Chapter 12

# Business Rules Scripting

**NOTE**: This chapter is published as a preview of our upcoming book on Enterprise and Cloud Database Development with Data Abstract for Xcode. It should be considered a first draft, and may be subject to change and revisions before final publication. This current draft is based on the “May 2011” releases of Data Abstract and Relativity Server.

In the previous chapter, you learned all about Schemas and how they contain the information your middle-tier server needs to know how to make data available to clients and – if necessary – to abstract the underlying physical data model in the database or databases from what is exposed to the clients. In this chapter, we’ll look at the one remaining piece of the puzzle necessary for creating a complete middle tier: Business Logic, and more specifically, Business Rules Scripting.

## What is Business Rules Scripting?

As we already discussed in previous chapters, and will delve into more deeply in *Part Three*, one of the central features of a middle-tier server is that it retains complete control over how client applications access data, what data particular clients are allowed to access, and how they are allowed to modify it. Business Rules Scripts are one way for Data Abstract middle-tier servers to define the business logic and implement rules, constraints and secondary logic that needs to take place in the middle tier for safe data access. This is done through snippets of JavaScript code that talk to a platform-agnostic API provided by Data Abstract and Relativity Server in order to inspect, modify or reject data requests and data updates received from the client.

When creating custom Data Abstract servers (a topic that is beyond the scope of this book, but something we touch on briefly in *Appendix E*), business rules scripting is only one of two options for implementing business logic, since custom servers can also contain custom code written using the development platform chosen for the server (typically a .NET/Mono language such as C# or Oxygene). However, when hosting your middle tier in Relativity Server, scripts are the sole means for expressing business logic.

In theory, schemas distinguish between two types of business rules scripts: scripts that run inside the server and scripts that get transferred to and will run on the client. However, Apple's current App Store Guidelines do not permit apps to download and execute custom code (including script code), so client-side scripting is not currently supported for iOS development.

## Server Side Scripts

Server-side scripting is the most important side of business rules scripts, and used to globally enforce data integrity on the server. One of the tenets of a secure multi-tier architecture is to have the middle tier validate all data access and ensure the data integrity, and never solely rely on the client application to submit proper data. A secure middle tier needs to cope with bad data received from buggy or malicious clients and ensure that no invalid data gets past its business rules checks. Among other infrastructure provided by the Data Abstract middle tier itself, server-side business rules should perform this task.

Scripts are stored within the Schema and developed within Schema Modeler. Written in JavaScript (a.k.a. ECMAScript), these can contain just about any logic that can be expressed in that language; they can, for example, define custom functions and classes, if needed, or use any standard JavaScript code constructs or APIs. The main entry points for business rules scripts are a set of well-defined events that Data Abstracts allows the scripting code to handle by implementing functions. Data Abstract also provides an API that scripts can use to interact with the system, query data, etc. (comparable to how the scripting engine of a web browser exposes an API for working with the document model of the HTML page the browser is showing).

There are three levels at which the middle tier provides events that can be handled.

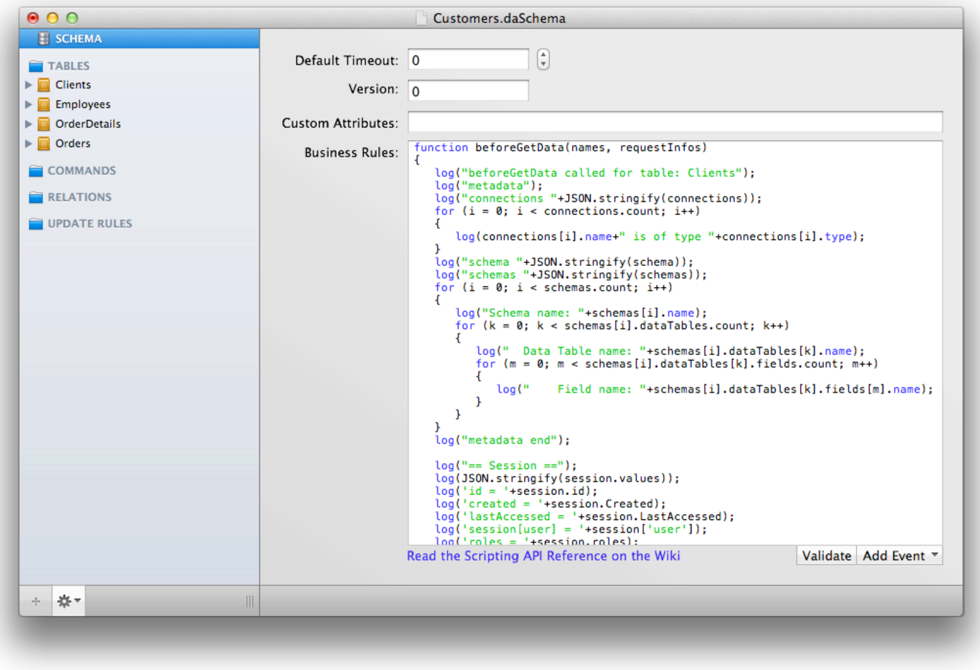
- Some events are associated with specific objects within the schema and their handlers should be written on a per-object basis. For example, one of the most common events to be handled is `beforeProcessDeltaChange()`, which gets called before a particular change gets applied to the database; this event is most commonly handled per-table, with an event handler that performs checks that pertain to the table in question.
- Other events are specified globally for the entire schema. For example, the `beforeCommit()` event is fired before a transaction is committed to the database. Since transactions can span multiple different tables, there is little sense in implementing `beforeCommit()` for each table; instead, the event has one handler that applies to the entire schema.
- Finally, there is a set of events that is global to an entire domain in Relativity Server, even if that domain contains multiple Schemas. For example, the `afterLogin()` event is called after a client has authenticated itself with the server.

