Building Scalable Apps featuring CockroachDB Presenter's Guide

Objective: Prepare yourself for this workshop, “Building Scalable Apps Using CockroachDB”. In this presentation participants will learn about what a database is, impacts of database design on app performance, and how to build a scalable database.

About This Workshop

In this workshop you will teach participants about what a database is and how database design impacts the scalability of an app. Participants will then build a scalable web app using CockroachDB and test the app’s ability to scale.

After this workshop participants will better understand how the design of an app impacts its scalability and become more adept app designers as they can better plan and design products for long-term use.

How to Prepare

The best way to prepare for this workshop is by familiarizing yourself with the slides and this presenter’s guide. During the workshop you’ll be able to use this guide as a reference of what to say and how to direct participants.

Before this workshop you should:

- Have a Glitch account
- Read over the Background material on different kinds of databases
- Try the demo application: http://mlhlocal.host/cockroachdb-demo
- Attempt this workshop 2-3 times so that you feel confident and familiar with the material and equipped to help workshop participants.
- Make sure to customize any slides in the presentation that ask you to do so! Don't change any of the information on the other slides.

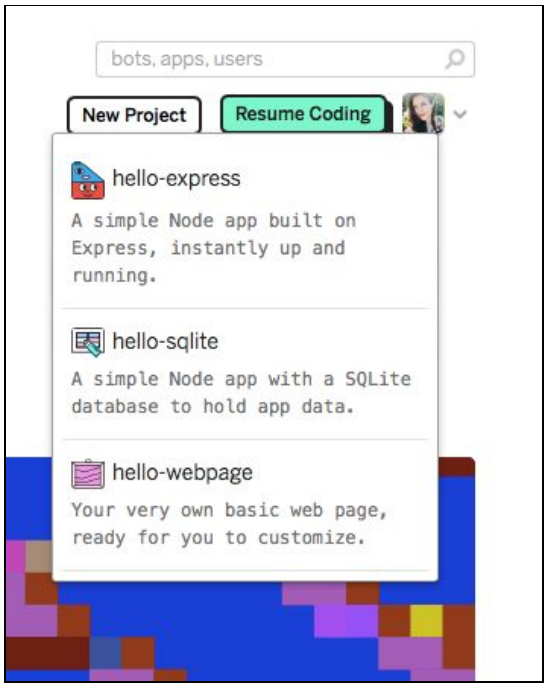
Setting Up Glitch

To better understand the capabilities of CockroachDB, participants will build a simple website that displays a different Star Trek quote every five seconds by retrieving those quotes from a database managed by CockroachDB. To see the final version of the website, visit: http://mlhlocal.host/cockroachdb-demo

To start, we need to set up Glitch. Go to www.glitch.com. Glitch is a combination of an app development community and a hosting service.

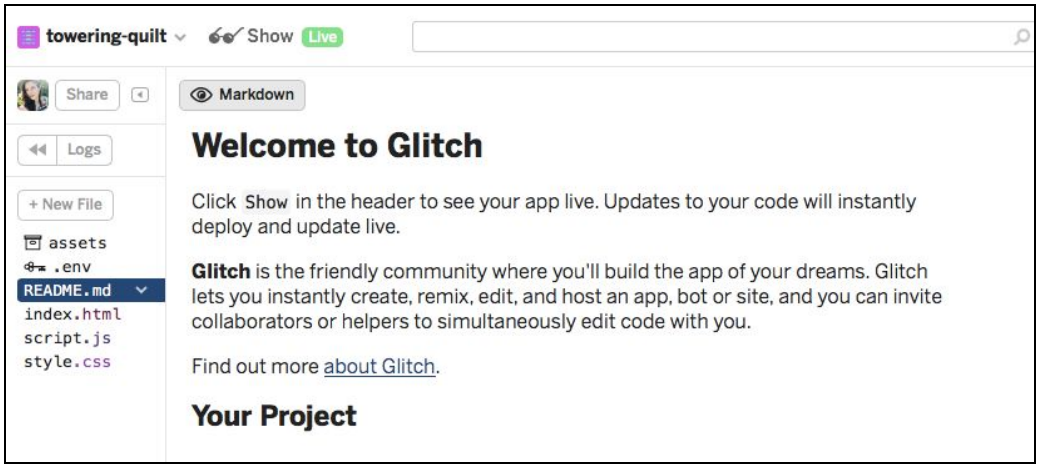
Participants can sign in using their GitHub or Facebook accounts.

