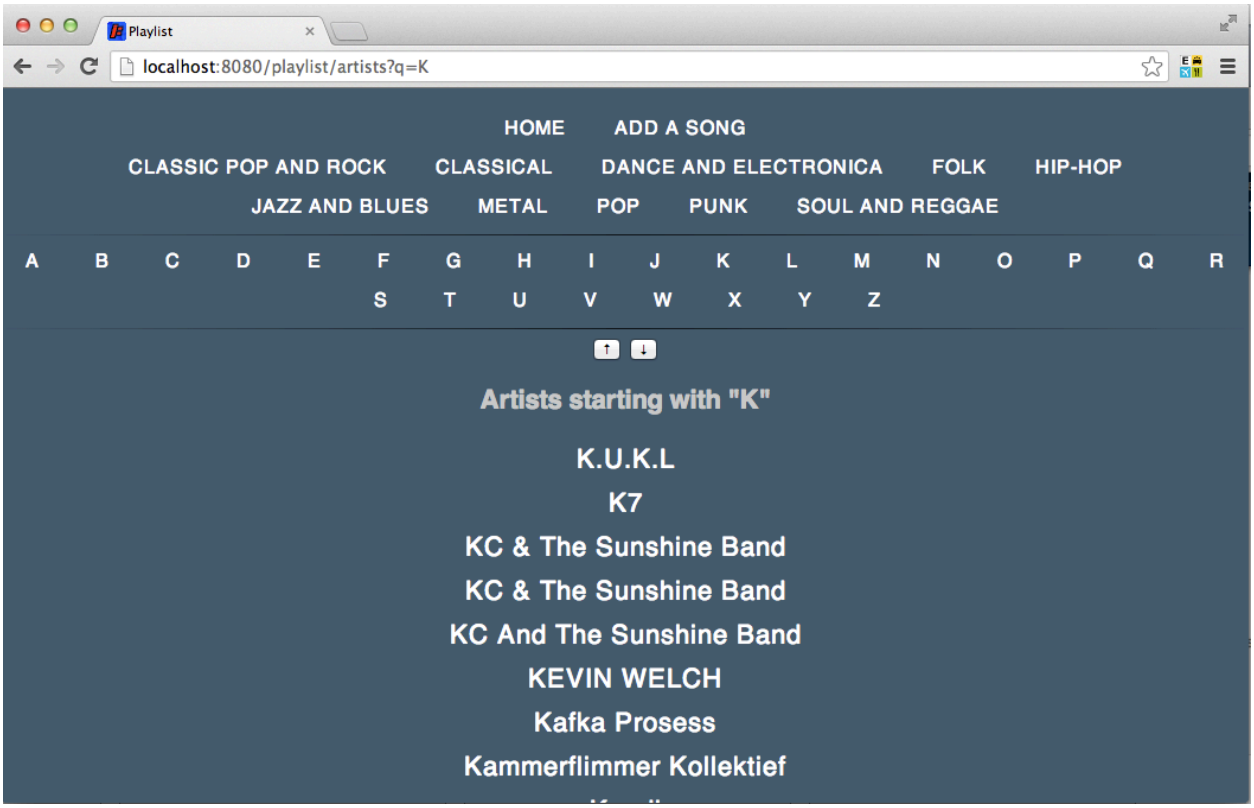
Session 2

OVERVIEW

In this set of exercises we will start building out our playlist application. The application has several distinct components including the Song Database, User Registration, users' Playlist Pages, and Statistics. This module focuses on implementing the Song Database. It consists of a list of Artists and their Tracks. You can browse Artists by the first letter of their name, drill down to list their songs, and browse songs by genre.



Our Data Model

We have a table of Artists grouped by the first letter of their name and a table of Songs organized by artist name. In subsequent sessions, we will develop tables and code to handle users and playlists. We will set up our tables and then extend the data model as necessary.

The Code

The code to the playlist application is organized in a model / view / controller architecture. The following is a list of the directories and files in the project to give you an overview of all of the files in the package. It's not necessary to understand the structure of the program to complete the exercises, but it's handy if you would like to understand how the application works.

Project Directory Structure

Files	Purpose
src/main/java	Java sources
src/main/webapp	Web files like JSP pages
src/test/java	Our JUnit Tests

Copyright © 2013 DataStax, Inc.

pom.xml	A declaration of the project which informs tools like Maven and Eclipse how to build the project. This file also includes a list of the dependent libraries including the DataStax Java Driver
scripts	CQL scripts and CSV files
target	If you build the project with Maven, this is where the artifacts wind up.