We will be looking at these events and this API in far more detail throughout the course of this chapter, and see how to achieve many common tasks through scripting. A thorough overview of all the events and the entire scripting API provided by Data Abstract can also be found online at `http://remobjects.com/wiki/Business_Rules_Scripting_API`.

## Editing Scripts in Schema Modeler

Schema Modeler provides two places where scripts can be added to a schema.

The root “SCHEMA” node provides an area for scripts and event handlers that will apply to the entire schema, regardless of what objects the events may pertain to. This is the place to implement events that are not tied to a specific table (such as the afore-mentioned `beforeCommit()`), but you can also put handlers for table-specific events in this area; these event handlers would then trigger for any table in the schema. For example, if you implemented a handler for beforeProcessDeltaChange() here, it would be called for any change to any of the tables in the schema. Your event handler code would need to take this into account, for example by checking the name of the table and adjusting its checks accordingly.



Default Timeout: 0

Version: 0

Custom Attributes:

Business Rules:

```
function beforeGetData(names, requestInfos)
{
   log("beforeGetData called for table: Clients");
   log("metadata");
   log("connections "+JSON.stringify(connections));
   for (i = 0; i < connections.count; i++)
   {
      log(connections[i].name+" is of type "+connections[i].type);
   }
   log("schema "+JSON.stringify(schema));
   log("schemas "+JSON.stringify(schemas));
   for (i = 0; i < schemas.count; i++)
   {
      log("Schema name: "+schemas[i].name);
      for (k = 0; k < schemas[i].dataTables.count; k++)
      {
         log("  Data Table name: "+schemas[i].dataTables[k].name);
         for (m = 0; m < schemas[i].dataTables[k].fields.count; m++)
         {
            log("    Field name: "+schemas[i].dataTables[k].fields[m].name);
         }
      }
   }
   log("metadata end");

   log("== Session ==");
   log(JSON.stringify(session.values));
   log('id = '+session.id);
   log('created = '+session.Created);
   log('lastAccessed = '+session.LastAccessed);
   log('session[user] = '+session['user']);
   log('roles = '+session.roles);
```

Read the Scripting API Reference on the Wiki

*Figure 12-1: Schema-wide Events*

Each Data Table and Command node also has an area for writing event handlers that pertain to only one specific object. For example, if you implemented a handler for `beforeProcessDeltaChange()` under the *Clients* table, it would only trigger for changes to *Customers*, but not for changes made to – say – *Orders*. This allows you to write more to-the-point and table-specific checks.