If you want to further explore Glitch, you can set up a sample web page.



Start a new webpage by clicking “New Project” then navigating down to “hello-webpage”.

You will be brought to a new Project page.



To see what the project looks like to your viewers, select the Button “Show Live”.

You will see a basic web page that says “Hi there!”.

Introduction to Databases

Objective: Help participants to understand the fundamental ideas of a database and its role in an application, as well as different kinds of database designs.

Activity: Put participants in the shoes of potential app designers. They have to imagine they need to balance between developing an application *quickly,* one that will run *fast,* provide reliable *access* to users, not use too much *processing power*, and have *consistent* data between different devices, and can *scale* to allow for many users. Sometimes building to optimize one of these qualities can hurt performance in another area. A good app developer understands the business needs of the overall company and their customer and creates the kind of database structure that meets these goals.

CSV (formatted text files) works best with flat data structures and small applications. It demands less technical skill (making it an accessible language for early stage developers), and can be accessed by most applications. It is not recommended for large-scale or complex data projects.

SQL stands for Structured Query Language and is the foundational language for more complex database management systems. SQL lets you submit queries, update, delete, and retrieve data, and is supported by many database products. SQL however sometimes has a hard time running complex operations on data retrieved.

PostgreSQL is frequently used for web databases and handles both structured and unstructured data. It can handle large volumes of data and concurrent users, and can also scale. However, speed sometimes suffers when an application has to read through such high volumes of data.

Knowing the pros and cons of each of these types of formatted text files and databases, now it's time to ask:

- Which would be better to use for a user's settings on a phone app?
- Which would be better to use for an app where a user can create and search through music collections?
- Which would be better to use for a large online forum with many user profiles, friends lists, and many interactions between users (such as Tumblr or Discourse)?

When participants have finished the quiz and know the answers (provided on the slides) encourage them to share projects they are working on, and what their largest concerns are (is it speed? Reliability? Scale?) Having a personal use for the knowledge presented in the workshop can often help participants feel more connected with the activities.

Databases

A database is an organized collection of data. This data can be anything, from text, to numbers, documents, etc. Databases allow web apps to have a **preserved state.** A preserved state means that when a user closes a web page, their information is still stored. Creating an account allows a web app to keep all of the text, documents, and pictures posted to the web page. Imagine if you had to recreate your Facebook profile every time that you logged in. Having a preserved state makes it so you do not need to do that.

A **database management system** allows application designers to manage how a database is structured. It specifically controls three things: the data, the database engine, and the database schema. The **data** we already know can include almost anything, including

pictures, text, and files. The **database engine** controls how data can be locked, accessed, and modified. A **database schema** defines the database's logical structure, or how all the different components fit together.

People interact with a database management system (DBMS) through **querying,** or sending commands to the database to retrieve information.

Types of Databases

There are four types of database management systems: relational, NoSQL (non-relational SQL), Key/Value, and NewSQL. Each of these types have trade-offs in terms of Consistency, Scale, and Speed. NewSQL, and CockroachDB, combines both Consistency and Scale.

In **Relational Databases**, data is structured in Tables (rows & columns) and is closest out of the database types to Excel/spreadsheets. Every row in a table has the same columns. SQL is used to query data from these types of databases.

In **NoSQL databases**, data is unstructured and stored in Collections. Every entry can have different columns. Javascript Object Notation (JSON) is the language used to query data.

Both of these types of databases have trade offs. Relational Databases that are run on a single-server (one computer) have *access* as a potential point of failure. If that computer cannot be reached, the data cannot be accessed. As a company's app reaches more users, it becomes more important to have data continuously accessible. Alternatively, Relational Databases that run on multiple servers have to deal with **sharding.** Sharding means that the data is stored on numerous computers. Sometimes this can make sense, such as an international company breaking up its databases into American and European siloes. However, if a person needs to search through both American and European sections, this can take longer than if the databases were not split apart. As well, more databases mean more potential for bugs to be created because more code is needed to set up and maintain these databases.

NoSQL databases have the hurdle of *consistency* because they do not have the same type of properties in their database transactions as SQL databases. SQL databases have **ACID transactions** which help to create consistency of data in database transactions. ACID transactions ensure that data from databases is sent with *atomicity*, ensuring that even for transactions that might have multiple statements, they either succeed or fail completely, even if only one statements was able/not able to be run. ACID transactions also necessitate *consistency,* guaranteeing that the information that is entered in a database must be valid according to all of the defined rules for data inputs. Data is also *isolated,* meaning that even if data is sent concurrently, it acts as if it is sent sequentially. SQL databases also are

durable, as once the transaction is committed, it will remain committed even if there is a system failure.

