatically by Darwino and can be utilized as-is, but they can also have their definitions overridden and customized (by adding indexes or fields, for example). It is always possible to create specific stores for these stores’ purposes; these pre-defined stores exist as a convenience.

- _default: This store can act as a placeholder. There is no index or field extraction for this store by default, but it can be used for quick-and-dirty storage of data.
- _local: This store is identical to the _default store, except that it is never replicated. It is useful for-among other things-storing device-specific data on a mobile device.
- _comments: The social data Comments are stored here by default. Storing comments here instead of inside the documents themselves avoids having the documents marked as modified every time a user adds a comment. It also avoids replication conflicts on the documents when multiple comments are added in a short period of time. Also, because comments can be complex, even including attachments, storing them here avoids making the documents overly complex.
- _design: Not currently used, this is a special store intended to store design elements of the application.

Document

Then, there is the document. In Darwino, a document is four things:

- JSON data. It’s not necessarily a JSON object; it could be a JSON array. Most of the time it will be a JSON object; making it a JSON object is a best practice, because doing so is necessary to allow use of features such as tagging and reader/editor security. These are system fields added at the root of the document, thus implying that the root object is a document.

- Metadata: the UNID, creation date, last modification date, creator name, last modifier name. There are also several replication-related metadata items, but these are generally not of interest to the developer. These include the last replicated time and the sequence ID used in detecting replication conflicts.
- Potentially, a set of binary attachments. The attachments consist of binary data identified by name, and with an associated MIME type so that consumers can know what to do with the data. A single Darwino document can have multiple attachments, but the attachment name is the key so there cannot be two attachments with the same name in a single document. Attachments are generally stored in the relational database, in a dedicated table, and referenced from the document. This can be customized and the files can be stored elsewhere, such as the file system or a content management system. In addition to the attachment name and pointer, Darwino also stores a length and a size, plus some replication-related information.
- A set of social data. Darwino, by default, is social-enabled. You can rate a document, you can share a document, you can vote for a document, you can tag a document, and you can comment on a document. The social data is ABOUT the document, but it is not stored IN the document. The social data is stored in a separate table, and it is user-based: if five users rate a document, there will be five records in the social table, one per user, all referencing that document. The table is replicated along with the documents. There is one exception to this separation of the social data from the document's JSON data: tags. Most of the social data are action based (vote, rate, etc...) while tags are entered by the user, so they are part of the document fields. Tags are stored in both the document and the social table. In the document, the tags are stored in the field called "_tags", making them easy for an application to edit. When the document is saved, the values are copied to the social table to enable querying. Comments are stored as child documents since they can have several fields as well as their own attachments.

Document Hierarchy

Documents can be organized hierarchically. Every document can have a reference to a parent document. That reference is stored in the document's "_parentid" field. This field is accessible for read/write through the API.

The parent document must be in the same database as the child. There is a specific syntax for this field: when the parent is in the same store, then the value is just the parent document UNID. When the parent is in a different store, then the syntax is UNID:STOREID.

> Note: The system does not enforce the validity of the parent. This can lead to documents pointing to a non-existent parent. In this case, they are called "orphan" documents.

Synchronization master documents

Darwino implements “functional replication”. This means that selective replication can be based on changes to an ancestor document, as opposed to the current document.

The synchronization master for a document is the document that is checked for changes when the replicator is testing for selective replication eligibility. When a replication formula is applied on a document for selective replication, it actually applies to the sync master if one is defined. When the sync master document changes, and only when it changes, will the child documents replicate as well. This is how a sync master is used to logically group a set of documents together, so that they get replicated as a whole. For example, child documents might use the root parent document as their synchronization master.

Sync master documents are identified using the same convention as parent documents: when the sync master is in the same store, then the value is its UNID. When the sync master is in a different store, then it is UNID:STOREID.

There is an option for the save() method that forces the master document to update when a document referring to it as master is updated. There are options at the Store definition level as well.

It is not necessary for a synchronization master to be an ancestor; other document relationships could benefit from the ability to specifically define under what circumstances they will replicate as a group. For example: a customer and all the documents related to this customer, or all documents related to a particular project.

Social Data Updates

The Store's setUpdateWithUserData() method specifies whether the index should be updated when a document’s social data, which is stored outside of the document, is changed, even if the document itself has not been changed. An example of this would be if you are tracking ratings for documents and you wish to display the average rating for each document. Rather than recalculate the average every time you query the index, you would store the rating average every time the ratings are changed. By default, the index is not updated when the social data changes.

Document security

Darwino implements multi-level security. You can assign security to the Server object; you can control who can and cannot access the server. At the database level, you can assign an ACL. In the ACL, you can define who can access the database, manage the database, read documents, create documents, delete documents, and edit documents. At the Document level, you can maintain a list of users who can read or read/write the document, within the limits defined by the ACL. For example, if a user is granted only Read access to the database via the ACL, they will be limited to reading a given document even if the document-level security is set to allow Edit rights.

UNID

Documents have two identifiers: the UNID and the docID. The UNID is a string provided either by the API when creating the document or, if it’s empty, it is generated automatically by the system. A UNID must be unique per store; this is enforced by a database unique index. When there are two replicas, for example one on the server and one on a mobile device, the UNIDs will match.

> Note: The parentID is used to identity the document’s parent. It is the UNID of the parent. Sync Master documents are also identified by their UNIDs.

docID

The docID is an integer that is unique inside the database. It is generated by the system; you cannot specify it when you create the document, and it cannot be changed. Being an integer makes it much more efficient in database operations than a string such as the UNID.

However, the docID is not universal; it won’t be the same in two different replicas. Because of this, you should not store it for programmatic use and try to rely on its always applying to that document. docIDs are used internally because in the database each document maps to a set of relational tables; the relations between these tables are expressed most of the time through the docID because it’s much more efficient than the UNID. It is also easier and faster to programmatically manipulate large lists of integers.

Darwino DB API

Defining and deploying the database

A Darwino database is actually a set of tables within a relational database. The schema of these tables does not depend on the Darwino application. The schema is defined by Darwino, and always remains the same for each database; only the table names depend on the database name. This is important because in some organizations altering tables' schema is strictly controlled. Policy may require that a new DDL be submitted for the administrator to apply. With Darwino, no new table definitions are necessary… UNLESS you want to add additional indexes. See Optimizing the database for details on defining additional indexes.

Configuring the database using managed beans

There are two very important points in the Darwino philosophy here.

The first is that Darwino is built in layers, and you pick the the layer you want to use. It's better and more effective when you pick the highest-level one.

The second is that there is nothing that is hard-coded. Everything is provided by extension points. Extension points can rely on managed beans. For example, the connection to the database is defined through beans. But this is not the only way to define your connection. There is an extension point for defining your connection, and one default implementation of the extension point is looking for beans. Everything in Darwino is built upon extensions, and to find the extensions there is the notion of a plugin. A plugin is a class that can be provided by your application or by a library and that adds implementation for extensions. Extensions can be contextual to a platform or not. For example: the location where managed beans are found is provided by an extension point, because on a mobile device you won't find the beans at the same place as on a J2EE server. On the server, this is done via the J2eePlatform.java plugin.

See Services and Extensions for more information on plugins.

Darwino DB API

Documents and CRUD operations

It is possible to perform create, read, update, and delete operations on documents.

At the base, the Darwino API is a Java API. There is also a JavaScript API which is a JavaScript flavor of the Java API, and there are the REST services.

You can load and create documents either from the database or from the store. Remember that the UNID is unique per store and the DocID is unique per database.

```
// Create documents
Document d1 = store.newDocument(); d1.save();
Document d2 = store.newDocument(); d2.save();
Document d3 = store.newDocument(); d3.save();

// Read a document
Document doc = store.loadDocument("1000");
s += ">> Document\n";
s += "  Unid: "+doc.getUnid()+"\n";
s += "  Id: "+doc.getDocId()+"\n";
s += "  Json: "+doc.getJsonString(false)+"\n";
_formatText("{0}",s);

// Update a document
JsonObject json = (JsonObject)doc.getJson();
json.put("NewField","This is a new field");
json.put("field1","This is an update field - Previous was: "+json.getString("field
1"));
doc.save();

// Delete documents
// Call remove() from the Document object
d1.deleteDocument();
// Call delete() from the Store, using a UNID
store.deleteDocument(d2.getUnid());
// Call delete() from the Database, using a DocID
store.getDatabase().deleteDocumentById(d3.getDocId());
```

When loading an existing document, there are a number of options you can pass.

```
Document doc = store.loadDocument(unid, options); // options is an int.
```

Document loading options (these are ORed):

- DOCUMENT_NOREADMARK - This will load the document but not mark it as read. Normally, the read mark is checked when using the loadDocument() method, but not when the document is accessed via a query. Examples of when you might want to leave the read mark untouched include:
    - when it is a background process loading the document, and thus the user is not personally reading the document
    - when there is no solid reason to mark the document read, and you want to save the database write that marking the flag entails
- DOCUMENT_CREATE - By default, when you attempt to load a document that does not exist, you will get an exception. If you pass this flag, then if the document does not exist a document will be created in memory and no exception will be thrown. The new document will not appear in the database until you explicitly save it. This is particularly useful when using a remote connection, as it saves a server connection. It acts as LoadOrCreate().

There are also options available when saving a document:

- SAVE_NOREAD - When a document is being created (and ONLY then), will not mark the document as read. By default, a new document is marked as read by the user who created it.
- SAVE_NOTOUCH - It is possible to specify, at the store level, that parent or sync master documents, and, optionally, their indexes and all progenitor documents, should be marked as modified ("touched") when one of their dependent documents is modified. Saving with the SAVE_NOTOUCH option will prevent the touch actions from occuring.
- SAVE_CHECKCONFLICT - If enabled, the save will not occur and an exception will be raised if the document's last modification date doesn't match what it was when the document was loaded. This would be the case if the same document has been saved by a different process/user after your code opened it.

Delete document options:

- DELETE_ERASE - When replication is enabled for a database, deleting a document will, by default, create a deletion stub to represent a deleted document so that the document will be deleted in replicas of the database during replication. Using the DELETE_ERASE flag during deletion will delete the document without leaving a deletion stub. Thus, the deletion will not replicate out, and the document will return the next time the database replicates with a copy of the database in which the document exists.

- DELETE_NOTOUCH - Like SAVE_NOTOUCH, this will leave related documents un-notified of the deletion.
- DELETE_CHILDREN - The option will recursively delete all descendents of the document along with the document itself. Since these deletions are performed within a transaction, it's all-or-nothing. You will not be left with a partially-intact document family. Also, they are all done within the same network process, so there's no wasteful back-and-forth that would be required if the deletes were done one at a time.
- DELETE_SYNCSLAVES - This is like DELETE_CHILDREN, but applies when you are deleting a sync master document. It and all of its sync slaves will be deleted.

Transactions

To do CRUD operations, you get the session and, from that, the database and store, and there you create, read, update, and delete documents. As long as you’re using the Java API locally, you can execute a series of operations within a transaction. At the session level, you can query whether the session supports transactions. If it does, you can start a transaction, perform a set of operations on documents, and then roll back the changes if needed.