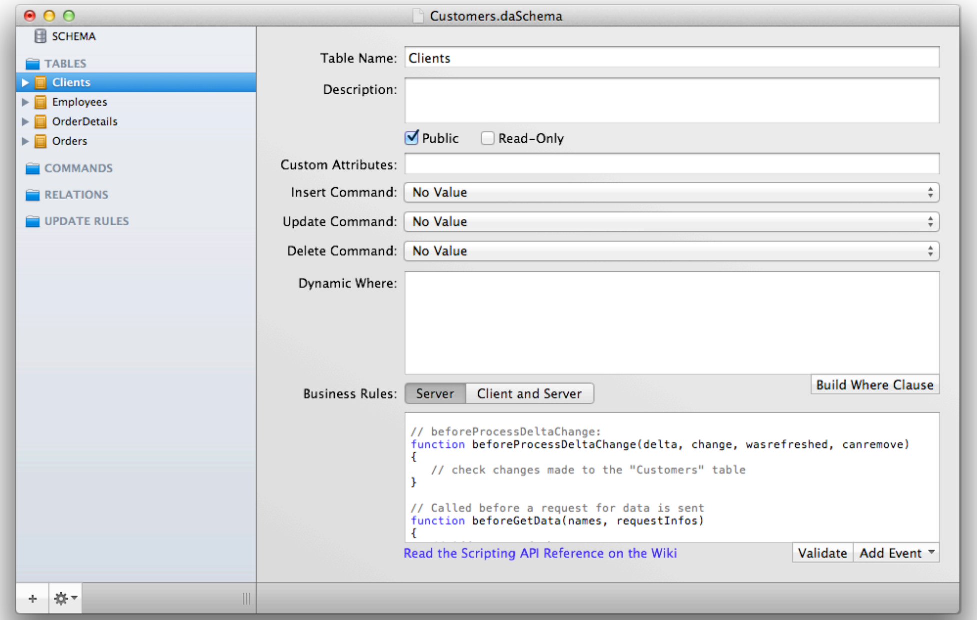


Table Name: Clients
Description:
[✓ Public] Read-Only
Custom Attributes:
Insert Command: No Value
Update Command: No Value
Delete Command: No Value
Dynamic Where:
Business Rules: [✓ Server] Client and Server

```
// beforeProcessDeltaChange:
function beforeProcessDeltaChange(delta, change, wasrefreshed, canremove)
{
   // check changes made to the "Customers" table
}

// Called before a request for data is sent
function beforeGetData(names, requestInfos)
{
```

*Figure 12-2: Per-Data Table Events*

Attached to the bottom of each script area you will find two buttons. The right-most button, titled "Add Event", will open a popup menu listing all the event handlers that are applicable to the current context. Select an event name from the list, and Schema Modeler will automatically add an event handler body to the script (or, if a handler for the event already exists, jump to the existing code block). The second button, titled "Validate" can be used to check the script for syntactical errors. (But note that since JavaScript is a very dynamic language, errors can still occur at runtime when the script is being executed, even if the syntax check succeeded. Thorough testing of your business rules scripts before deployment is important – more on that toward the end of this chapter)

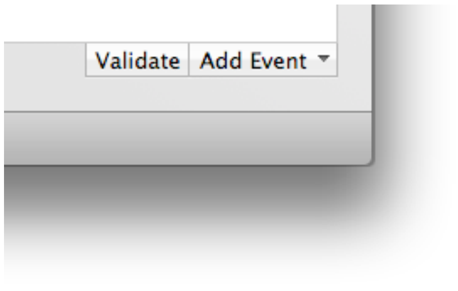


*Figure 12-3: Scipt Editor Tasks*

## The Scripting API

Apart from the general JavaScript syntax and JavaScript's standard object model (both of which are not the subject of this book, but can be found explained elsewhere; *Appendix F* provides a few references to suggested reading) there are essentially two areas that make up the API provided by Data Abstract and Relativity. On the one hand, there are the event handlers we already briefly touched upon in the previous sections of this chapter. There really is no need to "learn" all the different event handlers (there's about 25 of them), as Schema Modeler makes it easy to add stubs when necessary. The second and more comprehensive part of the API is the object model provided by Data Abstract. Some of these objects will be available globally, while others will be passed in as parameters to your event handlers. Let's have a brief look at what's available.

The Events can roughly be grouped into six categories:

- **Transactions.** These are schema-wide events that allow you to be notified and participate in the transaction management done by Data Abstract. These are `onCreateNewTransaction()`, `beforeCommit()`, `afterCommit()` , `beforeRollback()` and `afterRollback()`. The names are pretty self-explanatory – the first event notifies that a new transaction has been created, while the remaining four events trigger before or after a transaction is either committed or rolled back.
- **Deltas**. These events deal with the handling of changes or groups of changes received form a client, and can, for example, be used to inspect or validate changes in order to approve or reject them.

These include `beforeProcessDelta()`, `afterProcessDelta()`, `beforeProcessDeltaChange()` and `afterProcessDeltaChange()`.

- **Data Requests.** These events get triggered when clients request access to data. They are `beforeGetData()`, `afterGetData()` and `onValidateDataTableAccess()`.
- **Command Requests.** These events trigger when clients try to execute data commands (a topic we're not covering in this book), and are `beforeExecuteCommand()`, `afterExecuteCommand()` and `onValidateCommandAccess()`.
- **Error Handling.** The `onProcessError()` event is triggered whenever a delta change is rejected; it allows the scripting code to react to such failures.
- Finally, there are **Relativity Server-specific events**, including `afterLogin()`, `beforeLogout()` and `afterLogout()`.