NoSQL databases do not have these same checks, but work on a system of **eventual consistency.** This means that data from one server might be different from data on another server, and while over time the two sets of data would **converge** into one set, there would be instances of people working from two different sets of data, and when the data converges back together, some could be lost. NoSQL also has fewer standards between different products, meaning that one company's database management system might require different commands and syntax to access data than another's.

Introducing NewSQL

NewSQL is a class of database management systems that can scale like NoSQL, but is as consistent as a traditional relational database. NewSQL is specifically useful for financial organizations where consistency is paramount, but users also want fast response times. NewSQL achieves this by structuring data in **hash indexes** instead of traditional rows and tables.

A hash index is composed of keys and values. A hash function assigns an index number (0,1,2, etc.) in an array (a list) of all of the data. Hash indexes make it so if people have a specific value that they know they are searching for, they can easily find it in a database. NewSQL systems work best where users need to look up specific information (for example, the money in one person's account), and not doing joins (combining whole columns of different databases).

CockroachDB

CockroachDB is an open-source software project that is designed to store copies of data in multiple locations in a way that can scale *quickly* while maintaining *consistency*. CockroachDB was developed by Cockroach Labs, a company focused on building new kinds of database management systems for businesses.

The aim of CockroachDB is to create a database management system that is *both* scalable and consistent. CockroachDB offers an advantage over traditional SQL databases by being more *reliable,* harder to take down since data is spread across many servers. CockroachDB offers an advantage over NoSQL databases because it maintains greater *consistency* of data even if some of the servers go offline.

CockroachDB accomplishes scaling and failover by replicating data and connecting it in clusters. A **cluster** acts like a single application even though it can contain more than one database. A cluster is composed of **nodes**, or many instances of the same software running

simultaneously. Imagine having multiple tabs of the same website running. If you close one tab, the other one is still running. Just like this example, clustering makes it so if one node goes down, the others will still run.

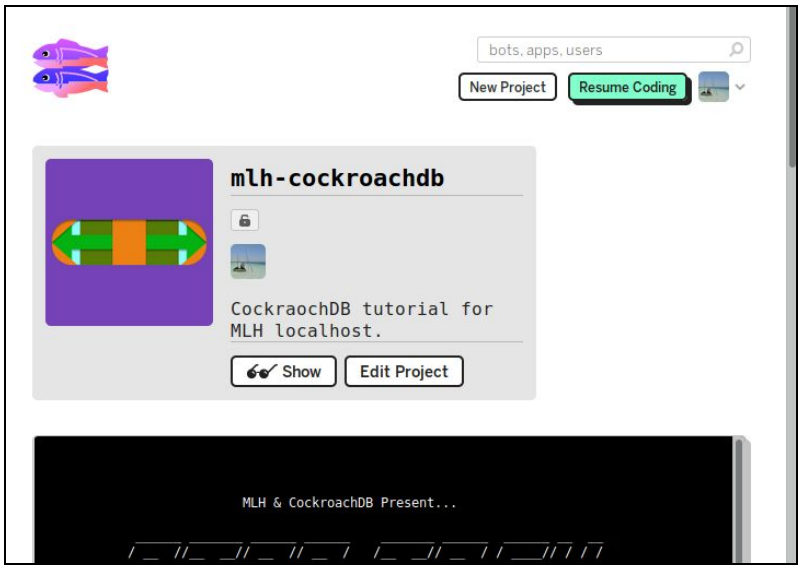
CockroachDB is *scalable* because it creates nodes by splitting data into **ranges**, or groups of data. These groups of data are replicated and distributed across many nodes. This way, if one node goes down, the data is still accessible from another node.

CockroachDB enables *consistency* through the **Gossip Protocol**. The Gossip Protocol means that each node tells the other nodes new data in pairs. The number of nodes with new data continues to double. This is like how new rumors spread.

But, unlike rumors where data remains unverified, the information spread through the Gossip Protocol is then checked for consistency using the **RAFT consistency algorithm.** The RAFT consistency algorithm starts with the nodes electing a "leader" that has the most up-to-date information. The "leader" handles how information is sent to "followers." The "leader" makes sure that incorrect information never surfaces, even if this means declining a request. You can read more about the Raft consensus algorithm here or here.