A transaction must be started (startTransaction()) and ended (endTransaction()). To get the transaction committed, one must call commitTransaction() in between start and end. Failing to do so, or explicitly calling abortTransaction(), will result in no changes being committted to the database.

```
// Simple transaction
session.startTransaction();
try {
    Document d1 = store.newDocument();
    id1 = d1.getUnid();
    d1.save();
      session.commitTransaction();
} finally {
    session.endTransaction();
}
```

Moreover, transactions can be stacked. In order to be committed, all the sub-transactions of the main transaction must be committed, else the entire top transaction will roll back.

```
session.startTransaction();
try {
    Document d1 = store.newDocument();
    id1 = d1.getUnid();
    d1.save();

    // Aborting a nested transaction will abort the whole transaction!
    session.startTransaction();
    try {
        Document d2 = store.newDocument();
        id2 = d2.getUnid();
        d2.save();
        session.abortTransaction();
    } finally {
        session.endTransaction();
    }

    session.commitTransaction();
} finally {
    session.endTransaction();
}
```

Access to the documents

There are three ways in Darwino to access documents:

- The Java API. This API talks directly to the database whenever it is a JDBC-based database, or is SQLite.
- REST services, which operate on top of the Java API. The set of REST services in Darwino supports everything the Java API allows in regard to document operations EXCEPT transactions. Because REST is stateless, transactions, which are stateful, cannot be supported.
- REST services wrapped for particular languages. Darwino provides two wrappers to assist in REST service work: a Java wrapper and a JavaScript wrapper. The Java wrapper is the Java API, but instead of being implemented to deal directly with the JDBC driver locally, it deals with the REST services. Nonetheless, it is the exact same API. The only difference is how you get access to the session. The JavaScript wrapper is intended for use in a JavaScript environment such as a browser or a server-side JavaScript environment like node.js. In the future there could be other

wrappers, just as PHP bindings, a Ruby binding, etc…

Access to the JSON content

The JSON document in Darwino is more than a JSON object; it has several components:

- JSON content – typically a JSON object, but it could also be a JSON array. It is unusual for it to be anything other than a JSON object, because only the JSON object supports use of system data that can be of use to Darwino.
- optionally, attachments
- metadata – For example, the UNID and the docID, modification date and modification user, as well as tags and other social data, and security-related fields such as READER and EDITOR fields which define which people and groups can access the document. These metadata are not modifiable manually by normal operations.
- system data that can be modified – For example, a list of tags. System fields are actually fields in the root of the JSON content object, but their names start with an underscore.
- optionally, transient property fields that can contain data we want to pass to document events but never want to store in the document. They are never saved. They can be set and read at the document level, but they are never persistent. A typical use of this is when you want to pass information to an event handler without having this information remain part of the document.

JSON content access methods

Methods are provided for accessing the JSON content in all data types. They all take a String as their only parameter, and return the value of the requested JSON field, assuming that the content is a JSON object:

- getString()
- getInt()
- getLong()
- getDouble()
- getBoolean()
- getDate()

These methods cannot access hierarchical data, but they are very convenient for accessing fields that are at the root of the document.

In addition, there is a method for executing JSONPath (XPath for JSON) expressions. JSONPath simplifies the extraction of data from JSON structures. It permits dot notation and bracket notation, and allows wildcard querying of member names and array indices. It is documented here. JSONPath expressions can be executed in Darwino via the jsonPath method:

- jsonPath(Object path)

Managing attachments

Every document can have a set of attachments. The methods for working with attachments are at the document level. Working with attachments is optimized; the attachments are loaded only when needed. When you create or update an attachment, nothing actually happens until you save the document. If the document save is part of a global transaction, then the work is postponed until the transaction is executed.

Document methods for Attachments:

- getAttachmentCount() returns the number of Attachments
- getAttachments() returns an array of Attachments
- getAttachment() returns the Attachment
- attachmentExists() – returns a boolean
- createAttachment(String name, Content content) – populates the content of the Attachment with the specified Content object, which can be Base64Content, ByteArrayContent, ByteBufferContent, EmptyContent, FileContent, InputStreamContent, or TextContent.
- deleteAllAttachments() – removes all Attachments

Attachment methods:

- getName()
- getLength()
- getMimeType() returns the MIME type of the content. If it wasn’t set when it was created, the system will base the returned value on the extension of the attachment’s filename. If there is no interpretable extension, it will assume binary.
- update(Content content) updates the content of the attachment
- getContent() returns the content of the attachment as a Content object.

- getInputStream() returns the content of the attachment as an InputStream
- There are three “readAs” methods intended for convenience, but are not meant to be used with large attachments. They are not as efficient as working with an InputStream: -- readAsBase64() -- readAsString() -- readAsString(String encoding)

The Content object has four methods:

- getMimeType()
- getLength()
- createInputStream()
- copyTo(OutputStream os) A Content object is an accessor to the data; it is not the data itself. This means that when creating attachments, the Content object does not actually load the data into memory until the document is being saved. Until then, it merely points to the data. This is an important performance consideration.

Darwino DB API

Cursors and queries

Cursors facilitate the selection of document sets from the database, and the extraction of data from the selected documents. You specify what you want to extract, and then you process the result. When you process the result, there is only the current entry in memory; it doesn’t load everything into memory and iterate through that set. The cursor lets you step through the results one by one.

> Note: A cursor is forward only. You cannot browse the resultset backwards.

A cursor consumes a database connection; thus, the connection has to be released when it's done. To avoid reliance on the intermittent garbage collector, Darwino provides a callback to the cursor. The cursor calls this cursor handler for every result. For example, when iterating through a result set, you call find(), passing a CursorHandler. The CursorHandler has one method, which is handle(), which handles the CursorEntry. The cursor is allocating the database connection, executing the SQL query, calling the CursorHandler for every result, and then closing and recycling the database connection.

There are methods designed for dealing with the subset of documents that are represented in the collection. For example:

- deleteAllDocuments(int options) – Will delete all of the documents not by iterating through and deleting each one-by-one, but instead will generate a SQL query to do the job in one fell swoop.
- markAllRead(boolean read) and markAllRead(boolean read, String username) will mark the cursors documents as read, either by the current user of by a particular username.

Query and extraction language

When searching for documents in a database, index, or cursor, there are many options available. For example, searches can be based on key, a range of keys, partialkey, parentid, unid, tags, and ftsearch. Since it is possible to force a unid's value in Darwino, using the unid as a key is a particularly efficient basis for searches.

> When we use the term "key", it means the unid when the cursor is querying a store, or the key when the cursor is querying an index.

- Search document by key

  ```
    public Cursor key(Object key) throws JsonException;
  ```

- Search document using a partial key, meaning document with a key starting with a specified string:

  ```
    public Cursor partialKey(Object partialKey) throws JsonException;
  ```

- Search a range of documents. When 'exclude' is not specified, then it means false (the document with the key is included). You typically specify a start and an end, although omitting the start means from the beginning, while omitting the end means up to and including the last document.

  ```
    public Cursor startKey(Object startKey) throws JsonException;
    public Cursor startKey(Object startKey, boolean excludeStart) throws JsonExc
  eption;
    public Cursor endKey(Object endKey) throws JsonException;
    public Cursor endKey(Object endKey, boolean excludeEnd) throws JsonException
  ;
  ```

- Select by parent UNID

  ```
    public Cursor parentUnid(String parentId) throws JsonException;
    public Cursor parent(Document parent) throws JsonException;
  ```

- Select by tag

  ```
    public Cursor tags(String... tags) throws JsonException;
  ```

- Select based on the doc id or the UNID

  ```
    public Cursor id(int id) throws JsonException;
    public Cursor unid(String unid) throws JsonException;
  ```

- Select using a full text search expression

  ```
    public Cursor ftSearch(String search) throws JsonException;
  ```

These methods are fast and efficient, but may not always provide the search term flexibility required by an application. Darwino's query and extraction language allows for queries based on complex criteria.

- Select using the query language

  ```
  public Cursor query(String query) throws JsonException;
  ```

The JSON query language has three variants; which you use depends on where and how you want to use it:

1. Query documents. In this case, it's a query condition. The result of a query that is applied to a document is true or false. Does the document match the condition or not? Does field "State" equal "New York"? Is the field "Price" greater than $100? The query condition is applied to every document in the database. The cursor will select only the documents for which the condition is true.
2. Data extraction. A document returned by a query is a piece of JSON. You may want to transform this JSON, for example if it contains a hundred fields and you're interested in only three of them. You don't want to download to a mobile client the entire document if most of it is not needed.
3. Calculating aggregation. When you have a cursor that is sorted by one or more keys, you can categorize based on the sorted values. The categorization groups the documents, and then you can calculate aggregate values for the groups.

When a cursor runs, it calls the cursor handler with all of the cursor entries. In the cursor entry are the key and the value, accessible via getKey() and getValue(). What these two represent depends on the source of the cursor. A cursor executed on a store will have documents as its result, the key will be the unids of the documents, and the value will be the JSON of the documents. If, instead, the cursor was executed on an index, then the key will be the key of the index, and the value will be either the value that’s stored in the index or the JSON value from the corresponding documents, depending on an option applied to the cursor.

For details on the query language, see Appendix 3. The Query Language.

Executing a query

A cursor is created at either the store or the database level. The store’s openCursor() method returns a cursor on which you then apply the selection condition. The Cursor methods return the cursor itself, which means that the methods can be stacked. For example:

```
Cursor c = store.openCursor().ftSearch(“version”).orderByFtRank().range(0,5)
```

This will perform a fulltext search on “version”, order the result by rank, and return the first five entries.

Cursor Options

When executing a cursor, there are options available that control the behavior of the query.

- DATA_DOCUMENT: Instead of returning the value of the column, the query will return the document itself.
- DATA_CATONLY: Only the category rows will be extracted from the query, and not the ones related to documents. If no categories are defined for the query, then an error will be raised.
- DATA_NOVALUE: The 'value' of each row entry will be null, instead of containing a JSON document. This is an optimization that prevents the actual extraction being done when not needed. This is useful if, for example, you're only interested by the document meta-data (unid, ...) or the hierarchy.
- DATA_MODDATES: Returns the creation and last modification dates of every document matched.
- DATA_READMARK: Returns a flag for every matching document that indicates whether it has read by the user executing the query.
- DATA_WRITEACC: Returns a flag for every matching document indicating whether the user executing the query is authorized to edit the document.
- HIERARCHY_SQL: By default, Darwino will utilize the underlying database's ability, if available, to perform recursive queries (CTE, or Common Table Expressions). This option will disable this support, forcing the runtime to perform the queries exhaustively, in the case where the built-in recursive query feature is deemed unreliable or inconsistent in the particular database. This is a potentially expensive option.
- QUERY_NOSQL: The option disables Darwino's default behavior of generating optimized SQL queries, forcing the runtime to do the query manually. Like HIERARCHY_SQL, this is potentially a very expensive choice and should be used only when absolutely necessary.
- RANGE_ROOT: When a query is using skip and limit, this option will cause the query to consider only the root elements when determining the skip and limit values.

By default, category and child entries are included when calculating the skip and the limit; this option overrides that default.

- TAGS_OR: By default, when querying by tag and specifying multiple tags, ALL of the tags must match for documents to be selected (the default is an "AND"); when TAGS_OR is specified, then ANY matching tag will allow documents to be selected.

Document Hierarchies in Cursors

The query engine natively understands Darwino document hierarchies, and the cursor takes advantage of that by allowing cursor queries to return not only the matching root documents but also their associated child documents. In other words, a cursor query can test for values in the root documents, and then return the matching root documents along with their children, regardless of whether the children match the query condition or not. The CursorEntry objects returned by the cursor will have an indentLevel property (an int) that identifies where they lie in the hierarchy, with 0 indicating a root document.

By default, when you query documents you get back a flat list of documents; there is no hierarchy. If you want to include the children of the selected documents, you have to explicitly specify that you want their children as well, up to a given level. By default, the hierarchical level is zero. If you specify hierarchical(1), then you will get the queried documents plus their direct descendents. Specifying hierarchical(2) would return all of those, plus the grandchildren.

The range() method

The range() method, given a number to skip and a number to return, will return a subset of a cursor’s entries, and can be controlled via the cursor options (by specifying RANGE_ROOT) to apply the skip and limit parameters only to the root documents, and to then return all associated child documents, without regard to the limit parameter.