A complete reference of all events, along with detailed descriptions and documentation can be found online at `http://remobjects.com/wiki/Business_Rules_Scripting_Events`.

The list of object types provided by Data Abstract Scripting can also be grouped into a handful of categories:

- Connection information
- Schemas Metadata
- Deltas and Delta Changes
- Dynamic Where Clauses
- Local Data Adapter
- Session Management

As mentioned before, access to these objects is usually provided via parameters passed into your event handlers (for example the `beforeProcessDelta()` event, which will pass you the `Delta` object representing the set of changes that will be processed), or are available globally (for example the global `session` object, which gives you access to the session data for the current client, or the local `lda` object that gives you access to a *Local Data Adapter* you can use to query data). A reference of all object types as well as all global objects and functions is also available online at the above-mentioned URL.

You will see all of this in more detail when we look at concrete examples. So let's write scripts for some common business rules tasks.

## Validating Changes

The `beforeProcessDelta()` and `beforeProcessDeltaChange()` events can be used to inspect, adjust and act upon changes received from the client before those changes get applied to the database. What's the difference between these two events? A *delta* is a set of changes received from the client in one batch, and essentially a group of individual *delta changes*. A delta might contain only a single change, or it might contain multiple related or unrelated changes to the same table. As a consequence, each time a client sends a delta to the server, `beforeProcessDelta()` will be called once, followed by separate calls to `beforeProcessDeltaChange()` for each individual change.

```
function beforeProcessDelta(delta) {}
function beforeProcessDeltaChange(delta, change, wasRefreshed, canRemove) {}
```

For data validation, you will usually want to inspect each change separately, so you have two choices: you can implement `beforeProcessDelta()` and loop across all the changes in the delta yourself, or you can implement `beforeProcessDeltaChange()` to handle each change individually:

```
function beforeProcessDelta(delta)
{
  for (int j; j < delta.count; j++)
  {
    var change = delta[j];
    // Inspect "change"
  }
}
```

or

```
function beforeProcessDeltaChange(delta, change, wasRefreshed, canRemove)
{
       // Inspect "change"
}
```

In both cases, `change` will be a `DeltaChange` object that gives you detailed access to everything you need to know about the change received from the client. Let’s have a closer look at the members exposed by `DeltaChange`.

The three boolean properties `isInsert`, `isUpdate` and `isDelete` let you determine whether the change represents a new row created on the client, a change to an existing row or a removal, respectively. Usually, you will want to perform different sets of validation, depending on the change type (after all, it makes little sense to check the consistency of fields in a row slated to be deleted, but it might make sense to check if the user is allowed to delete rows).

The `newValues[]` indexer give you access to the new (or changed) values of the database row, allowing you to validate the data to see if the changes are acceptable or not. Similarly, the `oldValues[]` indexer contains the old values, as the client saw them before making the change. Both of these indexers can be accessed by field name or index, such as `newValues[“Address”]` to get a value by name, or `newValues[3]` to get the value of the fourth (indexes start at 0, of course) field.

```
function beforeProcessDeltaChange(delta, change, wasRefreshed, canRemove)
{
  if (change.isUpdate)
  {
    if (change.newValues[“Address”] == “Happy St.”)
      fail(“Can’t move to Happy Street!”);
  }
}
```

It is important to realize that, to save network bandwidth, a delta change will usually not include *all* the fields of the table, but only those fields that have actually changed, as well as the Primary Key. For example, if the client changed the “Address” field of a row in our customers table, the delta change might contain the “ClientID” and of course the “Address”, but it will not have a valid value for – say – the “Name”. To make more complex validations easier and give access to the full row of data, two additional indexers are provided: `originalRow[]` and `finalRow[]`. As the names imply, `originalRow[]` gives your code access to the original data in the row as it is found in the database at that point in time (for obvious reasons, this is not available for new inserts), while `finalRow[]` gives your code a preview of what the final row would look like in the database, if the delta change was to be successfully applied – including all the changed and unchanged fields.

**But Note**: Since the full detail of the row is not included in the delta, a request to the physical database is required in order to populate the `originalRow[]` and `finalRow[]` values. As such, accessing these indexers can be costly, and should be used with care, especially for tables where a high volume of updates is expected.

```
if (change.isInsert || change.isUpdate)
{
  if (finalRow["Country"] == "USA" & finalRow["State"] == null)
    fail("A state is required for addresses in the U.S.");
}
```

The above code works reliably, because `finalRow[]` always gives access to the full row. If the code used `newValues[]` instead, the check would be unreliable, as the client app might have changed the "Country", but not the "State", and `newValues[]` would have no information as to whether the "State" field actually contains a value or not.