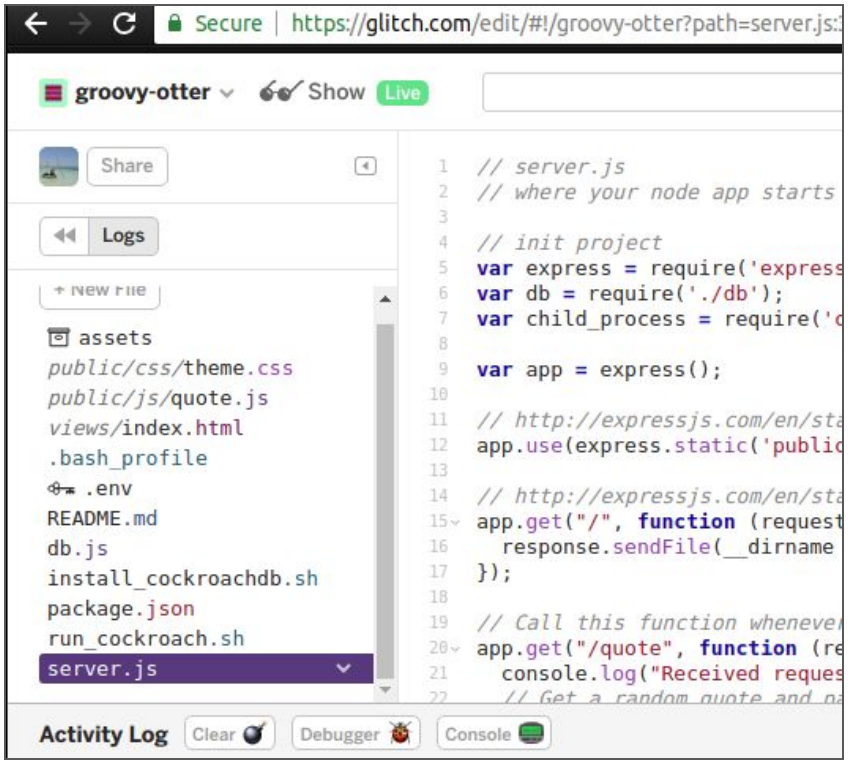
To start building your own version of this website, visit:
https://glitch.com/~mlh-localhost-cockroachdb

Scroll down past the preview:

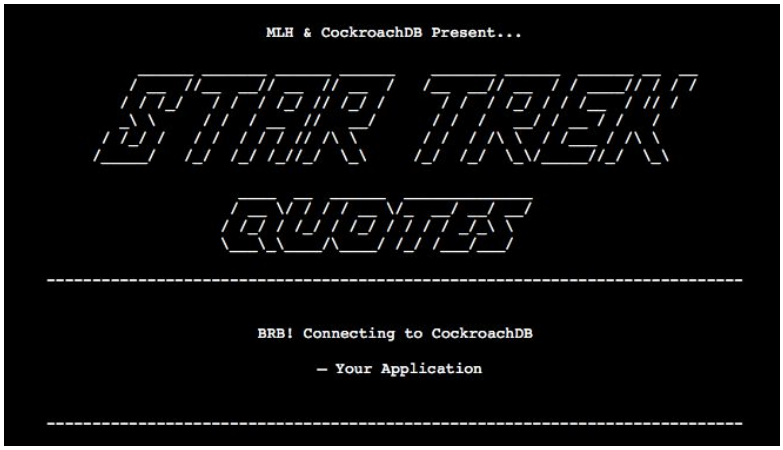


Click the button that says "Remix Your Own":

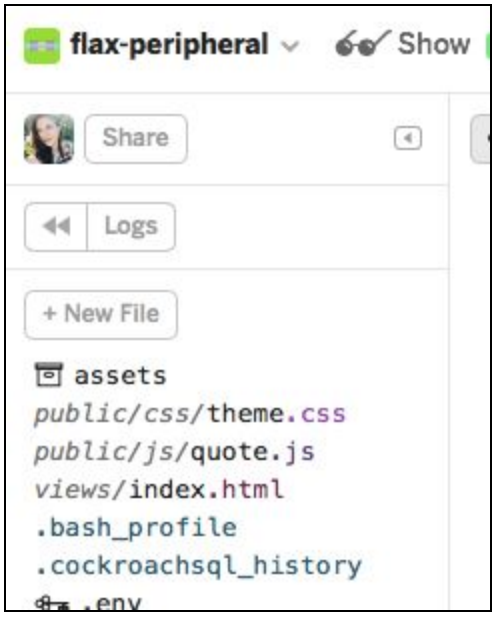
To make it so you can see the impact of what you are building, participants can run a live Demo of the website. Click on the button that says “Show Live”:



A version of the website will open. Instead of Star Trek quotes displayed, it will say “BRB! Connecting to CockroachDB”.