Cursor Sorting Options

orderBy(String…fields) sorts cursor results by field. This can be an extracted field (for example, @myfield) or a system field (such as _unid or cuser). If the RDBMS supports JSONQuery, then a JSON reference can be used, such as a JSON field name or a direct path to a JSON field. Optionally, the sort order can be specified by appending a space and the text “asc” or “desc” to each field/path value. For example: .orderBy(“@state desc”, “unid”) will sort by state descending, then by unid ascending (that being the default).

The fulltext rank can also be used for specifying the order in a cursor, by use of the orderByFtRank() method. This is a short cut to .orderBy("_ftRank"). orderByFtRank() is always descending, with the best best match first.

The ascending() and descending() global options apply to the orderBy() method results, but are overriden by orderBy() when "asc" or "desc" is specified.

If no order is specified, and if there is no index, documents will appear in a cursor ordered by unid. When there IS an index, the default order is the index key.

Browsing the Entries:

There are two ways to execute a cursor:

- Call find(), passing a CursorHandler. It will execute the query, returning the entries one-by-one and calling the CursorHandler for each.

```
final StringBuilder b = new StringBuilder();
c = store.openCursor().range(0,10);
c.find(new CursorHandler() {
  public boolean handle(CursorEntry e) throws JsonException {
    b.append("  "+e.getString("firstName")+" "+e.getString("lastName")+"\n");
    return true;
  }
});
```

- Call findOne(), if you are sure there will be only one entry returned, or if you are interested only in the first. There is no need to pass the callback Cursorhandler.

```
c = store.openCursor();
CursorEntry e = c.findOne();
```

There are equivalent methods, different in that they extract the document object, allowing it to be updated and saved back to the database. Extracting a whole document has an extra cost compared to just loading the cursor entries (but it is still faster than the dual steps of getting the entry and then loading the document).

The count() method will return the number of entries in the cursor. Behind the scenes, Darwino executes a SELECT count(*) on the table, which can be costly on large data sets.

The countWithLimit() method behaves similarly, but uses the limit parameter. If the actual count is greater than the specified limit, then the limit is returned.

Categorization and Aggregation

Categorization is a means to group documents, and, secondarily, to calculate or aggregate on the groups. Categorization organizes documents in groups based on shared key values, determined by the orderBy() of a cursor, in the Darwino API. Categorization is completely dynamic; being based on the sort order, which is not fixed, categorization can be calculated on the fly.

There can be up to as many categories as there are sorted fields. If you have four levels in the sorting, then you can categorize on the first, or the first and second, or the first three, and so on. When categorizing, it isn't required that you start with the first sorted field; you can start with a subsequent sorted field instead. However, once you've started categorizing on multiple fields you cannot skip a sorted field and continue. That is to say, it is possible to categorize on the second, third, and fourth sorted fields, but not on just the second and fourth.

To calculate aggregate data, such as average, minimum, and maximum, pass an aggregate query to the cursor. For example:

```
Cursor c = store.openCursor()
    .query("{Count {$count: "@manufacturer”}, Sum: {$sum: “@released”},
    Avg: {$avg: “@released”}, Min: {$min: “@released”}, Max: {$max: “@released”}})
;
```

In this example, we count how many times the field "@manufacturer" is not null for all of the documents belonging to each category and we return the result as "Count". For "Sum", we calculate the sum of the field "@released" for every document in the category, and return the result as "Sum". We also calculate the average of the "@released" field as "Avg", and we select and return the minimum and maximum values for that field as well.

Categorization adds entries to the result to define the categories. If an entry has a "category" value of true, then it is not a document... it is a category entry. The "categoryCount" value, a number, contains the level of categorization for the entry in a multi-category result: top level would be categoryLevel of 1, the next level down would be categoryLevel of 2, and so on.

The .categories(int nCat) method takes as its parameter the number of category levels to apply, based on the orderBy() fields.

When categorizing, it is possible to request that the documents not be extracted in the cursor; in this case, the result will consist only of the category entries.

Another option is to extract categories while skipping the highest level categories. For example, extract two categories, but start at the second-level category, returning levels two and three.

Optimizing queries

The query language is robust, with a lot of operators, and is optimized for the underlying database system. It will attempt to use only native database functions when constructing its SQL; if necessary functions are not supported by the database, it will still use those that ARE supported for parts of the query. For example, if there is an “AND” in the query, and only one condition is directly supported by the database, the query language will execute that part of the query first and only then iterate through the results one-by-one.

The query language’s API allows the cursor to be checked to determine whether a particular query is supported by the database. It compiles the query and answers whether it can generate SQL for the query, or it can generate only partial SQL, or it cannot generate SQL at all.

Darwino DB API

Accessing and storing social data

There is a set of social data that can be associated with any document. There are three kinds of social data:

- Comments: Darwino creates a default store for the comment data. Keeping comments out of the documents themselves prevents unnecessary updates to the documents which then must replicate and could result in conflicts. Moreover, comments can contain full JSON content with attachments.
- Tags: The tags are stored as an array in the _tags field in the document. The tags can be queried, and the store can return a list of tags that can be used, for example, to create a tag cloud.

> Tip: Private tags, visible only to their creator, can be stored as read-protected response documents associated with the document being tagged. Because they are read-protected, they will be private to that user. Because the store's list of tags respects this security, only the private tag's creator will see such a private tag in a tag cloud.

- User-dependent values, which are handled at the store level: These don't require that the document be loaded for them to be applied, and they are stored externally to the documents in order to avoid unnecessary document modification, replication conflicts, and excessive data in the documents (there can be a lot of ratings and read flags):

  -- Ratings: With the rate() method, a document is assigned an integer value associated with a user. Appropriately, only one rating per document per user is stored. There are three methods for retrieving rating data:

  - getRate() – Given a unid and userName, returns that user's rating for the document
  - getRateAvg() – Given a unid, returns the average rating from all users for the document
  - getRateSum() - Given a unid, returns the sum of all ratings for the document, for example to count votes where ratings would be defined by the application to

range, say, from -1 to 1.

-- Sharing: With the share method, a unid and username sets a Boolean flag indicating whether the document has been shared by that user. It is similar in concept to a Facebook “like”. Two methods exist to interrogate shares:

- isShared() – Returns a Boolean indicating whether the document is shared by a specified username
- getShareCount() – returns the number of shares by all users as an integer

-- Read: While not specifically social, the read flag is functionally similar to the other social data. At the store level, the readMarkEnabled option enables the auto-flagging of documents as being read when they are loaded. This applies only to documents that are actually loaded; their being included in a query result is not sufficient to mark them as read. A document load() option can also prevent the flag from being set, when necessary.

There are three methods in the store to enable manipulation and querying of the read flags:

- isRead() returns the read value for the specified document and username
- markRead() - Given a unid, a Boolean, and a username, sets the read flag for that document and user.
- getReadCount() – returns the number of reads for the specified document.

> Note: Darwino security mechanisms may be used to prevent someone from updating somebody else's data.

There are two ways to access this social data. The methods at the store level require the unid and the username, in addition to any flags being set, to identify the document. The same methods are available in the Document object, where they do not require those two identifying parameters. Which set of methods you choose will depend on the context. If you have the document loaded, then use the Document methods, if just for simplicity. If you’re in a view, use the Store methods since they will be significantly more efficient (they will not require loading the documents).

The following fields are available in the query condition:

- shared
- shareCount
- rate
- rateAvg
- rateSum

- read
- readCount

These queries are generating extra database requests. For performance reasons, it will be better to store these values into extracted fields, or within an index. Then, the query will only be executed when the document is saved.

This social data can be used in extracted fields in the documents, making it easy to create indexes based on their values for querying and sorting.

Darwino DB API

Registering and handling events

At the Server Object level, you can register an ExtensionRegistry. This registry provides a set of functions – currently a set of five:

- BinaryStore: When Darwino stores attachments, they can be stored either inside the database or apart from the database. The BinaryStore is an interface that facilitates storing the attachments outside of the database by providing a set a CRUD methods. The attachments can be stored, for example, directly in the file system or in a CMS.
- DocumentEvents: When an operation is being performed on a document, the runtime will call the methods in the DocumentEvents:
  - postNewDocument – called right after a document has been created, so that you can, for example, change document values.
  - postLoadDocument – called right after a document has been loaded.
  - querySaveDocument– called immediately before a document has been saved. It is possible to cancel the save from within this event simply by throwing an exception. The exception can include the reason for the save cancelation.
  - postSaveDocument – called immediately after a document has been saved.
  - queryDeleteDocument – called before a document is deleted, EXCEPT when a group of documents is being deleted. Group deletes are performed directly by a SQL statement, and so, for performance reasons this event is not raised.
  - postDeleteDocument – called after a document delete, except, as with queryDocumentDelete, when a group of documents has been deleted. In such a case, you could add a trigger at the relational database level to, for example, log the deletion in a queue for processing.

  Note that these events are also raised when called via HTTP.

  Transient properties set at the document level can be accessed from within these events, despite the fact that these properties are never saved. This can be used, for example, by the calling code to pass information to the save event.

- SynchronizationEvents: As synchronization is taking place, this set of events will be raised, allowing customization of the synchronization actions. Like the DocumentEvents, code here can manipulate values or cancel the action altogether.

    - queryCreateDocument
    - postCreateDocument
    - queryUpdateDocument
    - postUpdateDocument
    - queryDeleteDocument
    - postDeleteDocument
    - conflictAction – raised when the runtime has detected a synchronization conflict, this event provides information about the conflict, including what changed in the source document and what is in the target document. Code here will return a ConflictAction, which will be one of the following: DEFAULT (the default handler should be applied), SOURCE (the source should win), TARGET (the target should win), or CUSTOM (call the handleConflict method, where the conflict can be handled by custom business logic. For example, in an HR application where several people interviewing an applicant each have access to a different section of the document. In this case, you would choose to merge the different sections.)
- FieldFunction: Functions used when extracting fields from documents or computing indexes are registered here.

- InstanceFactoryImpl: A database can have multiple instances, each with its own security configuration; in other words, there is per-instance security. That instance security is dynamic, based on business logic.

  When a database is opened for a particular instance, then an instance object is created in memory through an instance factory. Once the instance factory has been implemented and the instance is created based on the database and the instance name, the contribute() method of the instance is the mechanism for adding roles and groups to the user context.

  For example, the ACL of a database, and the reader/writer fields in the documents, may specify that only the members of a particular group may have access to the data. It is the job of the instance's contribute() method to add to the current user their list of roles and groups; this list is determined dynamically depending on business logic, and that logic is free to make use of any directories and database data available to it.

There is a default implementation of the ExtensionRegistry called DefaultExtensionRegistry. Use this to associate particular document event handlers with specific database stores. Once you create an instance of the DefaultExtensionRegistry, its registerDocumentEvents() method can be used to define specific cases of the document events. By specifying the database and store, you define which document events you want to override; for example here you would code your custom querySaveDocument event.

```
Public class AppDBBusinessLogic extends DefaultExtensionRegistry {
    registerDocumentEvents(“<My Database Id>”, “<My Store Id>”, new DocumentEvents
() {
        @Override
        public void querySaveDocument(Document doc) throws JsonException {
        }
    });
}
```

You can register events globally, and at the database level, and at the store level. If an event is registered at the store level, that is one that will be called for documents in that store. If, instead, there is no event registered at the store but there is one registered at the database, then the database registration will be in effect. The most-local (most precise) registration is the one that is used.

This is also the case for registered field functions.

Darwino DB API - Security

Database security

Darwino implements multi-level security. We have the notion of the user, which is mapped by the User class. A user has a CN, a DN, a list of groups, and a list of roles. The directory is used to authenticate the user and to get the list of groups. Generally, the groups come from the directory, while the roles are very specific to the application. Mapping between roles and groups can be done within a particular application.

