Serverless Single Page Web Apps, Part Five

CSCI 5828: Foundations of Software Engineering
Lecture 25 — 11/15/2016

Goals

- Cover Chapter 5 of Serverless Single Page Web Apps by Ben Rady
  - Present an introduction to Amazon's DynamoDB
  - Demonstrate how to integrate reading and writing documents to DynamoDB from LearnJS

Current Status

- We have a basic serverless single page web app in place
  - Displays a set of JavaScript puzzles
    - Users can navigate the puzzles
    - They can enter a solution and see if it's correct
      - They receive visual feedback when submitting their answers
  - Users can login to the system using Cognito and Google Plus
    - They can also use the system anonymously
  - Has all the basic components in place
    - events and event handlers, routers, templates, view functions

What's Next?

- Now that we support user login
  - we can demonstrate how our web app can create and access data in a database
    - Not a local database but one accessible via a third-party web service
- Our book makes use of Amazon's DynamoDB
- We will use this database to store any answer that is correct for a question
  - When we return to that question, we will access the database and automatically fill in the correct answer

Amazon's DynamoDB (I)

- DynamoDB is a NoSQL database service
- It offers
  - fast, consistent performance at any scale
    - Amazon advertises "single-digit millisecond" service latency
    - It provides this via automatic partitioning of data and the use of SSDs
  - highly scalable
    - Amazon places almost no limits on the tables you create
      - You indicate the throughput you need (requests per second) and pay for that plus storage
  - Plus: fully-managed, fine-grain access control, event-driven triggers, and flexibility: can be used as key-value store or document store.

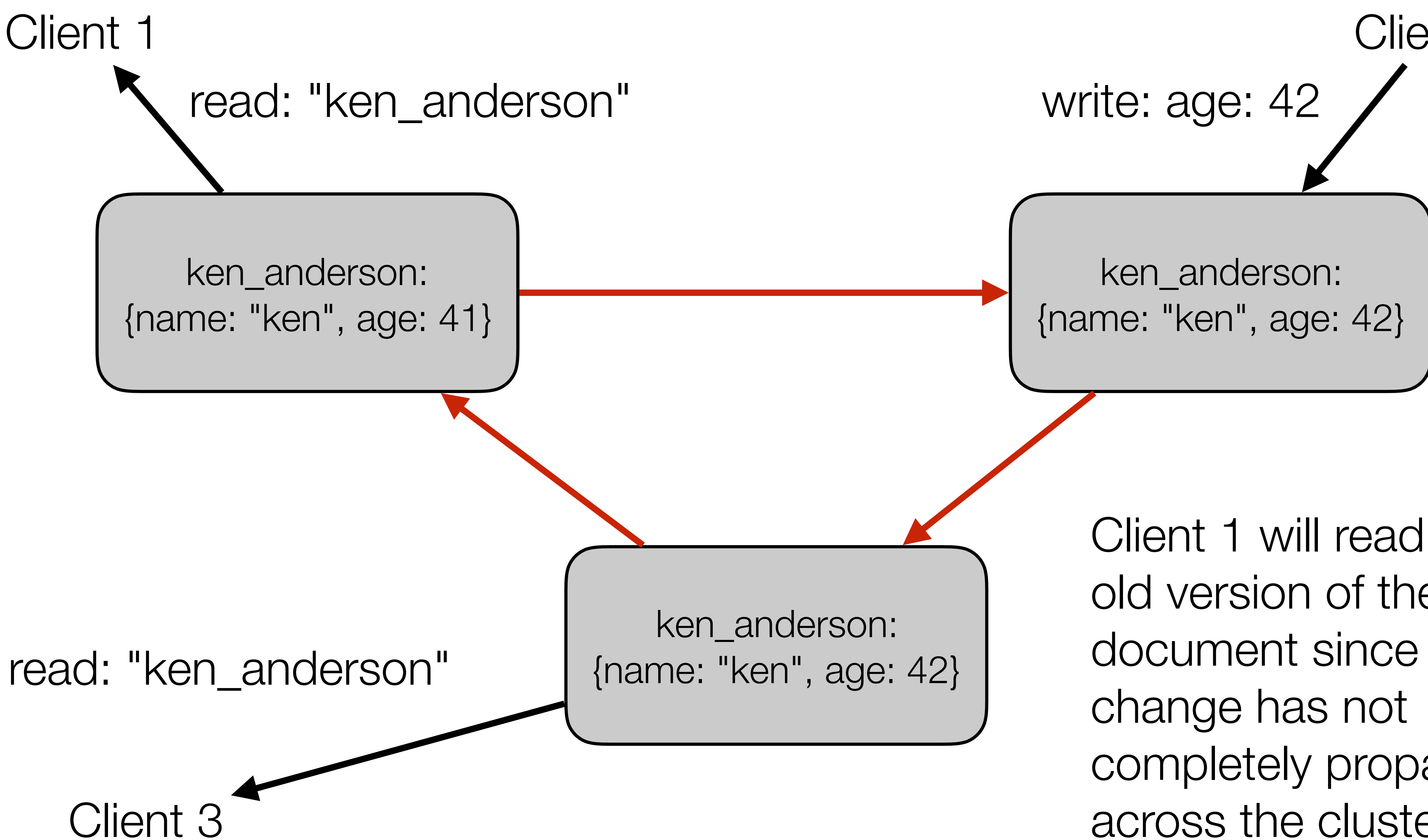
Amazon's DynamoDB (II)