In addition to the use of the `DeltaChange` object, you might have noticed that our script also contains a call to `fail()`. `fail()` is one of the global methods provided by the Business Rules Scripting API, and it essentially aborts the current method with an exception, which in turn causes the Data Abstract framework to reject the change and return the error message to the client. To the client, this will look the same as any other failure (such as an error returned from the back-end database).

As you have seen in earlier chapters, the client can use this error to present a proper UI and ask the user to correct the data to a valid format.

## Working with Sessions

Relativity Server uses the concept of *Sessions* to maintain the identity of a client between requests. For scalability reasons, multi-tier servers treat every request individually, and Relativity Server doesn't keep a dedicated DataService instance around for each client. Instead, each request received from a client will be served by a random available service instance in order to keep a low memory profile. (This is a concept discussed in more detail in *Part Three.*) The only thing tying individual requests from the same client together is the Session.

You can think of a session as a collection of named values that the server stores in a central location (possibly even shared by multiple physical servers), identified by a Client ID (also known as a Session ID).

The first time a client authenticates with Relativity Server, a new session is created using the Client ID, a unique GUID transmitted from the client (it is important to note that Client IDs are usually generated when the client application starts, so each Cient ID uniquely identifies a particular *instance* of the client application). Relativity Server stores a few relevant pieces of information in the session (such as what domain the client logged into, what username it logged in with, etc.), and this information is then available for all future requests the server receives from the same client (i.e. with the same Client ID), until the session expires – typically after 20 minutes of inactivity.

In addition to Relativity-specific data, business rules scripts can also store custom data inside the session, to preserve it from one request to the next.

The first contact your script has with the session is in the `afterLogin()` event, which you can implement on the schema level to get informed when a new session was created and a new user logged in:

```
function afterLogin(userName, parameters)
{
  log("after login");
  log("user "+session[“Relativity.UserName”]);
  log(JSON.stringify(session.values));
  log("after login end");
}
```

The event handler above will print out the content of the freshly initialized session to the server log:

[Screenshot pending some product changes, will be delivered later]

*Figure 12-4: Server Log Showing Session Data*

You will notice that all values set by Relativity Server start with a “Relativity.” Prefix; these values are read-only, meaning your script can evaluate them, but not change them. Your script is completely free to store any values you see fit, without a prefix, as in the following sample:

```
function afterLogin(userName, parameters)
{
  session[“MyCustomValue”] = “Hello”;
  log(“Custom Value: “+session[“MyCustomValue”]);
  log(“User Name: “+session[“Relativity.Username”]);
  //session[“Relativity.Username”] = “Peter”; //this would fail
}
```

The main reason to store custom values in the session is to preserve information for future calls. For example, you might use the `afterLogin()` event to perform a (maybe relatively costly) check to determine what tables the authenticated user is allowed access and store a flag in the session with the result. On subsequent requests, your scripts can just check that flag in the session, for example in the onValidateDataTableAccess:

```
function afterLogin(userID, parameters)
{
  if (session[“Relativity.Username”] == “admin”)
    session[“CanAccessEmployees”] = true;
}
…
function onValidateDataTableAccess(name, parameterNames, parameterValues,
currentAllowed)
{
  if (name = “Employees” & !session[“CanAccessEmployees”])
    fail(“You’re not allowed to access the Employees table.”);
}
```

The above sample is trivial (and in fact you might just as easily check `session[“Relativity.Username”]` inside `onValidateDataTableAccess()`), but you could imagine a more elaborate check that would, for example look up a *Permissions* table from the database based on the value of `session[“Relativity.Username”]`. In one of the projects we have worked on in the past, we used the `afterLogin()` event to build a (complex) “Dynamic Where” clause that specified which projects in a bug tracking system the user had access to. This where clause was stored in the session and later (within the `beforeGetData()` and `beforeProcessDeltaChange()` events) appended to any queries received from the client, in order to filter the *Projects* and *Bugs* the user could see. In essence, this allowed us to dynamically adjust the business rules for data access for each individual user.