At the server level, you can control who can and cannot access the server, based on user ID, group, or role.

At the application level, you can add dynamic roles to each user. The JSON store will trust the roles that are defined for the user in the User object.

With the Darwino Enterprise Edition, we have the notion of instances. An instance can contribute roles and groups to a particular user. For example, in a multi-tenant application running in IBM Connections with Communities, one tenant will be one Community. You can define the readers of a tenant as being all of those who are members of a Community, and the editors could be all of those who are owners of the Community. This will change for each Community, because the Communities each have their own lists of members and owners. In other words, you have an extension point that allows the Instance Manager to augment a user with roles and groups–dynamically–for a particular instance. The UserContext in the JSON store is the User plus what has been contributed by the Instance Manager.

At the database level, you can assign an ACL. In the ACL, you define who, among those who have been allowed server access, can access the database, manage the database, read documents, create documents, delete documents, edit documents, and update someone else's social data. When defining the database ACL, you are defining access rights to the database based on user IDs, groups, or roles.

Document security

At the Document level, you can maintain a list of users who can read or read/write the document. Document security is based on a simple set of rules involving fields specifying read-only and read/write access. Entries in these fields can be the names of users, roles, and groups.

There are four types of document security fields:

- reader
- writer
- excluded reader
- excluded writer

The same principles apply to both readers/writers and excluded-readers/excluded-writers.

Entries

Each entry can be:

- a person
- a group
- a role
- everybody, *

An entry can be read-only (reader field) or read/write (writer field). A writer entry is automatically a reader as well. If an entry appears in both the readers and writers, then it is a writer.

Security behavior

If there are no entries attached to a document, then there is no document security. The user's access to the documents will be determined solely by the higher levels (database and server). If there is at least one entry (reader or writer, or both), then there is document security. If everybody should be a reader and writers should be limited, the solution is the following:

- writers entries should contain the limited list
- reader should contain one entry: everybody *

Storing readers/writers

Readers are stored in the _readers field, while writers are in _writers. These fields can directly contain an array of entries (see "1 - Entries") or an object containing several arrays, one per property.

For example:

```
{
_readers: ['carol', 'ted'],
_writers: ['alice']
}
or
{
_readers: {
        field1: ['carol', 'ted'],
        field2: ['bob']
},
_writers: ['alice']
}
```

Having sub-objects is the preferred method, as it allows a finer-grained management of the entries. For example, a workflow engine can add a field containing the participants for the current step, and this can be removed after the step is completed.

Helpers and Options

There is a Java class, SecurityHelper, that can be used to manipulate these fields.

When document security is enabled, Darwino, when composing a SQL query, adds a subquery to exclude what is not allowed to be seen. This incurs a cost. To avoid this when possible, there is a database property indicating whether document security should be enabled. When it is not enabled, generated queries can avoid the step of running the subquery. A result of this is that if the flag is not set, readers and writers on documents will be ignored in all of the database’s stores. Options for this property are: no document security, reader/writer security only, ereader/ewriter security only, and all security features.

REST Services Restriction

Even though a particular user has access to a database, you can restrict their access via REST services. For example, in your application you may want to expose a business API; you may want to manipulate and return objects. Physically, they are stored as documents. You might want to expose your objects through REST services, while preventing direct access via the default REST services serving documents; you want people to have access only through your API. In this case, you could choose to prevent access to this particular database via REST services.

Dynamic Filtering

There is a DocumentContentFilter interface for the REST services that allows dynamic filtering of the document data that is being produced, typically for security purposes. Along with that is a feature of the API that allows reconciliation of filtered documents upon save, so that if a section was filtered for presentation, that filtered data is not lost from the document when saving.

Darwino DB API

Darwino API over HTTP

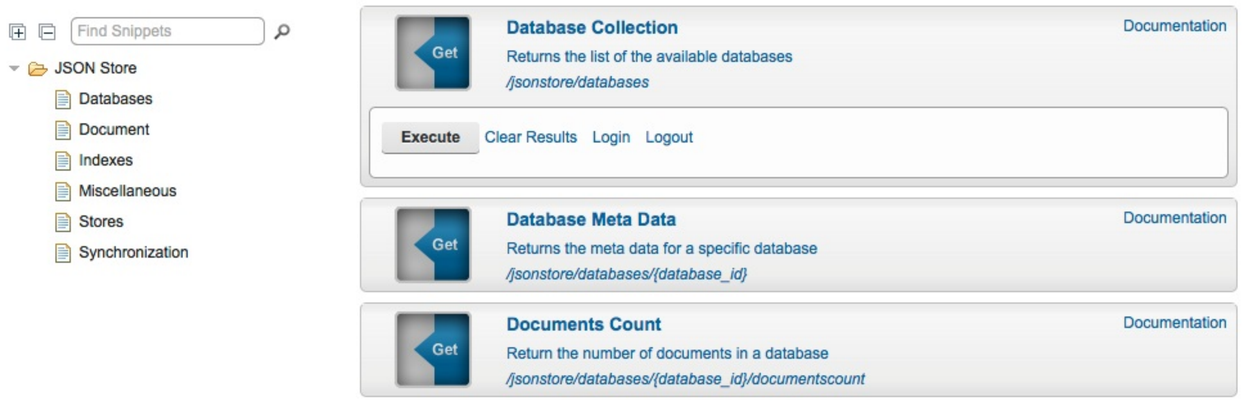
The entire JSON store API is exposed through REST services. Everything is supported except transactions, due to the stateless nature of HTTP. There are wrappers for Java and JavaScript, with more to follow.

You start with the session, and from the session you get access to all of the functions, and it's either going locally or it's going remotely through REST services. If you are going to be generating a lot of database accesses, you should do that on the server through custom REST services in order to minimize the number of remote calls. It is good to avoid using the Darwino API to perform a lot of remote database transactions. As a general rule, put the business logic server-side and call it through REST services.

REST API

All of the features of the JSON store are exposed through REST services, and the REST services are wrapped in the various language binders. The REST services are optimized for performance and designed to be easy to use. They execute in different environments, server-side and in mobile hybrid applications, and can be coded once and run in multiple platforms.

In the Darwino Playground is an API Explorer; this can be used to experiment with the REST Services' capabilities. All of the services are covered there, with documentation for all of their parameters.


Find Snippets
JSON Store
Databases
Document
Indexes
Miscellaneous
Stores
Synchronization

Get **Database Collection** Documentation
Returns the list of the available databases
/jsonstore/databases
Execute Clear Results Login Logout

Get **Database Meta Data** Documentation
Returns the meta data for a specific database
/jsonstore/databases/{database_id}

Get **Documents Count** Documentation
Return the number of documents in a database
/jsonstore/databases/{database_id}/documentscount

The Darwino framework allows the REST services, available by default, to be disabled at the database level, or to be overridden and enhanced. By overriding the service factory, is it possible to permit access dynamically based on the database, store, and current user.

The REST services can be extended to accommodate JavaScript components, jqGrid for example, that expect the JSON they're consuming to be in a particular format. One way to satisfy such a component would be to transform the JSON client-side, but that is not particularly convenient or efficient.

Using Darwino's JsonStoreServiceExtension, custom REST services can be defined and the existing REST services' output can be modified. Being server-side, the Java API can most efficiently render the JSON as needed before it is emitted.

JavaScript APIs

Darwino includes a JavaScript implementation for the JSON Store, the User API, and the Preference API.

Here, we will look at loading and making use of the capabilities of the JavaScript APIs.

JavaScript APIs

1 Loading the Javascript files

The JavaScript implementation’s source is compacted and compressed into one file: Darwino.js. This is the one file to include in applications; there is no reason to include the non-compressed files.

```
<script src=”$darwino-libs/Darwino/Darwino.js”></script>
```

Note: While it won’t break anything to load the JavaScript library more than once, it should be avoided because the browser will waste time loading and parsing it.

The JavaScript API can be explored in the "JavaScript Snippets" section of the Darwino Playground.

JavaScript APIs

2 Generic APIs

Darwino uses the namespace “Darwino”. Everything that belongs to Darwino is within the Darwino object. This cannot be changed.

Included in the darwino.js is Darwino.jstore, which is an entire JSON store API. This is the entry point when you want to use the JavaScript wrappers for the JSON data store.

You can directly call the JavaScript services, or you can use these wrappers, or you can use both. It’s a matter of convenience.

From the darwino.jstore REST API, you can call createRemoteApplication(), passing it a url for where the Darwino runtime is running (for example: “$darwino-jstore”). This returns a pointer to the remote server. From that, you can call createSession(). With no parameters, it creates a session for the anonymous user, or, if logged in, for the current authenticated user. Passed a username and password, it will create a session on behalf of the specified user. Every operation performed from that session will use the rights and identity of the session’s user.

Once you have the session object, you have the exact same API capabilities that you have in Java. There are, however, several details that are specific to JavaScript:

- The system constants (for example “SYSTEM_READERS” and “SYSTEM_WRITERS”) that are defined in Java are also defined in JavaScript. They are accessible through standard dot notation, as in: Darwino.jstore.Database.STORE_COMMENTS
- JSONPath is implemented in JavaScript as it is in Java.
- Another point specific for JavaScript is the way we handle binary content. Because JavaScript is restricted to manipulating the binary data as Base64, it is more efficient to do such work on the server in Java via REST services and just display the value, or values, or links inside the HTML. JavaScript is not designed for this.
- Synchronous vs. Asynchronous calls To create an application that is responsive and not often blocking the user you have to use asynchronous JavaScript, which means that when you call a service you’re not blocking the UI thread. The entire JavaScript

Darwino API allows you to do asynchronous calls. You may choose to do either synchronous or asynchronous calls, but synchronous calls should be used only when the application demands them. Asynchronous is the default; if you want to do synchronous calls, you have to pass parameters, either at the session to change the default (session.Async(false)) or with each individual call.

By default, when you call a JavaScript function that triggers a call to a service, what it returns is a promise. The latest generation of browsers supports promises, but because not all browsers do Darwino provides an A+ Compliant version that is backwards-compatible with older browsers.

A promise is a call that will eventually be executed. For example, session.getDatabase() will return a promise. The promise itself has a “then” method which takes as its parameters a function to execute upon successful completion of the promise, and a function to execute in the case of failure.

Promises can be chained.

Some functions, such as getDatabase(), will return a promise, while others, such as getStore(), will return a real value. There is no way to differentiate between the two types other than the fact that if a function has a parameter called “header” then it is an asynchronous function and will return a promise.

The Darwino Playground is a resource for examples of synchronous and asynchronous calls and promise handling.

```
 var s = "";

session.getDatabase("playground",null,function(database) {
  var store = database.getStore("pinball");

  // The document is loaded asynchronously
  store.loadDocument("1000", null, function(doc) {
      s += ">> Document\n"
      s += "  Unid: "+doc.getUnid()+"\n"
      s += "  Id: "+doc.getDocId()+"\n"
      s += "  Json: "+doc.getJsonString()+"\n"
      darwino.Utils.setText("content","{0}",s);
  });

  // This document does not exist
  // So the function is only called in case of success
  // Nothing happens in case of an error
  store.loadDocument("1000FAKE", null, function(doc) {
      s += "!!! Should never be displayed as the document does not exist\n"
      darwino.Utils.setText("content","{0}",s);
  });

});

s += "Loading document...\n"

darwino.Utils.setText("content","{0}",s);
```

Developing a Darwino J2EE Web Application

To support a Darwino J2EE application, you need to have an application server such Tomcat or WebSphere that supports the Servlet API.

Darwino provides a broad set of features and support services to a web application. It is possible to create an application that makes use of Darwino DB without using the other services that Darwino provides, such as creating connections and handling replication, but by using the pre-built services in the full Darwino package, the programmer will avoid a lot of unnecessary work.