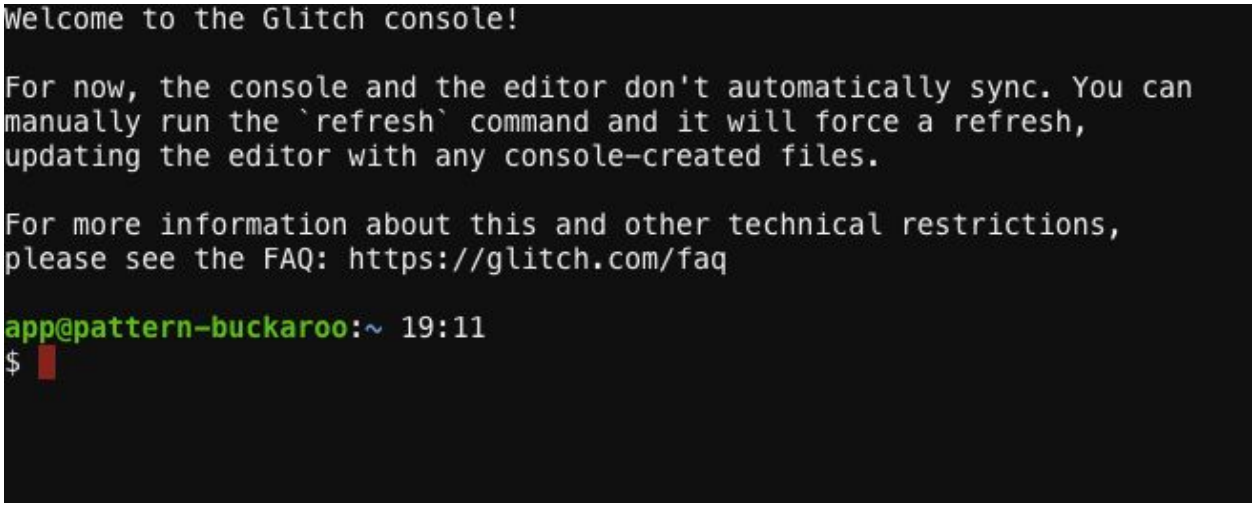


On the left hand side there is a “Logs” button. Click it!



At the bottom of the page the Activity Log will appear.

Next, click the “Console” button. The Console will open in a new tab.



Console Vs. File Browser

There are two different ways for a human to “talk” to a computer. One way is through a **File Browser**. This is the typical way that people interact with their computers, by navigating through and selecting visual representations of files and documents. A **console** is an older way of interacting with the computer, where commands are typed in and data is sent back via text.

Installing CockroachDB

On the Glitch console, type the command `“sh install_cockroachdb.sh”` and press Enter. This script installs CockroachDB.

```
$ sh install_cockroachdb.sh
Installing CockroachDB
--2018-07-09 21:01:58--  https://binaries.cockroachdb.com/cockroach-latest.linux-amd64.tgz
Resolving binaries.cockroachdb.com (binaries.cockroachdb.com)... 52.84.122.105, 52.84.122.215, 52.84.122.60, ...
Connecting to binaries.cockroachdb.com (binaries.cockroachdb.com)|52.84.122.105|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 20665898 (20M) [binary/octet-stream]
Saving to: 'cockroach-latest.linux-amd64.tgz'
```

Starting Your First Instance

Now that CockroachDB is running in your environment, the next step is to start a Cluster with one Node. We're going to do this within the "data" folder of CockroachDB. You enter the "data" folder of CockroachDB by typing in `"cd ~/.data"` into your console. Next, type `"cockroach start --insecure --host=localhost --background"` into the console. Each "word" of this command tells the computer to do a different thing. `"cockroach start"` starts a new node (an instance of cockroach). `"--insecure"` makes it so any user can access this database and change it. `"--host=localhost"` tells the CockroachDB API that the website is being hosted on your computer. "--background" sets the node running in the background while you can give the computer other commands. **If you close the window, you will need to run all commands again.**

Create Sample Data

The next step is to create the database to put our Star Trek quotes in. The database will be named "startrek" and have two tables called "quotes" and "episodes." Type in "cockroach gen example-data startrek | cockroach sql --insecure" into the console and hit "Enter." Not working? If you get back a message that says "Segmentation Fault," you will need to Refresh your Console and re-enter your commands.