Depending on the Login Provider your Domain is using (Login Providers were discussed briefly in *Chapter 10*), there might be additional helpful values stored in the session for you. For example, if you are using the DbTableLoginProvider, it will automatically populate the “User.” namespace within the session with all fields from the user’s record in the database. For example, if your Users table looked like this:

- ID
- Username
- Password
- IsAdmin

*Figure 12-5: Users Table*

you could use `session[“User.IsAdmin”]` to check that flag from within the scripts, without needing to run any extra database requests.

## Accessing Additional Data Using the Local Data Adapter

In many cases, your business rules will need to rely on additional data from the database that is not directly available via the session or via the request that is currently being processed. For example, as we hinted at in the previous section, you might need to query a Permissions table to determine what sort of operations the current user is allowed to perform. Or maybe you need to validate a change against different tables in the database before allowing a delta to be processed. In our bug tracker project, for example, we needed to check if a given *Category* was valid for the *Project* a *Bug* was in – a check where a separate *ProjectCategories* relation needed to be consulted for.

The scripting API provides an object for these sort of tasks, the Local Data Adapter, available though the global `lda` variable. As the name implies, the LDA is a close relative of the *Remote Data Adapter* we already used on the client side, in *Chapters 2* through *7*. It provides much of the same functionality – querying data and applying updates – except it does this *locally* within the middle tier, rather than *remotely* from a client.

The most commonly used function of the LDA is `selectSQL()`, which executes a DA SQL query against the schema and returns a recordset with the resulting rows. For example, we could use

```
var users = lda.selectSQL(“SELECT * FROM Users”);
```

to retrieve the records in a *Users* table to work with locally. Or, given the name of the currently logged user, we could request a list of permissions for that user (for example to check what operations we want to allow this user to perform):

```
var permissions = lda.selectSQL('SELECT * FROM UserPermissions WHERE UserName =
@UserName”, { UserName: session[“Relativity.Username”] });
```

Note how this second call uses parameters, rather than directly putting the UserID into the DA SQL. It is safe practice to never put external values directly into a query using string concatenation, to avoid SQL injection attacks (although we are fairly safe in this case, because we can rely on `session[“Relativity.Username”]` to have a proper value and also, DA SQL itself protects from SQL injection very well, since it does not directly pass SQL through the database, and most SQL injection attacks rely on SQL commands that DA SQL does not permit).

Once you have a result from selectSQL, you can loop over it to access the individual rows, for example:

```
var clients = lda.selectSQL(“SELECT * FROM Clients”)
for (i = 0; i < clients.count; i++)
{
  var client = clients[i];
  log(clients[0]+”: “+clients[“ClientName”]);
}
```

As you can see, the select result exposes a count property, and acts as an indexer letting you access rows 0 through `count-1`. Each row itself can be indexed either by field name or field index to access the individual field values; the above example would print out the *ClientID* (the first defined field) and the *ClientName* of each client.

In addition to querying data, you can also use the LDA to make changes to the database. The LDA exposes three functions for that, aptly named insert(), update() and delete().

You can insert a new row to the Clients table like this:

```
lda.insert(“Clients”, { ClientName: “Peter Venkman” } );
```

update a row like this:

```
lda.update(“Workers”, { WorkerID: 122 }, { WorkerLastName: “Frankie” });
```

and delete a row like this:

```
lda.delete(“Orders”, { OrderId: 1109 });
```

In all three cases, the first parameter to `insert()/update()/delete()` is the name of the data table you want to change. The second (and in case of `update()`, third) parameter is an object containing name/value pairs for field names and their values. For inserting a row, only one set of values is needed, the new values to be inserted. For updates, the first object needs to provide all the fields that make up the primary key (in our example just the *WorkerID*) to identify the row, while the second provides all the fields you want to update. Finally, for deleting a row, you once again need one object with all the fields of the primary key to identify the row in question.

Just like when making changes to data tables on the client, calling `insert()/update()/detete()` just creates local delta changes in memory. It is not until you call `lda.applyChanges()`, that those changes actually get applied to the database.

Of course, since you are writing scripts to create business logic that reacts to "real" changes from a client, you will, in most cases, use `insert()/update()/detete()` in combination with information received from a client as part of a change request. For example, if we look back at the `beforeProcessDeltaChange()` event we discussed earlier, we might write:

```
//BeforeProcessDeltaChange() for Orders
function beforeProcessDeltaChange(delta, change, wasRefreshed, canRemove)
{
  if (change.isUpdate)
  {
    lda.insert("OrderHistory", { ID: newGuid(),
                                 OrderID = change.originalRow["OrderId"],
                                 User: session["Relativity.UserName"],
                                 Date: new Date() });
    lda.ApplyChanges();
  }
}
```

to record a history of all changes made to an order, in a separate *OrderHistory* table. Note how we construct the new row partially from new data (a new GUID and the current date) and partially from information in the session and in the received delta change.