- NoSQL databases
  - NoSQL stands for "No SQL" or "Not Only SQL" meaning that it is not making use of the standard relational model found in RDBMS
  - Number of Interesting Capabilities
    - A schema is typically not enforced
      - One "row" of information may have a completely different set of attributes from other "rows" in the same "table"
    - The database is designed to run on a cluster of machines
      - data is automatically distributed among the machines
        - often replicated too
      - horizontally-scalable: the more machines, the better
    - Ad hoc queries are typically not supported

Amazon's DynamoDB (III)

- Given the cluster-based nature of NoSQL data stores, they often only provide "eventual consistency" guarantees rather than "strong consistency"
- Example:
  - Create a document: {name: "ken", age: 41}
  - Store it using the key "ken_anderson"
  - Change the document: {name: "ken", age: 42}
  - Store it again with the same key
  - Ask the database for the document with key "ken_anderson"
  - Receive: {name: "ken", age: 41}
- Second Example:
  - Popular Facebook posts; view the post one time and see "1000 likes"
  - Refresh the post and see "2500 likes"; Click like yourself and see "3700"

What's going on?



Client 1 will read the old version of the document since the change has not completely propagated across the cluster.

Amazon's DynamoDB (IV)

- Concepts
  - DynamoDB stores *items* that have an arbitrary set of *attributes*
    - Each attribute has a *name* and a *value*
  - The only required attributes are its *primary key* attributes
    - The primary key can have either one or two *dimensions*
  - If the primary key has only one dimension, its value must be unique
    - This *hash primary key* will be used to store the item on a server (and store its replicas on other servers)
    - They are stored in an unordered fashion
      - Their values can be strings, numbers, or base-64 encoded binary data.

Amazon's DynamoDB (V)

- Concepts (continued)
  - If the primary key has two dimensions, then
    - the first value is called the *hash attribute*
    - the second value is called the *range attribute*
  - items are kept sorted by the range attribute
    - it is then possible
      - to scan through all values in a table in order
      - to submit queries that filter via the range attribute

Amazon's DynamoDB (VI)

- Limits
  - Items can be up to 400KB in size (including all attribute names and values)
    - attribute names can be up to 255 bytes
      - Need to be careful
        - String.length("Århus") => 5
        - byte_size("Århus") => 6
- Types
  - Attributes can be scalar: number, string, binary, or boolean, or NULL
  - Attributes can also be multivalued: StringSet, NumberSet, and BinarySet
  - Attributes can have "document types": Lists and Maps
- This means that DynamoDB is very good at storing JSON documents!