Java Code (model, controller, and the Jetty mainline);

Files	Purpose
src/main/java/playlist/controller (Java package playlist.controller)	Controller Classes which decide how to process Web Requests
HomeServlet.java	Render the home page
ArtistServlet.java	Receive a letter, and render the artists starting with that letter
TrackServlet.java	Receive either an artist or genre, and render those tracks
src/main/java/playlist/model (Java package playlist.model)	The model files that interact with Cassandra
CassandraData.java	The parent model class which stores our Session object
CassandraInfo.java	Return the Cassandra version and cluster name.
ArtistsDAO.java	Data Access Object (DAO) that reads from the artist table(s)
TracksDAO.java	DAO that reads and inserts track information. A instance of a TracksDAO *holds* the information of a track
src/main/java (Default Java Package)	Classes not in any package
StartJetty.java	The main class to start the program

Web Files (View in MVC)

src/main/webapp	All of the Web-related files for rendering the pages
web-inf/web.xml	Deployment descriptor. This tells a server such as Jetty or Tomcat how to map the URLs in the website to the controller classes in our application.
css/playlist.css	The stylesheet for formatting the web pages
home.jsp	The home page
index.jsp	Redirects to the /home URL
notfound.jsp	A page that comes up when an un-implemented page is requested
artists.jsp	Page that shows the list of artists
tracks.jsp	Page that shows the list of tracks

Copyright © 2013 DataStax, Inc.

add_track.jsp	Form for adding a track to the system
lists.jspf	The lists of letters and genres
trackheader.jspf	The top portion of the tracks and artists pages.

SETUP

Download and unpack the Session 2 archive (session2.zip or session2.tar.gz). Load the project into Eclipse in a same way we did last session in exercise 4 (**Hint**: if working in the same Eclipse project folder from Session 1, consider first deleting Session 1's project, so there is no conflict in project naming for "playlist"). For the first 2 exercises, we will turn away from Eclipse, and use Cassandra utilities.

EXERCISE #1: Create a Keyspace and Tables for the Playlist Application

1. Start Cassandra and connect using CQL shell.

2. Create a keyspace called "playlist". We need to use a replication factor of 1 since we only have a single node system.

```
create KEYSPACE playlist WITH replication = {'class':
'SimpleStrategy', 'replication_factor': 1 };
```

3. Make this the default keyspace for your session;

```
use playlist;
```

EXERCISE #2: Create and Load the Artist Table

4. Recall we need a table that we can search on by the first letter of an artist's name. We'll create a table with just 2 columns, the first letter, and the artist name. The artist name is unique, so we need to include that in our primary key. In addition, we need in to include the first letter. Create the following table:

```
create table artists_by_first_letter (first_letter text,
artist text, primary key (first_letter, artist));
```

   The compound primary key will allow us to query by the first letter.

5. Now lets load the data. In the archive file which you extracted, there is a delimited file of artists and first letter, and the delimiter is the vertical bar "|" character. You can find it in the 'scripts' directory of the archive. Load this file

Copyright © 2013 DataStax, Inc.

into the new table with the COPY command. Be sure to put the correct path to the file.

```
copy artists_by_first_letter (first_letter, artist) from
'artists.csv' WITH DELIMITER = '|';
```

6. Run a couple of queries on the table to verify that the data has loaded:

```
select * from artists_by_first_letter LIMIT 5;
```

You should see output something like this:

```
 first_letter | artist
--------------+---------------------------------
            C |                   C.W. Stoneking
            C |                             CH2K
            C | CHARLIE HUNTER WITH LEON PARKER
            C |                     Calvin Harris
            C |                           Camané
```

Note that the first_letter field is not returned in alphabetical order, but the artist is, as artist is a cluster column within the primary key.

EXERCISE #3: Start the Application in Eclipse and View the Artists

This week, we will be making modifications to the project code, and therefore we have not shipped compiled code. To run the project, we will be doing it in Eclipse.