1 Application initialization

The Darwino application object should be initialized before anything else. In order to create the application, the context listener must be included in the web.xml:

```
<!—Application initialization -->
<listener>
    <listener-class>
        demoApp.app.AppContextListener
    </listener-class>
</listener>
```

The listener is called when the application is started, and again when it is stopped. The listener will create the application object, and destroy it when it is no longer needed. It can also be used to initialize the relational database by creating the tables, assuming that the RDBMS user has the rights to modify the database schema.

```
<context-param>
     <param-name>dwo-auto-deploy-jsonstore</param-name>
     <param-value>true</param-value>
</context-param>
```

The possible values for dwo-auto-deploy-jsonstore are:

- true: If the database is not yet deployed, deploy it. If it is already deployed and not at the latest version, upgrade it.
- false: Do nothing.
- force: Erase what exists and redeploy from scratch. This is mostly for developers who are still making a lot of changes and want to start fresh every time.

If you don't have the authority to update the database schema, there is an option you can set in the JDBC definition level (in the JDBC connector, darwino/jdbc, which defines your connection to the database) to have Darwino not try to create the tables, but rather assume that they exist and just save the JSON definition of the database - the characteristics that are not related to the DDL itself, such as the extracted fields.

In practice, the developer will create their own class, extending AbstractDarwinoContextListener, where they will handle their application’s initialization needs, and refer to that class in the web.xml so that it is called.

The DarwinoServiceDispatcher

By default, a set of services is created by Darwino, such as the service to access the JSON store. This is done by the service dispatcher. It is possible to override the dispatcher. It contains a set of methods that can be disabled or added to.

```
Protected void initServicesFactories(HTTPServiceFactories factories) {
  addResourceServiceFactories(factories);
  addLibsServiceFactories(factories);
  addJsonStoreServiceFactories(factories);
  addSocialServiceFactories(factories);
  addApplicationServiceFactories(factories);
}
```

It is also possible to register custom services by using an extension point.

Darwino Application filter

The Darwino context should be created when a request comes in, and it should be deleted when the request is satisfied. In order to provide the context for an application request, a J2EE filter specified in the web.xml is called. This filter must be specified immediately after authentication (if Darwino is handling authentication) in the web.xml file so that it is executed before any others. Only in this way can subsequent filters have access to the context created here.

```
<!-- Filter for creating the Darwino Application, Context and DB session -->
<!—The filter must be first so all the other filters can access the app -->
<filter>
    <filter-name>DarwinoApplication</filter-name>
    <filter-class>com.darwino.j2ee.application.DarwinoJ2EEFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>DarwinoApplication</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

The filter is executed before all else when a request comes, and last when a request is handled. Because this is first in line, it can do processing before the request is even seen by the servlet, and then again after the request is processed.

The filter is typically executed for all requests, but the filter mapping allows the filter to be executed conditionally.

Darwino libs and URL rewriting

The DarwinoRewriting filter transforms some urls such as “$darwino-libs” into an actual path. This remapping enables platform independence without requiring code changes to accommodate different platform configurations.

This filter can also perform HTML rewriting. When the HTML is served to the client, it can be transformed. This is useful in CDN scenarios, where the urls being sent to the client may need to be modified to point to alternate locations.

```
<filter-name>DarwinoRewriting</filter-name>
<filter-class>com.darwino.j2ee.servlet.resources.DarwinoGlobalRewriterFilter</filt
er-class>
</filter>
<filter-mapping>
    <filter-name>DarwinoRewriting</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Serving application resources

In order to determine which requests should be handled by Darwino services, this filter analyzes the incoming requests, looking to see if there are internal services to handle them, and delegates them to Darwino if appropriate and passes them along if they are not.

This service is for serving 'static' resources, like HTML, CSS, JavaScript, etc... It ensures that it works on all the platforms, including J2EE and mobile. It serves the resources located in the platform specific directories (ex: web app for J2EE, assets for Android...) but also the ones packaged in the jar files under /DARWINO-INF/resources. (META-INF/resources cannot be used on Android).

Also, depending on the execution mode (development vs. production, as defined at the Platform object level), it can choose to load the minified version of the files or the full commented one. The minified versions have a ".min" inserted to their path, like myfile.min.js or mytheme.min.css.

```
<filter-name>DarwinoServices</filter-name>
<filter-class>DWOTPL_PAGEAGENAME.app.DarwinoServiceDispatcher</filter-class>
</filter>
<filter-mapping>
    <filter-name>DarwinoServices</filter-name>
    <url-pattern>/*</url-pattern>
    <!—DarwinoRewriting can trigger a forward -->
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

Enabling GZIP compression

Not all web servers implement GZIP. You can use this filter to have Darwino process GZIP requests and return GZIP content. This can work for requests as well as response content.

```
<filter-name>GZipFilter</filter-name>
<filter-class>com.darwino.j2ee.servlet.gzip.GZipServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>GZipFilter </filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

Enabling CORS

To have Darwino implement the CORS (Cross-Origin Resource Sharing) standard, enable this filter. This provides support for cross-site access controls.

```
<filter>
    <filter-name>Cors</filter-name>
    <filter-class>com.darwino.j2ee.servlet.cors.CORSFilter</filter-class>
    <init-param>
      <param-name>cors.allowed.methods</param-name>
      <param-value>GET,POST,PUT,DELETE,HEAD,OPTIONS</param-value>
    </init-param>
    </filter>
    <filter-mapping>
    <filter-name>Cors</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
```

This CORS filter is based on eBay's implementation documented here.

Authentication and Authorization

A Darwino web app is not just serving up the UI; it is also serving data through the REST services, so you want to be sure to properly secure the application.

There are several ways to secure the application. One way is to utilize the web application server's security mechanisms. The web.xml file contains the J2EE specification, and defines how security is handled by the J2EE container.

```
<!-- Enable this to use the J2EE CONTAINER security -->
    <security-role>
        <role-name>user</role-name>
    </security-role>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>ApplicationRoot</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>demoapp</realm-name>
    </login-config>
```

Above, we see a role named "user" (at the web application server, we would map that name to "users"). That role is then granted access to resources via the url-pattern (in this case, all resources). BASIC authentication is then specified, which will prompt for a username and password.

As you can see, this is standard J2EE security. It is often enough to do the job, but in some cases it is not sufficient. The web containers are implementing only a small set of authentication methods, and you cannot mix and match them. For example, you cannot combine the very secure form-based authentication for your web application with basic authentication for your REST services, unless you're willing to rely on extensions for your web container.

When you want to use the J2EE container's built-in capability, the J2EE container must be able to directly access the directory in order to authenticate the user. Thus, the directory that you're using must be available to the web application server. Some web application servers have proprietary APIs that give them access to non-standard directories, but this is not a standard – it depends on the particular application server. This is a problem if you want an application that is portable.

Darwino provides a set of J2EE filters to support form-based authentication and basic authentication. Using these filters, you can protect different parts of your application using different mechanisms. These filters are a subset of the features implemented by Apache Shiro. These filters are fully portable across web application servers.

A side effect of this filter authentication is that the users are unknown to the web application server; to the server they are always anonymous. Darwino has the responsibility for authentication and access control.

Security can be done at the web application server level, for example with J2EE’s Container security or WebSphere’s Administration Console. Alternatively, it can be handled via Darwino’s authentication filter.

Darwino’s authentication filter provides basic authentication and form-based authentication, and it can work with a directory implemented as a Darwino database.

Accessing the current user in Darwino code is done through the DarwinoContext.

Developing a Darwino Mobile Application

General information about mobile applications

Creating a mobile application in Darwino is like creating a J2EE application, except that instead of building a project that generates a WAR, you build a project that generates whatever the mobile device is expecting: an APK in the case of Android, or an IPA for iOS.

Because Darwino uses the Android and Multi-OS Engine SDKs, the projects that the Darwino wizard generates for those platforms must include what those SDKs are expecting.

When creating a mobile app, the wizard offers a choice of either native app or hybrid app.

Mobile Manifest

The Mobile Manifest contains the information consumed specifically by the mobile applications.

In the manifest you can define connections to different servers. By default, your application will be connected to one server at a time, but an application might allow the user to choose servers from a list. The manifest is where this list of servers can be defined. To this end, the DarwinoMobileManifest object includes a method called getPredefinedConnections() which, by default, uses a JSON file as its connection source list.

Some of the other mobile-specific options defined here include:

- isLocalDatabase() - whether the application is using a local database
- isWebMode() - does it have web mode enabled for hybrid apps
- isDataSynchronization() - does it synchronize with the server
- isEncryptedByDefault() - should the data be encrypted

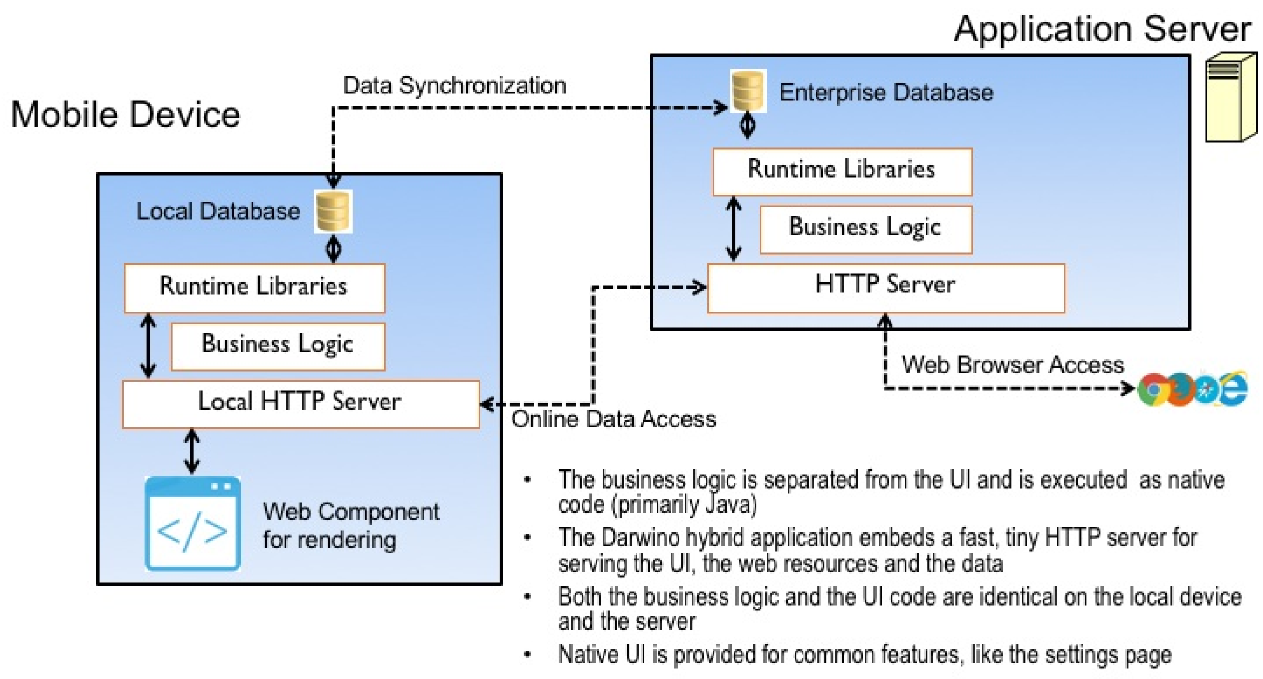
Hybrid applications

The idea is to have true portability regardless of the device where the application is being executed, and to work seamlessly offline and online. As the diagram below diagram depicts, the architecture of Darwino on the mobile device and on the application server is identical. They are both running an HTTP server; on the server, it is WebSphere or Tomcat or the equivalent, while on the mobile device it is a lightweight server based on NanoHTTPD. Both sides can run the business logic, written in Java and relying on Darwino's runtime libraries.