## Inspecting Schema and Connection Meta Data

The scripting API also provides access to metadata about the domain it is running in, including access to a list of defined connections, as well as to all the schemas contained within the domain. These are available via the global `connections` and `schemas` variables, respectively. In addition, the global `schema` variable lets your script directly access the current schema.

The following snippet prints a list of all connections and their types (connection types were discussed in *Chapter 10: Relativity Server and Server Explorer*):

```
for (i = 0; i < connections.count; i++)
{
  log(connections[i].name+" is of type "+connections[i].type);
}
```

For our sample domain, it will print out:

```
PCTrade is of type SQLite
```

The following, instead, prints out all the schemas, along with all the data tables they contain and all their fields:

```
for (i = 0; i < schemas.count; i++)
{
  log("Schema name: "+schemas[i].name);
  for (k = 0; k < schemas[i].dataTables.count; k++)
  {
    log("  Data Table name: "+schemas[i].dataTables[k].name);
    for (m = 0; m < schemas[i].dataTables[k].fields.count; m++)
    {
      log("    Field name: "+schemas[i].dataTables[k].fields[m].name);
    }
  }
}
```

All metadata classes provide access to all the information that's defined in the schema itself. For example, you could write a generic script handler that triggers for all tables in your schema and then checks the metadata of the table (or its fields) for custom attributes or other information to control what can be done. For example:

```
function onValidateDataTableAccess(name, parameterNames, parameterValues,
                                   currentAllowed)
{
  var requiredRole = schema.dataTables[name].customAttributes;
  if (requiredRole & !session.hasRole(requireRole))
    fail(“”Access to table “+name+” requires role “+requiredRole+”.”);
  return true; // else, allow access
}
```

In this example, the schema could contain the name of a role in the custom attribute field of each data table, and, if present, `onValidateDataTableAccess()` would ensure that this role is defined for the session, before allowing access. This way, access can be configured for each table, without modifying the scripts every time.

## Debugging Scripts

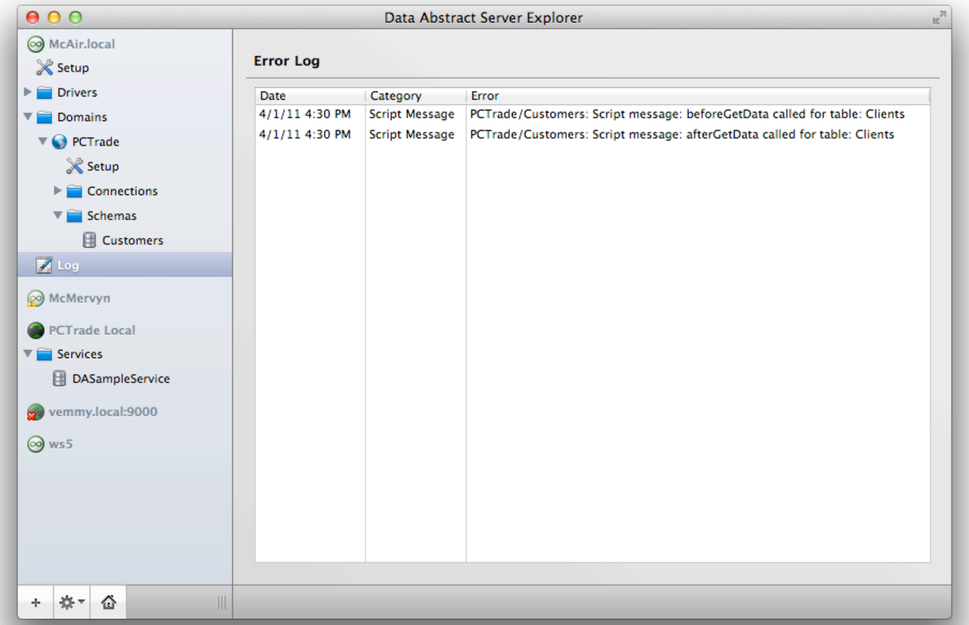
Compared to languages like C and Objective-C, JavaScript is a very dynamic language, and as such it is difficult for Relativity Server to fully verify a script “dry”, without actually running it. In strongly typed languages, the compiler knows what types look like and what members they expose, and can give you errors (or, as is the case with Objective-C, warnings), if you are accessing members that do not or may not exist. With JavaScript, this is not possible, because most objects your scripts deal with will be dynamically created at runtime.