Let's break down what each part of this command does. "cockroach gen example-data" Generates example data. The `"|"` called a "pipe" takes the output from the command on the left and uses it as the input to the command on the right. In this situation, it takes the sample data and uses it for the next command. The next command enters the SQL shell to query the database. "Cockroach sql --insecure" Then allows you to query the database you created.

Understanding the Database

Next we will dive into the Star Trek database. In your terminal, type in `"SHOW DATABASES;"`. Entering this command will show all the databases you can pull data from. To enter specifically the Stark Trek database, you then type `"SHOW TABLES FROM startrek;"`. There are two tables in the database, "episodes" and "quotes."

To see what episodes are contained in the database, we can run the command `“SELECT * FROM startrek.episodes;”`. This sends a GET request to CockroachDB’s server that asks for a table listing all of the quotes that the episode pulls from.

Now participants will learn how to retrieve a specific quote from the database. The full command is: `“SELECT * FROM startrek.quotes ORDER BY RANDOM () LIMIT 1;”`

`“SELECT”` tells your computer to read through all of the Star Trek database.

`“* FROM startrek.quotes”` tells your computer to pull from the “quotes” table in the “Star Trek” database.

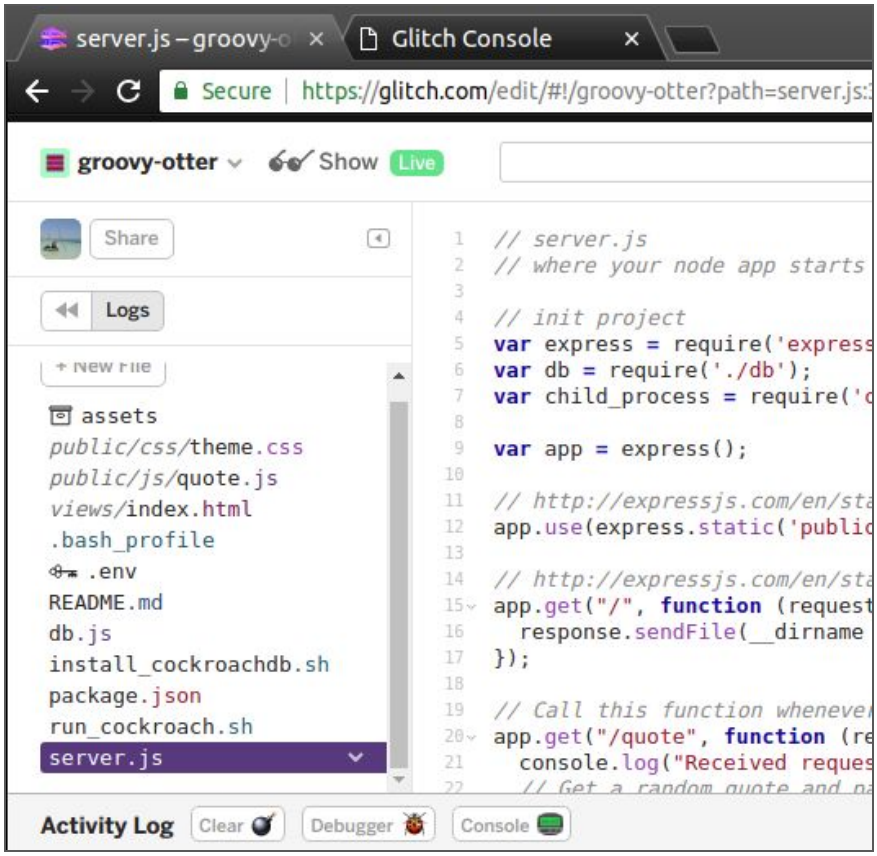
`“ORDER BY RANDOM ()”` orders the results randomly.

`“LIMIT 1;”` limits the results to one.

Enter the entire command. You should get back a table that includes a quote, the character that said the quote, and the episode the quote was from.