The hybrid app running on the mobile device instantiates a web component, but the web component is talking to the local HTTP server. The means that the application is truly portable between the web compoent in the hybrid app and the web browser. There are no differences. This allows Darwino to have a single app run in different platforms without any changes.

Note that the database access is abstracted by the runtime libraries. If you're offline, it will access the local database on the mobile device, and if you're online it will access the remote database located on the application server. This is completely transparent for your application.

This is better than a straight Apache Cordova application, although Cordova can be used if desired.



- The business logic is separated from the UI and is executed as native code (primarily Java)
- The Darwino hybrid application embeds a fast, tiny HTTP server for serving the UI, the web resources and the data
- Both the business logic and the UI code are identical on the local device and the server
- Native UI is provided for common features, like the settings page

Writing a Hybrid specific service

A hybrid app can be extended by providing services. The services will be available whenever the app is running on the server or locally. It is also possible to create services that are very specific to a hybrid app. For example, accessing the camera is something that makes sense only on a mobile device. To enable this sort of feature, it is possible to create actions that can be triggered from the web app and execute in the native code of the app.

The JavaScript API contains an object called darwino.hybrid that enables this action mechanism. Its isHybrid() function returns a Boolean indicating whether or not the code is running in a hybrid app. isHybridAndroid() and isHybridIos() do a similar but more specific evaluation. These functions are always available, without regard to whether the code is running in a mobile or a web app.

exec()

The exec() function is the equivalent of the shell() function found in a variety of other languages; it allows the calling of external functions. In this case, “external” means device-native activities. Exec() provides the bridge between the HTML side of the app and the native code.

exec() has four arguments: a verb, a set of arguments to be passed to the verb, a callback, and a Boolean specifying whether it should run asynchronously or not.

It is possible to create custom actions, and there is a set of predefined actions, including:

- switchToLocal
- switchToRemote
- switchToWeb
- synchronizeData
- startApplication
- openSettings

Registering actions is done in AndroidHybridActions, which itself is registered as an extension in AndroidPlugin, via the registerCommands() method. The equivalent methods exist for iOS, and other OS-specific implementations can be provided.

This makes it possible to register a command with a name. For example, to create a command that takes a picture and attaches it to a document, implement the execute method in the AppCommand class using the context to pass the necessary parameters, such as the docID. Because multiple processes may execute commands simultaneously, instance variables shoudn’t be used for storing data; commands should be called with their own local context.

RPC callbacks

In addition to exec(), it is possible to implement an RPC callback. RPC functions execute synchronously, and they can return a value. Unlike the commands, these functions are only available to hybrid apps.

JavaScript functions

As with registering commands, it is possible to register JavaScript functions. This is done via the registerFunctions() method in the JavaScriptFunctionExtension class. These functions are the ones called by the RPC mechanism described above.

Settings

exec() is a way for the web side of the app to communicate with native device code. The converse of that is done via registered listeners. For example, the settings listener will notify the app when something has changed in the device settings. The app could then use isDirty() to see if the settings have changed since the last refresh (after replication occurs, for example), and then read and set settings as required. Several functions are provided to assist in this:

- addSettingsListener()
- setSettings()
- getProperty()
- getMode()
- isDirty()
- setDirty()

Developing for Android

When creating a hybrid app for Android, the wizard generates the classes required by the Android SDK, including AndroidApplication.java, AndroidHybridActions.java, and SplashScreenActivity.java. It also generates the same classes used by the J2EE applications. When we are in mobile or web mode, many of the same principles apply, but they have different implementations.

On mobile the DarwinoHttpServer class is overridden, resulting in the DarwinoServiceDispatcher. In this class, you can define all of the services that you want to make available through the local HTTP Server. By default, its initServicesFactories() method calls a set of initialization methods:

- addResourcesServiceFactories() - provides all of the static resources: HTML, JavaScript, CSS, etc...
- addLibsServiceFactories() - serves the JavaScript and CSS libraries
- addJsonStoreServiceFactories() - provides the JSON Store REST services
- addSocialServiceFactories() - provides the User service (and more to come)
- addHybridServiceFactories() - provides the hybrid-specific services, such as commands
- addApplicationServiceFactories() - allows the creation of custom application services
- addLibrariesServiceFactories() - let you load libraries that register services through extension points. It will find an extension point for HTTPServiceFactory and it will add the result of that extension point. This lets you drop the library into the project and have its services automatically registered.

You can override any of these methods to stop it from registering its services. For example, if you won't be using the social services, you can override addSocialServiceFactories() and stop the services from being loaded.

The wizard’s output for a native app is smaller; there is no HTTP server included. The wizard will generate the DarwinoApplication class and the MainActivity, but it will leave creating the UI to the developer.

Developing for iOS --- Multi-OS Engine

This wizard generates an iOS project that utilizes the Multi-OS Engine SDK. As with Android apps, the generated project contains the core classes required for a basic application, including DarwinoServiceDispatcher and MainViewController.

Business APIs

General information

The business APIs are an integral part of the core Darwino environment. These APIs encapsulate a set of services, providing an easy-to-use, platform-independent interface to assist in coding common business application functions.

There are currently three business APIs: The User Service, the Mail Service, and the Preferences Service. This set of APIs is architected to grow over time. Eventually, it will cover the whole gamut of social services: file sharing, communities, etc...

User Service Overview

This user service has two functions: authentication and providing information about users. This service is used throughout the Darwino platform. For example, when creating a session, Darwino will utilize the User service to determine the roles and groups needed to assign the proper security.

User Information

User information is generally not managed by Darwino; it is stored somewhere else, such as in an LDAP directory, in WebSphere VMM, or, in the case of Domino, in the NAB. The User service provides access to the external, central directory. There may also be peripheral information about users. For example, the primary user directory may be in LDAP, while other information, such as the user's photograph, is stored in IBM Connections or Facebook. The User Service is architected to simplify working with such distributed user information. One directory is considered the main directory, and additional data can come from zero or more secondary directories.

The directories that work with the User Service are normally implemented as managed beans, and they are fully-extensible. Darwino provides beans for several LDAP directories, Oracle Directory Server, IBM Tivoli Directory Server, Microsoft Active Directory, the native Domino directory, and a static directory for development purposes.

To work with the User Service:

```
     return Platform.getService(UserService.class);
```

The User Service provides a set of function for finding users and retrieving details about users. Because multiple directories may be referenced, there could be multiple IDs for a single user. Nonetheless, there must be only one canonical distinguished name (DN). With this in mind, there are two operations available to find a particular user.

This method returns the User object corresponding to the provided DN:

```
public User findUser(String dn) throws UserException;
```

This method finds the user that best matches the provided ID. Depending on the directory configuration, the ID can be a DN, an email address, a short name, a common name, etc…

```
public User findUserByLoginID(String id) throws UserException;
```

findUserByLoginID() does not identify a user with certainty; only findUser() can do that.

There are also several functions for returning lists of users:

- findUsers() returns a list of users based on the provided String array of DNs. This is an optimization, reducing the cost of finding multiple users. This is one call for multiple users as opposed to multiple calls, each for a single user.
- query() takes an LDAP query (allowing ANDs and ORs) and returns a List of all matches across all directories.
- typeAhead() performs a simple “starts with” or "contains" query, depending on the implementation provided by the driver, and returns a List of all matches across all directories.

Once you have the User object, you can use its methods to query a provider for user details stored there. For example:

- getDN()
- getCN()
- getGroupCount()
- getGroups()
- getRoleCount()
- getRoles()
- getAttribute()

> Groups come from the directory itself, while roles are generally application-specific.

User Authentication

To use the UserService for user authentication, simply get the UserAuthenticator object using getAuthenticator() with the provider name as the parameter. The UserAuthenticator that it returns provides the authenticate() method for performing the authentication with the username and password. Authenticate() will return the user’s DN if authentication was successful.

When the web application authentication mechanism is being used, the J2EE server returns the user’s principal (a DN). This principal is used by the UserService to create the User object. This User object represents the user throughout Darwino; there is no other user identity.

On the mobile device, the implementation is different due to the lack of the LDAP API. Instead, authentication is provided by connecting to the server and storing the current user DN. This is what is done in the "Settings" page.

Darwino has the ability to cache the user information on the mobile device. Furthermore, using custom code a developer can provide a list of users whose information should be prepopulated in the cache via a background operation.

User Service Providers

Darwino has a user directory which functions as the main user directory (Darwino includes a set of main directory implementations via LDAP: IBM Domino, Tivoli Directory Manager, Oracle Directory, and Microsoft Active Directory, plus a native IBM Domino directory implementation). Every user in this directory is identified by a DN. This DN is fixed; it should not change. From this directory you can extract user data, for example the common name of the organization. On top of this directory, you can have user data providers. They provide data on existing users but from a different data source, such as an external LDAP directory, IBM Connections, or Facebook.

Suppose that your main directory is an LDAP directory, but you want to have the users' pictures coming from IBM Connections. You can write a user data provider that will use information you have in your main directory to locate the user in the IBM Connections directory to extract the data you need.

You can have as many user data providers as you want. A Gravatar implementation that can be used as a secondary directory is included with both Darwino Editions. The Darwino Enterprise Edition also includes a provider for IBM Connections.

Directories are made available to the UserService by registering them by name as providers. Registered providers are included when searching via the query() and typeAhead() functions.

It is also possible to get user information from a specific provider. The User object’s getUserData method, given the name of a registered provider, will return a UserData object. The UserData object has a getAttribute() method for retrieving value of the specified attribute, and a getAttributes() method to return all of the User’s attributes.

For retrieving binary user data, there is the getContent() method which returns the binary data of the specified type, such as “photo” or “payload”. Whereas attributes are cached, content data is not; it is always retrieved from the provider when it is requested.

Because it is possible to have multiple providers registered, user data may be spread across multiple directories, and there could even be duplications. For example, the user’s photo could exist in multiple providers’ sources. To accommodate this, the findAttribute() and findContent() methods will search exhaustively across all registered providers, starting with the current user object and stopping when they find the specified data. The developer doesn't have to be concerned about where the data is actually stored.

Another issue that can result from having multiple registered providers is the need to map a user's identity across providers. The UserProvider object includes the UserIdentityMapper() method, with converts between a user's DN and the provider's username format.

Mail Service

The Mail Service is a basic interface for sending emails. There is currently no REST service support for the Mail Service, and it is not supported on mobile.

To send a simple email, create a MailMessage object and set the mail parts via the MailMessage methods, then call send().

```
MailService mailService = Platform.getService(MailService.class);

MailMessage m = new MailMessage();
m.setFrom("playground@darwino.com");
m.setTo("darwinounit2@gmail.com");
m.setSubject("Simple email");
m.setContentText("This email is a simple one");
mailService.send(m);
```

HTML body content can be created via the setContentHTML() method:

```
m.setContentHTML(“Here is <b>bold</b> and <i>italic</i>.”);
```

To send more complicated messages, the MailMimePart class allows the creation of MIME content from text:

```
MailMimePart ht = new MailMimePart();
ht.setContent(new TextContent("Alternate <b>HTML</b> email representation",TextCon
tent.UTF_ENCODING,HttpBase.MIME_HTML));
```

Attachments are also supported:

```
MailMimePart at = new MailMimePart();
at.setContent(new TextContent("This one is <b>HTML</b>",TextContent.UTF_ENCODING,H
ttpBase.MIME_HTML));
at.setName("Attachment.html");
m.addMimePart(at);
```

To send images, pass the image string as BASE64 to setContent() and supply a filename for the attachment:

```
MailMimePart at1 = new MailMimePart();
String IMAGE = "R0lGODdhAAGAAKIAAP38+/3h3cjN5P3HwgAAAP8AoP8AGv8EIywAAAAAAAGAAAAD..
...";
at1.setContent(new Base64Content(IMAGE,HttpBase.MIME_IMAGE_PNG));
at1.setName("Attachment.png");
m.addMimePart(at1);
```

Preferences Service

The Preferences Service is an interface for setting, reading, and removing user and application preferences.