Nonetheless, Schema Modeler’s script editor provides a “Validate” button that lets the server perform a basic set of validations to make sure your script is ok, syntactically. But even a script that validates can still fail at runtime, if you, say, mistyped an object member or variable name. And of course, in addition to containing such errors, your script might also contain more traditional *bugs*, i.e. be written to be formally correct, but not behave the way you expect it to.

For that reason, it is important you test business rules scripts thoroughly before deploying your server for production use, and make sure that every code path is executed during your testing. For example, you might have a code path that only gets executed when a client sends a certain piece of bad data – if in your testing your client does not make this mistake, your corresponding script code will not get tested, and might therefore contain a bug that will come around to rear its ugly head much later, once your system is in production. This is of course not good.

Because scripts run in Relativity Server, not locally in Schema Modeler, debugging scripts can be tricky, and at the time of this writing, Relativity and Schema Modeler do not yet provide a fully interactive debugging environment where you could set breakpoints and step through scripts – although the Data Abstract team is working on such a solution, and it might in fact be available by the time you read this.

In the meantime, Relativity Server provides a couple of things that will help you test and debug scripts. For one, there is the `log()` function that you can use in your scripts to send texts to the Server Log, which you can then see in the Log node in Server Explorer, as shown in Figure 12-6.

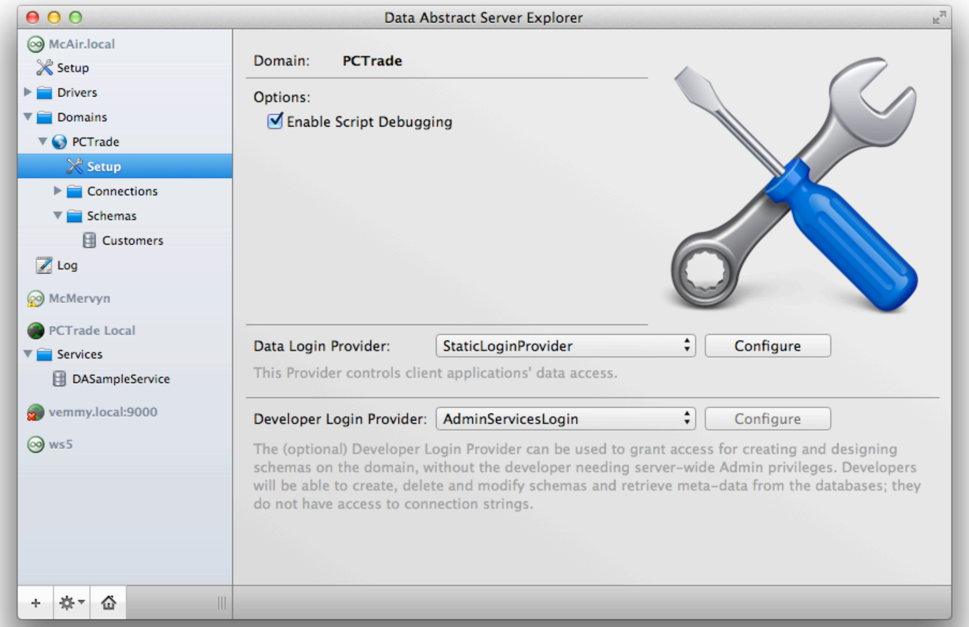


Error Log

| Date | Category | Error |
|---|---|---|
| 4/1/11 4:30 PM | Script Message | PCTrade/Customers: Script message: beforeGetData called for table: Clients |
| 4/1/11 4:30 PM | Script Message | PCTrade/Customers: Script message: afterGetData called for table: Clients |

*Figure 12-6: The Server Log*

In addition, there is a flag to “Enable Script Debugging” for a domain in Relativity Server, available on the domains’s Setup page. While this does not (yet) enable a full interactive debugging experience, as discussed above, enabling this will cause the script engine to run with extra debug information, and to, for example, report back exact error locations as part of the exception that gets sent back to your client app when a script fails.



Domain: PCTrade

Options:
☑ Enable Script Debugging

Data Login Provider: StaticLoginProvider
This Provider controls client applications' data access.

Developer Login Provider: AdminServicesLogin
The (optional) Developer Login Provider can be used to grant access for creating and designing schemas on the domain, without the developer needing server-wide Admin privileges. Developers will be able to create, delete and modify schemas and retrieve meta-data from the databases; they do not have access to connection strings.

*Figiure 12-7: Enable Script Debugging*

## Summary

This chapter rounded off our coverage of creating middle-tier servers in Relativity Server by looking at how Business Rules Scripting allows you to define business logic and control data access with JavaScript.