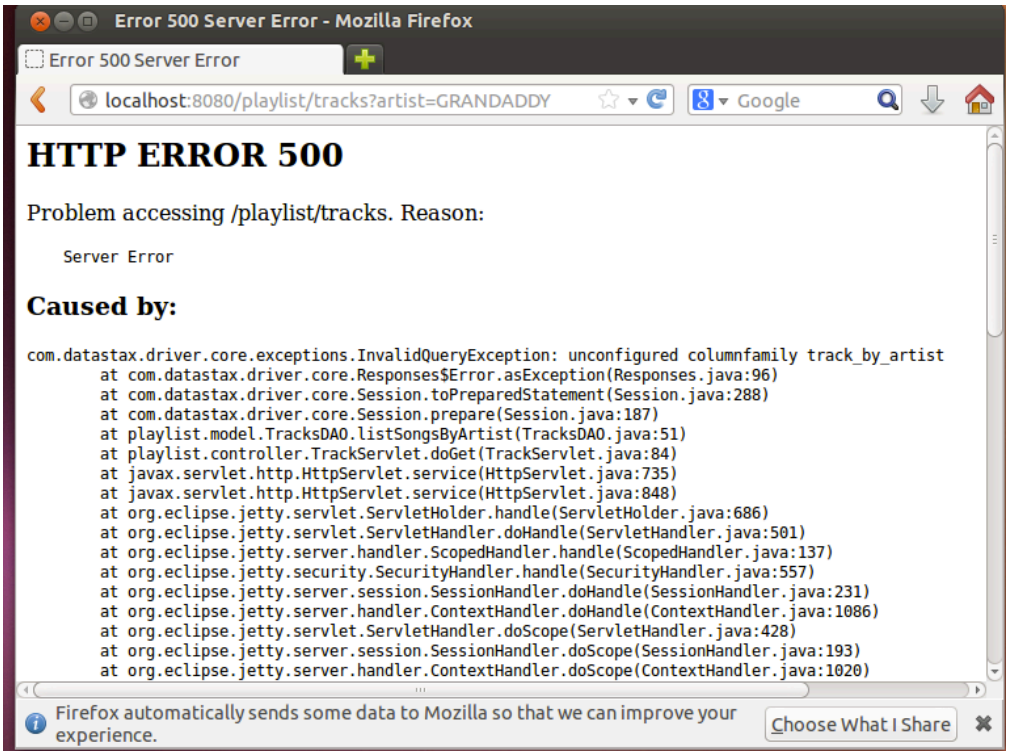
7. Start Eclipse, and import the project as a Maven project in the same way that you have done in Session 1. Run or Debug the application as a Java application, and choose the StartJetty class as the main class.

8. If you restart the application, Eclipse remembers your run configuration. You can restart the application by clicking the run or debug buttons in the toolbar:

9. Visit the application at http://localhost:8080/playlist. Click "VISIT THE SONG DATABASE" and notice that the pages exist this week. Click any letter, and you should see the artists that start with that letter.

10. Drill down on one of the songs, and notice that the application throws an InvalidQueryException as depicted below. We haven't created the table of track_by_artist yet to let us search the tracks organized by artist.

Copyright © 2013 DataStax, Inc.



11. If you click on any of the genre links, you should get no songs. You will add that functionality in a later exercise.

EXERCISE #4: Create and load the track_by_artist Table

12. Create a table called track_by_artist. It will have the following columns:

Column Name	Type
track	text
artist	text
track_id	UUID
track_length_in_seconds	int
genre	text
music_file	text

Choose an appropriate primary key so that we can query on the list of songs by artist (**HINT:** read http://www.datastax.com/documentation/cql/3.0/webhelp/index.html#cql/ddl/ddl_anatomy_table_c.html#concept_ds_cz4_lmy_zj).

Copyright © 2013 DataStax, Inc.

13. Import the data from the songs.csv file in the same way that was done for the artists_by_first_letter table. The CQL shell command is show below. We use the "HEADER=true" option as the first line of the songs.csv file contains a header line, and we can Cassandra to ignore it.

```
copy track_by_artist (track_id, genre, artist, track,
track_length_in_seconds, music_file) FROM 'songs.csv' WITH
DELIMITER = '|'  AND HEADER=true;
```