To read a preference, create a PreferencesService object, then call getPreferences(), passing the username and preference name as arguments. Then call get(), with the preference name as the argument.

```
PreferencesService prefService = Platform.getService(PreferencesService.class);

Preferences p = prefService.getPreferences("user1","pref1");

// This default service has pref1 & pref2 defined, but not pref3
_formatText("  pref1: {0}",p.get("pref1"));
_formatText("  pref2: {0}",p.get("pref2"));
_formatText("  pref3: {0}",p.get("pref3"));
```

To set a preference value, first get the preference and then call set(), passing the preference name and value as arguments:

```
PreferencesService prefService = Platform.getService(PreferencesService.class);

// Preference is set locally in the object
Preferences p = prefService.getPreferences("user1","myprefs");
p.set("custom","value 1");
_formatText("  custom: {0}",p.get("custom"));

// Does not appear because it is not yet saved
Preferences p2 = prefService.getPreferences("user1","myprefs");
_formatText("  custom: {0}",p2.get("custom"));

// Save it, and it is now in the DB
p.save();
Preferences p3 = prefService.getPreferences("user1","myprefs");
_formatText("  custom: {0}",p3.get("custom"));
```

To delete a preference, call deletePreference(), passing the username and preference name as arguments:

```
PreferencesService prefService = Platform.getService(PreferencesService.class);

prefService.deletePreferences("user1","myprefs");
```

Optimizing the database

It is not often necessary to consider the underlying RDBMS when creating an application in Darwino; the services and API abstract the details away so you can focus on the business logic. Still, when performance is a serious concern you may find a benefit in considering how Darwino works with the tables and their indexes.

When you execute a query in Darwino (see Appendix 3. The Query Language for details on queries), the system will generate and execute a SQL statement to satisfy the request. Darwino will do what it can to make that SQL query as efficient as possible, utilizing the indexes that are available. However, if the necessary indexes aren't there, you inadvertently could force a table scan. With small or infrequently-accessed tables this may not be an issue, but there are many cases where it's going to have a noticable effect.

To be sure, your Darwino application will work without any indexes beyond those that Darwino provides by default. Additional indexes come into play only if needed to make the application faster once it is in production.

Default indexes

There is one set of relational tables per Darwino database. The names of the tables depend on the database name, and the definition of the tables is static for a given version of Darwino. This allows the tables to be created once by a DBA, and then used as-is even as the application evolves. See Appendix 2. Mapping between a Darwino DB and a relational database for details on the Darwino application tables.

Darwino defines a minimal set of RDBMS indexes to assist generic access to the database (get a document by id, get document by UNID, index by key, synchronization...). Because there is a cost involved in creating indexes, Darwino makes no presumptions about your application's needs beyond these basic functions. It is up to the application developer to track the requests being emitted to the database and then add additional indexes if necessary.

> Note: On the database systems that support native JSON access, JSON access indexes can be added. Refer to your RDBMS documentation for details and best practices.

The database customizer

You can use your RDBMS's administration tools to create indexes while you're developing and debugging your application. A problem arises once you have deployed the application: you, as developer, may no longer be in control of the database, and you cannot be sure that your indexes will remain.

Fortunately, Darwino has the ability to generate the indexes for you. The JdbcDatabaseCustomizer has a method, getAlterStatements(), that is called when you deploy the database. It takes a set of SQL statements and executes them for you. Those statements can create indexes, stored procedures, triggers... anything that is understandable by your database. Because the database schema is being modified, it is necessary that you have authority to perform database DDL write operations.

> Note: While you can code logic in Darwino to act when a document is deleted individually, there is no event raised in Darwino when a document is being deleted as part of a group operation. If your needs demand it, you could code a database trigger to do processing when database records are deleted. For example, a trigger could act on a delete, copying the record to a separate table for archiving. Such a trigger could be defined in the database customizer.

The database customizer includes a version number. When the database is loaded by Darwino, that version number is compared against the last-used customization version. If the customizer returns a version that is higher, indicating that there are changes needing to be applied, then Darwino will call the customizer so it can do its job.

Appendices

Utility Libraries

Darwino comes with a set of general utility classes. These classes are located in a variety of projects. A few of the most noteworthy are described here.

StringUtil

In dwo-commons is StringUtil. It consists of routines to simplify handling Strings.

For example, throughout Darwino a null string and an empty value are considered as equivalent. This is similar to how JavaScript handles the two. For convenience and simplicity, Darwino brings this approach to Java via the isEmpty() and isNotEmpty() functions in StringUtil.

AbstractException and AbstractRuntimeException

All of the Java exceptions that are thrown by Darwino inherit directly or indirectly from these classes. Darwino’s exception classes provide additional features on top of the classes provided by Java, in particular for debugging. They implement and enforce an exception-chaining pattern; every time an exception is caught in Darwino and another exception is thrown, the original exception is passed as a parameter to the new exception.

```
public AbstractException(Throwable nextException) {
    this(nextException, nextException==null?””:nextException.getMessage() );
}
```

To enforce this behavior, all of the constructors of Darwino exceptions must have this parameter. It may be null, but it must be present; a conscious decision is required to omit the passed exception.

A clear benefit of this exception passing is that stack traces are more informative than they would be otherwise.

Messages

The Messages class provides a simple mechanism for accumulating information about errors and warnings. A Message consists of a severity int and a message String. The Messages that are accumulated can then be handled as a group, perhaps for presentation to the user.

Profiler

There is a profiler bundled with Darwino. There is no UI provided, but beyond the Java interface there is a REST service that can provide access to the profiler data. This is an application profiler, as opposed to a low-level profiler. It provides the ability to add hooks into application code to monitor high-level routines and then to dump that collected information later.

HttpClient

The HttpClient service is an easy-to-use implementation that is JSON-friendly. Methods such as getAsJson() (which parses the value and returns it as a JSON object), putAsJson(), deleteAsJson(), and postAsJson() simplify working with REST services. Because it works the same on all Darwino platforms, code implementing it doesn’t have to be concerned with platform-specific differences.

Tasks

A task is a piece of code that can be executed synchronously or asynchronously. Darwino’s task framework encapsulates the standard task execution implementation of each supported platform, allowing application code to remain unconcerned with the particulars of each platform.

When code executes a Task, it has access to the TaskExecutorService. Which enables passing parameters to tasks. Thus, tasks can run with different contexts. When the tasks’s execute() method is called, it is passed a TaskExecutorContext. This context contains the parameters.

If the platform’s task executor maintains progress information about its tasks, the Darwino’s TaskExecutorService can provide that progress information to the context. Darwino provides progress dialogs for the various platforms.

The TaskExecutorContext includes an updateUi() method with a Runnable that allows the backend task to update the user interface as needed. The UI task is then executed in the UI thread, which is required on client apps.

There is also a task scheduler. It allows one-time executions and scheduling by periodic intervals, and it supports time ranges (for example, “run hourly between 7:00am and 5:00pm”).

Tracer

The HttpTracerService can trace all of the requests that are coming to the server. As long as the requests are being served by the HttpService, the tracer (a managed bean) can be told precisely what should be traced. Tracing can be restricted to specific urls and particular types of data (such as headers, details, and content).

Mapping between a Darwino DB and a relational database

A Darwino database is mapped to a set of relational tables. These tables store all of the documents for all of the stores in all of the instances of the database.

To optimize the performance of the database, one would add indexes depending on the nature of the queries that are being performed. There is an art to this optimization; beside application-specific factors, there may be considerations related to the underlying database engine… Postgres, DB2, and MySQL could have different optimizations.

There is one set of relational tables per Darwino database. If the names of the tables depend on the database name, the definition of the tables is static for a given version of Darwino. This allows the tables to be created once by a DBA, and then used as-is, even when the application evolves.

For performance reasons, indexes on columns can be added. As the platform doesn't know most of the queries that will be executed by the application, it predefines a minimal set of indexes to speed up the generic access to the database (get a document by ID, index by key, synchronization...). But it is up to the application developer to track the requests being emitted by the database and then add additional indexes as required.

On the database systems that support native JSON access, JSON access indexes can also be added. Please refer to your database system documentation for best practices.

Darwino application tables

The prefix of Darwino's table names is the name of the Darwino database. This restricts us to names that are compatible with the rules of the relational database system. The suffix is always an underscore followed by three characters. Let’s take a look:

- _dsg: The Design table. This includes the lists of fields, stores, indexes, etc… There is at least one record in this table, with the value “DATABASE” in its type column. The name column is empty, and the json column contains the definition of the database. When you initialize a Darwino database, it will create this set of tables and store the database definition in that single record in the _dsg table.

In the database definition is a field called “version”. When you initialize the database, the version will be “1”. Every time the database definition changes, increment the number. This is important in Darwino because the application is disconnected from the database, so it is possible to have a database design that is not at the level expected by the application. When your code is opening the database, it will open this record from the _dsg table, extract the version value, and do a compare.

If it’s a match, it will open the database, return a handle, and off you go.

In another case, the application may be expecting to work with a higher version of the database. When opening, you specify how to update. One choice is to upgrade the database by running a function that you provide to give the new database definition.

If, on the other hand, the application is expecting a lower level of the database, it will fail. The application will not be able to update the database because it won’t know how.

In the DatabaseDef class, the loadDatabase method does the job of checking the version number and returning either null or a handle to the desired database (referenced by its name).

If the runtime itself has been updated, the “tableVersion” in the database definition comes into play. It is not managed by the application; it is managed by the runtime. If Darwino has been upgraded and needs to upgrade the design of its default tables, it will do so transparently to the user, as long as the RDBMS user has the rights to run DDL statements.

There is also versioning associated with the DatabaseCustomizer, which is where the developer, using a set of DDL statements, defines additional indexes, stored procedures, and triggers, typically for the purpose of optimizing the performance of the application. See Optimizing the database for details.

- _doc: The document table. There is one row per document. This row contains the JSON value as well as some metadata.
    - docid – autogenerated key value. This is dependent on the database and on the instance in the database.
    - instanceid – identifies the instance within the database that “contains” this document
    - storeid – along with the docid and instanceid, define the unique primary key of

  a document.
  - unid – The document's unique identifier
  - repid – ID of the server where this document was last created/modifed. This helps tracking where the doc comes from, and optimizing replication by not sending a document back to where it changed.
  - pstoreid – a pointer to the store of the parent document
  - parent – a pointer to the parent document
  - smstoreid – pointer to the syncmaster store
  - smunid – pointer to the syncmaster document
  - seqid – sequence number used internally in replication
  - updid – internal replication version ID
  - udate – the last replication time of the document. This is updated automatically
  - sftdel – soft delete flag, not currently used
  - cdate – creation date of the document
  - cuser – the user that created the document
  - mdate – the date of last modification
  - muser – the user that last modified the document
  - rsec – used internally to support reader fields
  - rsed – used internally to support writer fields
  - json – the JSON data of the document
  - sig – a signature for the document, not used
  - changes – used internally to support replication
  - cdatets – an easily readable and queryable copy of the creation date in timestamp format
  - mdatets – an easily readable and queryable copy of the modification date in timestamp format

  Note: the date fields (udate, cdate, and mdate) are stored as integers. They are the Java date converted to a long. They represent the number of milliseconds since the 1/1/70. This is to accommodate the precision required for replication, and the requirement that the dates be completely compatible with all possible relational database systems.

- _bin table: Used to store binary data associated with documents, but outside of the documents. Here data is stored with a computed key based on the hash of the file’s contents, and can be shared between multiple documents pointing to the same bin record.