Connecting Data to the Application

Navigate back to the Glitch Console. In the file tree on the left side of the file browser tab, click **server.js.**



Server.js will open.

Lines 5-17 and 31-37 provide the boilerplate code in basic HTML. The last line is commented out so that the server does not run automatically yet.

Enter this code on line 11:

```
// Call this function whenever someone requests the /quote path
app.get('/quote', function(req, res) {
  // Get a random quote and pass it to the browser.
  client.query('SELECT * FROM quotes ORDER BY RANDOM() LIMIT 1')
    .then(data => res.send(data.rows[0]))
    .catch(err => res.send({ error: "BRB! Connecting to CockroachDB" }))
})
```

Now whenever someone goes to “localhost:8080/quote” this code will run on the backend. It runs a query to the backend to get a random quote. If successful, the server sends back the data from a given row. If unsuccessful, the server will return the error message “BRB!”

Next, we will connect to CockroachDB in db.js.

```
01 // Connect to the startrek Database on a node in our CockroachDB cluster
02 var pg = require('pg');
03 var config = { user: 'root', database: 'startrek', port: 26257 };
04 var db = new pg.Pool(config);
05
06 // Log when we connect to the CockroachDB node
07 db.on('connect', function(client) { console.log("Connected to CockroachDB.") })
08
09 // Log any errors we encounter
10 db.on('error', function(e) { console.error("Err connecting to CockroachDB.") })
11
12 // Export the database connection so anyone can use it
13 module.exports = db;
```

Line 2, "`var pg = require('pg');`" requires "postgres", an SQL client. Line 3 specifies that the computer should be accessing information from the Stark Trek database. Line 7 logs "Connected to CockroachDB" when a successful connection is established with the database. Line 10 logs "Err connecting to CockroachDB" if a connection cannot be made. "Module.exports = db" allows code previously written to be used by other files.

To see the final product, go back to the tab with the app running and press the "Show Live" button. Now you should be able to see the website! Sit back and you should see a new quote appear!

Creating More Nodes

Now we are going to add greater reliability to your application by adding a second node to our cluster. Each node in CockroachDB stores a compressed version of all of the data. This means adding a second instance running the same application.

Navigate back to your terminal and Enter "`cockroach start --insecure --host=localhost --background --join=localhost:26257 --store=mlh-node2 --port=26258 --http-port=8081`" into your console. Again CockroachDB is started, given the "`insecure`" instruction that lets you query the database, say that localhost is the host, and set the database to run in the "`background`" while you enter more commands.

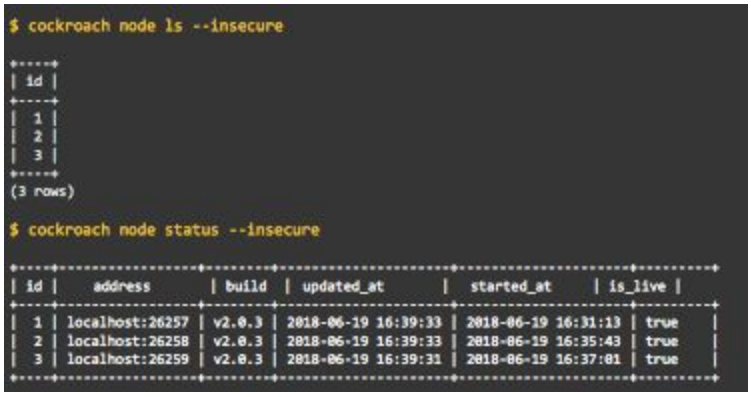
"`--join=localhost:26257`" assigns a port to one instance of CockroachDB. "`--store=mlh-node2`" gives a name to this new instance, "`--port=26258`" creates another instance of CockroachDB, while "`--http-port=8081`" assigns both of these ports to the same cluster.

To add a Third Node, enter the same instruction again, but this time change the node number from node2 to node 3 and increment the port number by 1. The cluster number should remain the same. Now with a third node, even if one of the clusters goes down, it does not bring down your entire application.

The code should look like this:

“cockroach start --insecure --host=localhost --background --join=localhost:26257 --store=mlh-node3 --port=26259 --http-port=8082”

To check whether all three nodes have been created and are running properly, type in “cockroach node ls --insecure” into your console. The “ls” command lists the available nodes and their statuses.

