Using SQL to access a REST service requires a little extra syntax. Pyrrho allows a modified CREATE VIEW syntax called RESTView:

```
CREATE VIEW id OF rowtype AS GET url
```

where rowtype is like a table definition, giving column names and domains. The url part can be a single-quotes string, but syntactically it is just Pyrrho-specific Metadata and various flags and additional items are possible for specifying mime types etc. Full details are in the Pyrrho manual, and we will see some examples in this demo.

To support transactions, we base our implementation on the use of entity-tags as in RFC 7232 (Fielding & Reschke, 2014). Very few actual REST services comply with this standard, so Pyrrho provides a suitable service. Then, as we will see, a REST round-trip is a branch transaction, that occurs at the main transaction commit point.

The motivation for this example is the classic example of a transfer between two bank accounts, but here it is between two different banks, each with their own databases and need for serialized transactions. Communication between the banks will be over the Internet.

[Slide 158 @ 57:37 in the video]

This time the PyrrhoSvr needs to be started up with the command line shown to start the HTTP service (by default at localhost:8180) and provide some diagnostic feedback:

```
PyrrhoSvr –d:\DATA +s –H
```

We check the database folder is empty and provide blue and green windows for the two bank clients.

[159 @ 55:22]
Names of banks and accounts are kept short to fit on the presentation slides. In the blue window, give the command

PyrrhoCmd DB

to create the database for bank B. At the SQL> prompt set up an accounts table for bank B with the SQL statements

    create table AB (id int, bal int)
    insert into AB values (100,1000),(110,1400)

This gives us two bank accounts id=100 with balance 1000, and 110 with balance 1400.

To enable access to this database over the network, at least one role and one user are required:

    create role DB
    grant DB to "current user"

where the current user identity is inserted surrounded by straight double-quotes.

And we define a RESTView for bank C to use:

    create view RC of (id int,bal int) as get etag milli 'http://localhost:8180/DC/DC/AC'

The url is enclosed in single-quotes as specified above, and the two metadata flags tell Pyrrho to enforce RFC 7232 with millisecond precision.

Do the same, mutatis mutandis, for bank C:

PyrrhoCmd DC

and then

    create table AC (id int, bal int)
    insert into AC values (200,2000),(220,1800)
    create role DC
grant DC to "current user"
create view RB of (id int,bal int) as get etag milli 'http://localhost:8180/DB/DB/AB'

This gives us two bank accounts id=200 with balance 2000, and 220 with balance 1800.

We refer to these windows as blue and green for clarity. We now have the following situation:

The current state of the account in the two banks can be verified by table AB in the blue window and table AC in the green window.
Two overlapping transactions will conflict on account 200.

In this slide, bank B starts a transaction and takes £50 from account 100 to be transferred. Nothing is written to disk yet.

```
begin transaction
update AB set bal=bal-50 where id=100
```

The report on the update makes the point that the new record is in the transaction, rather than the database.

[163 @ 1:02:12]

In the green window, someone in bank C also starts a transaction to transfer money from account 200 to remote account 110.

```
bEGIN TRANSACTION
UPDATE AC SET bal=bal-30 WHERE id=200
```
Back in the blue window, bank C is taking the next step in its transaction, adding the £50 to remote account 200.

```sql
update RC set bal=bal+50 where id=200
```

This is still in the transaction, and nothing has been committed.

However we see Internet activity as bank B checks that 200 is an account in bank C. we suppose that the balance data will be withheld, because of the permissions, but bank B does receive an ETag. The response on bank B indicates that the changes are not for the local bank.

It is hoped that COUNT(*) can (always) be used instead of selecting columns to create ETags: will become clearer during the next phase of implementation.
The green teller also starts a transfer, of their £30 from account 200, to account 110, and receives an ETag for remote account 110.

At this point both tellers are about to commit their transaction. The first to do so will succeed, but no changes have been made yet.

[166 @ 1:03:38]

The green teller gets there first, and commits their transaction.
Looking at the feedback in the server window: The first step is a HEAD request to verify the ETag sent to the green database is still valid. (It is.) Then the update is POSTed to bank B.

On receiving OK from the remote bank B, the change is committed to local account 200 in bank C. This will invalidate the ETag previously sent to the blue client.

The response to the commit indicates that the change to local account 200 is now recorded durably in the database.

[167 @ 1:04:20]

Sure enough, when the blue client attempts to commit, and sends its change to bank C, bank C reports that the ETag is invalid, and the transaction is aborted. The failure of this HEAD roundtrip rolls back the blue transaction entirely.
In this case, if the green client committed first, the local update would fail and a normal (non-ETag based) transaction conflict condition would be raised.