- _dov table: This stores the list of fields extracted from documents. The extracted data is stored in one of four columns, one for each possible data type; they are named ftxt, fnum, fbol, and fdat.
- _idx: This is where indexes are stored. There are entries for each document, and for each entry there are stored keys and values.
- _idv: Like the _dov table, but for the index level, because we can store fields at the index level.
- _lck: This is for document locking; not currently used.
- _rep: Stores the replication information. It stores the last replication date for one replication profile. The last replication date for each replication profile is stored in the target of a replication. When pushing replication changes, the first step is to ask the target for the last replication time. The target checks this table and returns the value. The source then composes the list of changes and sends that. This is because you want to base the replication on the target's clock.
- _sec: Stores the reader and writer information. For every document, it stores the entry name and whether it's read-only or read/write.
- _sed: Like the security table, but for ereaders and ewriters.
- _stu: This is the deletion stub table, used during replication to convey that a document has been deleted.
- _tag: Stores the social data tags. It is indexed by the docid, and there is one row for every tag.
- _usr: The user-related social data, such as the rating and sharing information, as well as whether the document has been read and when it was last read. It also stores the replication time information for this data.

Database definition class

This class defines the JSON database, including the stores, the fields being extracted, the indexes, the security, and any other database options.

- setACL() is used to set the access levels of people, groups, and roles (who can read, edit, create documents, etc…). These access rights can be resolved dynamically. In particular, they can be resolved for an Instance.

- setDocumentSecurity(int documentSecurity) determines how readers and writers fields will be handled. Choices include no reader/writer security, reader only, writer only, etc...
- setInstanceEnabled(boolean instanceEnabled) – are Instances allowed or not.
- setPreventRestAccess(Boolean preventRestAccess) – Darwino provides a set of REST services so that data can be read and written via REST services. Disabling REST services prevents raw REST access to document data, ensuring that all access be through the appropriate business logic. If this is disabled, then, even if REST services are deployed, Darwino will deny REST access to the data.
- setReplicaID - internal
- setReplicationEnabled(boolean replicationEnabled) – If replication is enabled, Darwino records more data to support replication, including deletion stubs. This overhead can be avoided by disabling replication.
- setSoftDeleteEnabled – not implemented
- setTableVersion – internal
- setTimeZone – Darwino stores dates in ISO 8601 format, by default using GMT as the time zone. Setting this value will override that default. Dates will be stored instead using the specified time zone. There is never a loss of certainty; Darwino always stores the values with a time zone; this merely determines which zone is used as the default.

Stores

Physically, a store is nothing; it is just a concept. It is actually a logical collection of documents in a database. There is not one table per store; instead stores are implemented as a column value in each document. Every document in a database is identified by its UNID and its storeID; together, these two fields define the document's key. This way of implementing stores limits the actions needed to maintain the database's DDL, and it allows cross-store queries in the same database

Stores have several options:

- setAnonymousSocial(boolean anonymousSocial) – Enabling this allows tracking of the social activities of the anonymous user; for example, tracking when anonymous reads a document. This defaults to false, since there are few cases where it would be desired.
- setFTSearch(_FtSearch ftSearch) – Specifies what data should be extracted from the JSON to a table so it can be fulltext-indexed. If, for example, you might specify "$" if you want to index the entire document, or just the JSON path for the field

"Title" if you want that field to be the only indexed data.

- setFtSearchEnabled(boolean fulltextEnabled) – Enables full text search of the documents in the store. Darwino uses the full text search engine provided by the host database. This results in maximum performance and low overhead. It is possible to test at run time, using the Store-level method isFtSearchEnabled(), whether the database supports full text search, so the UI can be adjusted accordingly. None of the databases know, now, how to do full text search on the JSON documents. The _fts table contains the names and values of the fields that you wish to fulltext index.
- setLabel(String label) - User-friendly label displayed to the user.
- setPreventRestAccess(Boolean preventRestAccess) – If REST access is enabled at the database level, it can be prevented at the store level.
- setReadMarkEnabled(Boolean readMarkEnabled) – This applies to the social data read marks, and is set to false by default. If enabled, when a document is read by a user who is not anonymous (unless setAnonymousSocial is enabled), that document is marked as read by that user. This option exists so that the write operation required at every read to support read makrs can be avoided.
- setTaggingEnabled(boolean taggingEnabled) – Enabling this allows Darwino to maintain an array of tags for each document. You can search documents by a tag or a combination of tags. There is also a well-optimized function at the store level that returns a tag cloud.
- setFields() has two forms. The simple form takes the name of a field as its parameter and it indexes that field. The other form takes an array of FieldNodes.
- addQueryField() has five forms. The first takes one parameter, that being that name of the field. It will use that value both as the field name and as the path to the data. The next takes three parameters: the field name, the data type, and a Boolean determining whether the field is multiple. The third adds a specification of the path in the JSON. The fourth form takes a single parameter, this being a callback fieldFunction, which itself has several parameters: the field name, the data type, the multi Boolean, the name of a registered callback function and a JSON path to the data in the JSON document which acts as the parameter to the referenced callback function. The fifth form is like form #4, but uses a Darwino query language statement in place of the function name and parameter.

By allowing a callback function or query result, the function allows sophisticated processing to be performed when creating the field value, which can then be used in a query.

Indexes

In Darwino, an index is the MAP action in MAP/REDUCE. It allows fast access to data, as well as pre-computing of some data (ex: number of children, social data...) and then querying these data. It associates a key with a value for a selected set of documents. The value can be computed from the actual JSON document.

The store.addIndex() method creates an index based on a subset of the data in the JSON documents. Once you add the index, you define the keys and the values to extract from the JSON document.

When you execute a query on the index using the Darwino API, you can choose to return either the values in the index, or the JSON value in the document itself.

For example, index.keys(“_unid”) will set the unid as the key. index.valuesExtract(“\”$\””) will specify the root of the JSON value (the entire document) as the value to extract. This is using JSON Path expressions.

When specifying the keys, you can specify whether the keys are unique or not. This is done by calling the setUniqueKey(Boolean uniqueKey) method.

The Query Language

The query language in Darwino is modeled on the one used in MongoDB. This is unsurprising, given that MongoDB is a database for JSON documents. It uses JSON as a query language; the queries are JSON.

Darwino's query language is used to apply an evaluation formula on top of the JSON store, but it can be used beyond that. It can be used to query any kind of JSON document; the JSON doesn't have to reside in the database.

Working in the context of the JSON store, the query language is used to populate cursors. After opening a cursor, pass a query string or a JSON object as the argument to the cursor's query() method.

The syntax is straightforward. All queries are enclosed in braces. A very simple query would be a search for a specific value in a field at the document root level. This would take the form:

```
{city:'Springfield'}
```

This {fieldname:'value'} syntax is a shorthand form for {fieldname: {$eq: 'value'}}, using the "$eq" operator. This shorthand exists because most of the time we want to test for equality.

The field name can be either a literal string or a complete JSON path. The string does not have to be quoted; this is a benefit of Darwino's extension of JSON standard notation. JSON specs have field names between a pair of double quotes. Darwino respects that, but also permits single quotes or none at all.

Expanding on this example, we can add an "AND" operator:

```
{$and: [{city:'Springfield'},{state:'TX'}]}
```

Operators

Naturally, there is a set of comparison operators. Here is a partial list; more wil be added over time.

> Note: All operators start with the dollar sign. Arguments are typed: "1" is not equal to 1.

- $and - Filters on all documents that match every field/value pair in the provided array of conditions.

  `{$and: [{city:'Springfield'}, {state:'New York'}])`

- $contains - Is the first argument contained within the second? The search value can be an array or a string.

  `{array: {$contains: 'a'}}`

  `{string: {$contains: 'g'}}`

- $eq - Tests for equality. `{city: {$eq: ‘Springfield’}}`

  There is a handy shorthand version of this syntax, which does not require the "$eq". `{city:'Springfield'}`

- $exists - Tests whether a field exists or not.

  `{field1: {$exists: true}`

- $gte - Greater than or equal to.

  `{qty: {$gte: 20}}`

- $gt - Greater than.

  `{qty: {$gt: 19}}`

- $in - Test whether a value is in an array of values

  `{it: {$in: ['can','we','find','it']}}`

- $lte - Less than or equal to.

  `{qty: {$lte: 20}}`

- $lt - Less than.

  `{qty: {$lt: 21}}`

- $ne - Tests for inequality.

  `{Joe: {$ne: 'Joseph'}}`

- $nin - Tests whether a value is NOT in an array of values.

  `{findme: { $nin: ['can','we','find','it']} }`

- $nor - Returns documents that fail to match both conditions.

`{$nor: [{price: 1.99 }, {size: large'}]}`

- $not - Used in conjunction with other operators, it will negate the result, returning documents that do not match the query.

  `{cost: {$not: {$gt: 10.00}}}`

- $or - Returns documents that match both conditions.

  `{$or: [{price: {$lt: 10}}, { size: small'}]}`

- $path - Evaluates the provided JSON path and returns the value.

  `{$upperCase: {$path: 'g.il'}}`

- $type - Tests whether the value is of the specified type. `1: number` `2: string` `3: object` `4: array` `8: Boolean`

- $upperCase and $lowerCase - Converts the case of the argument.

  `{$upperCase: {'cobol'}}`

The MongoDB Query documentation serves as a good reference for the Darwino query language, as long as you keep in mind that the two are not 100% identical. When using the MongoDB reference, ignore the MongoDB API and focus on the query language itself.

Optimization

Darwino will take these queries and transform them to SQL as much as it can, given the capabilities of the underlying RDBMS for which the query generator is coded. There are three possible outcomes of this attempt:

- The entire query can be converted to SQL.
- Only fragments of the query can be converted.
- No conversion is possible.

In cases where only partial or no conversion is possible, Darwino will use the SQL that it CAN generate to first select from the database, and then it will "manually" filter the result fully to satisfy the query. It will do this by loading the resulting documents one at a time and then applying the remainder of the conditions against the in-memory document.

It is good practice when querying large datasets to keep in mind the capabilities of the RDBMS, and to write your queries in such a way as to permit the query generator to do the best possible job in translating the query into SQL. See Optimizing the database for details.

Extraction Language

A query will return the entire document; you will often be interested in only certain values from the document. Darwino can apply an extraction to the result of a query.

The extraction formula is in the form of a JSON document. The JSON document is a list of columns, and each column has a value. If the value is a literal string, it will be evaluated as a JSON path and the value of that path will be the result.

In this example, the result wil be two columns named "first" and "last", populated with the values found in the JSON paths "firstName" and "lastName" in the document.

```
{ first: "firstName", last: "lastName" }
```

Extending this, we can apply functions during the extraction to transform the results.

```
{ first: "firstName", last: {$upperCase: "lastName"}}
```

All of the query operators and functions can be used here, and it is also possible to add your own functions, making this a very powerful feature.

The extraction takes place server-side. While this clearly has the performance benefit that comes from not transmitting unneeded data to the client, it also allows your functions to utilize server resources, such as the cache and connections–such as to LDAP directories for lookups.

Aggregation

There is a set of aggregation operators in the query language, permitting actions such as counting and summing and categorization of query results. These operators apply on the entries belonging to categories. Similar to the extraction language, the syntax is a JSON document in which every entry is a column. Also similar to extraction, the order of the entries doesn't matter.

The aggregate operators ($count, $sum, $avg, $min, and $max) use a JSON path as a parameter.

```
{Count: {$count: "@manufacturer"}, Sum: {$sum: "@released"}, Avg: {$avg: "@release
d"},
        Min: {$min: "@released"}, Max: {$max: "@released"}}
```

Behind the scenes, Darwino constructs a SQL query that uses the database's native aggregation operators. This is good in terms of efficiency: the document selection, value extraction, and aggregation is done server-side using efficient SQL statements. The database does the work.