14. Now re-visit the Song Database web page, and see that we can now list the songs for a given artist.

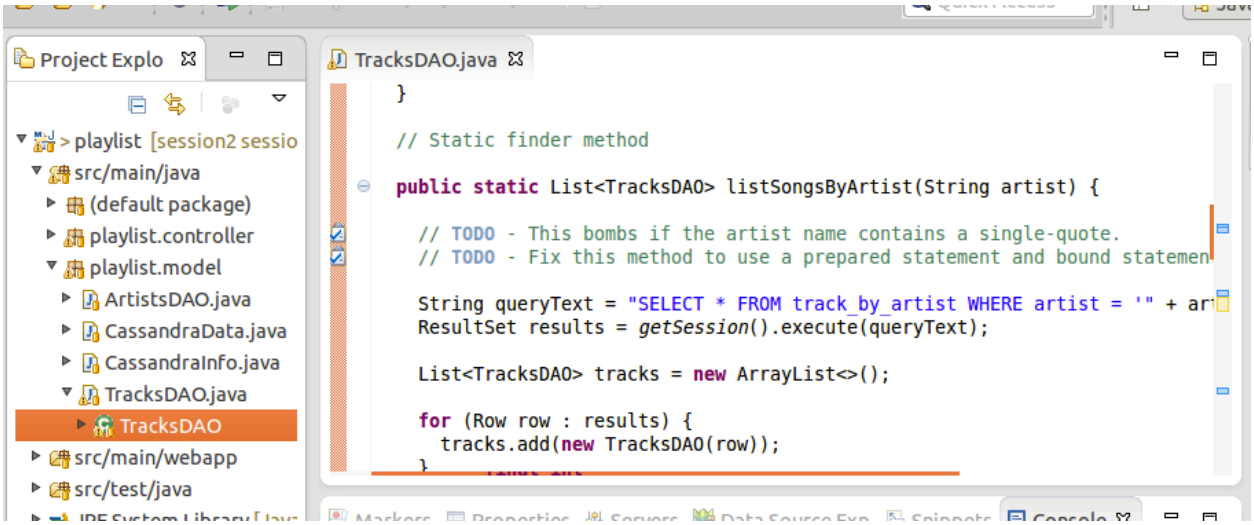
EXERCISE #5: Implement a Prepared Select Statement

15. Visit the Song Database. Click on "P". A few lines down, choose "Pagan's Mind" (The one with the single-quote). This causes a CQL exception (apologies for the small font):

```
com.datastax.driver.core.exceptions.SyntaxError: line 1:81 mismatched character '<EOF>' expecting '''    at
com.datastax.driver.core.exceptions.SyntaxError.copy(SyntaxError.java:35)       at
com.datastax.driver.core.ResultSetFuture.extractCauseFromExecutionException(ResultSetFuture.java:271)    at
com.datastax.driver.core.ResultSetFuture.getUninterruptibly(ResultSetFuture.java:187)       at
com.datastax.driver.core.Session.execute(Session.java:126)  at com.datastax.driver.core.Session.execute(Session.java:77)
 at playlist.model.TracksDAO.listSongsByArtist(TracksDAO.java:54)
```

Notice the last line of the exception shows that we were in a class called TracksDAO in the method listSongsByArtist, before we made a call into the DataStax Java Driver. This is the model class that is responsible for retrieving track information from Cassandra.

Let's visit that file in Eclipse:



Copyright © 2013 DataStax, Inc.

Notice that we simply build the query as a string:

```
String queryText = "SELECT * FROM track_by_artist WHERE
artist = '" + artist + "'";

ResultSet results = getSession().execute(queryText);
```

If artist contains a single-quote, we will build a mal-formed CQL statement with un-balanced single-quotes. It will look something like this:

```
SELECT * FROM track_by_artist WHERE artist = 'Steve's
Best';
```

We then execute the statement directly with the execute() method.

16. Replace those two lines with code that will use a prepared statement, and bind the artist variable (**Hint**: You will use create a PreparedStatement object as well as a BoundStatement object; for a reference to the Java Driver API, please visit: http://www.datastax.com/documentation/developer/java-driver/1.0/webhelp/index.html#common/drivers/reference/apiReference_g.html).

17. Stop and start the application, and validate that you can click on artists that have a single-quote in their name.

EXERCISE #6: Querying By Genre

18. As mentioned earlier, if you click on any of the genres in the Song Database, the page turns out blank. In this exercise, implement the missing code in the method TracksDAO.listSongsByGenre(). Think about how Cassandra can and can not search data. Create any additional tables as required, and load the necessary song data.

19. Validate that you can now list all of the songs for a particular genre in the application.

EXERCISE #7: Implement Insert Track

In this exercise, we'll see how UUIDs are used as a unique id for the tracks in the next session.

Each song has a unique track_id which is of type UUID. Each time we create a new track, we need to generate a new UUID for the track_id. This is done when we mint a brand new TracksDAO object in its principal constructor (the method that sets up a new TracksDAO object):

Copyright © 2013 DataStax, Inc.

```
public TracksDAO(String artist, String track, String genre,
String music_file, int track_length_in_seconds)
```

We generate the track_id in Java as follows:

```
    this.track_id = UUID.randomUUID();
```

Note: There is one additional constructor that creates an object from a row found in Cassandra. We do not need to generate a UUID in that one, as it reads it in from the database.

20. Click the "Add a Song" link and add a new song. Click on the genre link for the genre of the newly added song, and validate that it's in the list. If it's missing, examine the method TracksDAO.add(), and make any appropriate changes.

21. Add a new song, and re-test the application.

Copyright © 2013 DataStax, Inc.