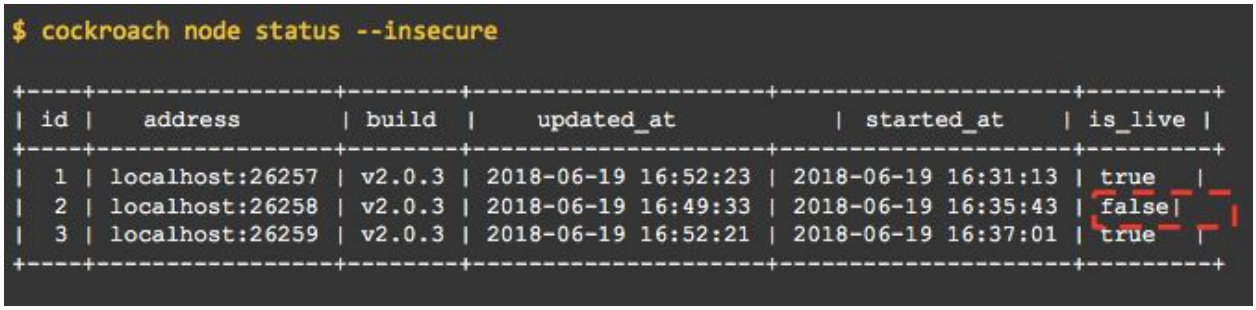


Testing Fault Tolerance

Fault Tolerance is the ability for a system to continue operating even in the event that part of the system failing. When a CockroachDB node goes offline and then comes back online, it gets updated with changes to its data. We are going to test out our system’s fault tolerance by taking a node offline and see if the system still runs. While one of our nodes is offline, we will change some of the data and see how the system restores itself. To do so, type in “cockroach quit --insecure --port=26258”. This quits the node 26258.

To see whether the node is still running, type “$ cockroach node status --insecure” and run it into the Console.

You should see one of the nodes is not Live.



Next we will test what happens when we change some data and bring the port back online.

Type in `“UPDATE startrek.quotes SET characters = 'your_name_here';”` (but you should replace `‘your_name_here’` with your name). This changes the attribution of the Star Trek quotes to your name. Not working? Check whether you have used single or double quotes around the `‘your_name_here’` command.

Refresh the app to see if your name now appears below the quotes.

Now bring all the Nodes back online to test how the network deals with aligning data from a node that was down. Type in `“$ cockroach start --insecure --host=localhost --background --join=localhost:26257 --store=mlh-node2 --port=26258 --http-port=8081”` into your Console.

Next, connect to the Node that was initially taken offline.

```
$ cockroach sql --insecure --port=26258


# Welcome to the cockroach SQL interface.
# All statements must be terminated by a semicolon.
# To exit: CTRL + D.

root@:26258/> SELECT * FROM startrek.quotes ORDER BY RANDOM() LIMIT 1;
+--------------------------------------+----------------+----------+---------+
|                quote                 |   characters   | stardate | episode |
+--------------------------------------+----------------+----------+---------+
| We have phasers, I vote we blast 'em! | your_name_here |   1514.2 |      10 |
+--------------------------------------+----------------+----------+---------+
root@:26258/> \q
```

The Node should have your name, instead of the name of the character from the episode.

Next, check the Node that we did not update. It too should have your name as the Character as well.

```
$ cockroach sql --insecure --port=26257


# Welcome to the cockroach SQL interface.
# All statements must be terminated by a semicolon.
# To exit: CTRL + D.

root@:26258/> SELECT * FROM startrek.quotes ORDER BY RANDOM() LIMIT 1;
+--------------------------------------+----------------+----------+---------+
|                quote                 |   characters   | stardate | episode |
+--------------------------------------+----------------+----------+---------+
| We have phasers, I vote we blast 'em! | your_name_here |   1514.2 |      10 |
+--------------------------------------+----------------+----------+---------+
root@:26258/> \q
```

Test Your Skills!

Through this workshop participants have learned to connect CockroachDB to an existing application that randomly generates Star Trek quotes. CockroachDB then split and replicated data across different Nodes in our Stark Trek cluster. Participants shut down a Node, changed some data, and then observed how the Nodes worked with each other to update data when the downed Node was taken back online.

Participants can test how much they remember about CockroachDB here:
http://mlhlocal.host/quiz

Participants if they are curious to know more about CockroachDB, they can join the CockroachDB community by posting questions for future projects on Stack Overflow:
http://mlhlocal.host/cockroach-stackoverflow