[168 @ 1:04:53]

The teller in bank B can verify that the transfer from account 100 has not occurred, while bank C’s transaction transferring £30 has been committed in both databases.

[169 @ 11:05:11]

As an extra demo, we will use Visual Studio to explore RowSet Review, using as an example a viewed join of a local table with a remote view. This is part of the test23 suite in the Pyrrho V7 alpha distribution, where it is shown to be updatable.

The example is a simplified system of a company employee database that records job titles and salaries, but where the employees’ names are stored in another database.

Database A:
create table D(e int primary key,f char,g char)
insert into D values (1,'Joe','Soap'), (2,'Betty','Boop')
create role A; grant A to "MALCOLM1\Malcolm"

Database B:
create view W of (e int, f char, g char) as get etag milli 'http://localhost:8180/A/A/D'
create table H (e int primary key, k char, m int)
insert into H values (1,'Cleaner',12500), (2,'Manager',31400)
create view V as select * from W natural join H

We start as before with a debug session on Visual Studio where the command line arguments are again –d:\DATA +s –H. The top row shows a command window for the clients, the debugger’s server window, a glimpse of a file manager showing the empty database folder.

We first create database A with the SQL statements on slide 169.
Now create the database B using the statements on slide 169.

The image shows table “Log$” has just also happened, and we can see the RESTView at position 419.

Stop the server: we do this for reproducibility, as compiled object uids are different during the session in which objects are created.

Restart the server.

Find Context.cx in Level4 and double-click it. We are going to place breakpoints in the Context.Review() method.
The breakpoint for the start of Context.Review() is at line 717 in Context.cs.
The breakpoint at the end of the Context.Review() method is at line 843 of Context.cs.
When we reopen database B, it is loaded into the server, and the compilation process for the view calls Context.Review()
Click Continue twice to get the normal SQL> prompt.

[177 @ 1:07:46]
We are going to select some items from V.

[178 @ 1:07:55]

The given select statement is `select e,f,m from V where e=1`.

(Note the position of e in this statement: it occurs at position 8, so and the equals sign of the where condition is at position 28. We will see these numbers later.)

RowSet Review for the select statement stops at the breakpoint we set.
We are going to review the given RowSet $r$ and its sources using the Watch window. It is a SelectRowSet.

[179 @ 1:08:30]
We see the source is a rowSet #14. To examine other rowSets we need to look at the data BTree in Context.

[180 @ 1:08:42]

Begin to expand data. RowSet #14 will be near the middle.
Expand node root.slots[1].
We see both SelectRowSet #7 and TableExpRowSet #14 are here. Let's use a Notepad to list #7 and its source #14.

Scroll up to collect the value of SelectRowSet #7 for our Notepad:
Then expand slot [3] here to get the value of TableExpRowSet #14.
The source of TableExpRowSet #14 is rowSet %10, which we can see at the bottom of the picture. Expand and copy it too.

[184 @ 1:09:37]
It is a JoinRowSet, with sources %18 and %15. These will both be in the gtr node of the data tree.

[185 @ 1:09:50]

%18 is an OrderedRowSet, with source %9.
Now do %15.

[186 @ 1:10:01]
It is time for a short cut here: %18 has source %9 which we can see is a RestRowSet, and %15 has source %12, which is a SelectedRowSet. So let’s pick them up later, at the end of the Review. Click Continue.

This is the breakpoint at the end of Review. Return to the Watch window.
Visual Studio reports changes to all of these RowSets. It will also be out of date, so repeat the above extraction from the data tree, starting with the modified SelectRowSet #7 (scroll down for it).

We see the changed version: copy its value to the Notepad and continue with its sources etc as we did before.
We see that a number of rowsets that were there before have simply been removed: there is no need for %9 or %12 and the OrderedRowSets %18 and %15 have been replaced by RestRowSet %18 and SelectedRowSet %15.

Importantly, the where condition e=1 (where (#28) matches (#8=1)) has been analysed.

The Review process has noticed that the where-condition collapses both sides of the join to a single row, so that the ordered_rowsets are not required.

And the RestRowSet now has the matches condition #8=1 which will be included in the HttpRequest to the remote database.

Click Continue.

[191 @ 1:11:59]
During the HttpService, our breakpoint gets hit again. Click Continue twice (before the timeout!),

[192 @ 1:12:09]

We get the results of course, but importantly we see that only one row was requested from the remote database. (More generally, aggregations can also result in only one row being returned.)

This concludes the demonstration of RowSet Review.