Amazon's DynamoDB (VII)

- Going back to consistency
    - DynamoDB provides both *eventually consistent* reads or *strongly consistent* reads
- Strongly consistent reads are more expensive
    - You have to purchase more capacity to use them
- They are also more likely to fail; if so, you have to try again
- Eventually consistent reads are the default; they are less expensive and more likely to succeed
    - You just have to understand that they can return out-of-date data

Creating a Table

- In order to use DynamoDB, you must first create at least one table
  - The minimum things that need to be specified are:
    - *attribute definitions* for *required* attributes; all items must have these attributes within the table; each item can have more attributes
      - You typically only define the attributes that are going to be used as your primary key
    - a key schema that indicates the role ("hash" or "range") for the attributes that serve as part of the primary key
    - the amount of provisioned throughput that your application requires
      - specified in terms of *read capacity units* and *write capacity units*
        - the free tier allows up to 25 read and 25 write capacity units

Understanding Capacity

- Each read unit gives allows you to perform one strongly consistent read per second of an item 4KB or less
  - or two eventually consistent reads per second of an item 4KB or less
- A write unit allows you to write one item per second of 1KB or less
- These units scale linearly
  - If you write one 4KB item in one second, that's 4 write units
  - If you read a 24KB item in one second that's 6 read units
- If you ever exceed your capacity, your read/write operation will fail with a *ProvisionedThroughputExceededException*
  - The book goes into some of the complexities around capacity, especially with respect to how it gets allocated to your key space (which is split among multiple partitions)

Authorization

- Behind the scenes, the sspa script sets up an access policy that allows our users to apply the following DynamoDB operations to documents they create
  - BatchGetItem, BatchWriteItem, DeleteItem, GetItem, PutItem, Query, UpdateItem
- The policy restricts these items in this way by requiring that a cognito identity be provided when performing these operations
- This policy ensures that multiple people can use our app and update our table but never see the data created by another user
  - See the book for details

Using DynamoDB

- We now return to the LearnJS web application
  - We are going to update the app such that
    - it writes a document to DynamoDB containing the answer to each question a user answers correctly
    - when a correct question is displayed again, our app will read its associated document and display the correct answer automatically
- As promised, the code that does this makes use of the "refresh" functions we discussed in Lecture 24 with respect to keeping up-to-date tokens from Google Plus and Cognito
  - The code also makes heavy use of promises to do its job
    - Let's take a look!

Generic code for interacting with DynamoDB

We create a promise and start a long running operation that will either reject or resolve the promise.

We then return the promise so it can be chained.

```
learnjs.sendDbRequest = function(req, retry) {
  var promise = new $.Deferred();
  req.on('error', function(error) {
    if (error.code === "CredentialsError") {
      learnjs.identity.then(function(identity) {
        return identity.refresh().then(
          function() {
            return retry();
          },
          function() {
            promise.reject(resp);
          });
      });
    } else {
      promise.reject(error);
    }
  });
  req.on('success', function(resp) {
    promise.resolve(resp.data);
  });
  req.send();
  return promise;
}
```

Saving an Item; Clever code at the end for retry()!

```
learnjs.saveAnswer = function(problemId, answer) {
  return learnjs.identity.then(function(identity) {
    var db = new AWS.DynamoDB.DocumentClient();
    var item = {
      TableName: 'learnjs',
      Item: {
        userId: identity.id,
        problemId: problemId,
        answer: answer
      }
    };
    return learnjs.sendDbRequest(db.put(item), function() {
      return learnjs.saveAnswer(problemId, answer);
    })
  });
};
```

Adding Save Functionality

- With the two functions above
  - the only thing that needs to change in our web app is
    - to add a call to the saveAnswer() method
      - when checking a submitted answer
  - `learnjs.saveAnswer(number, answer.val());`
    - answer points at the DOM element that contains the user's submitted answer; we use `val()` to retrieve the actual value

Loading an Item

```
learnjs.fetchAnswer = function(problemId) {
  return learnjs.identity.then(function(identity) {
    var db = new AWS.DynamoDB.DocumentClient();
    var item = {
      TableName: 'learnjs',
      Key: {
        userId: identity.id,
        problemId: problemId
      }
    };
    return learnjs.sendDbRequest(db.get(item), function() {
      return learnjs.fetchAnswer(problemId);
    })
  });
}
```

Adding Load Functionality

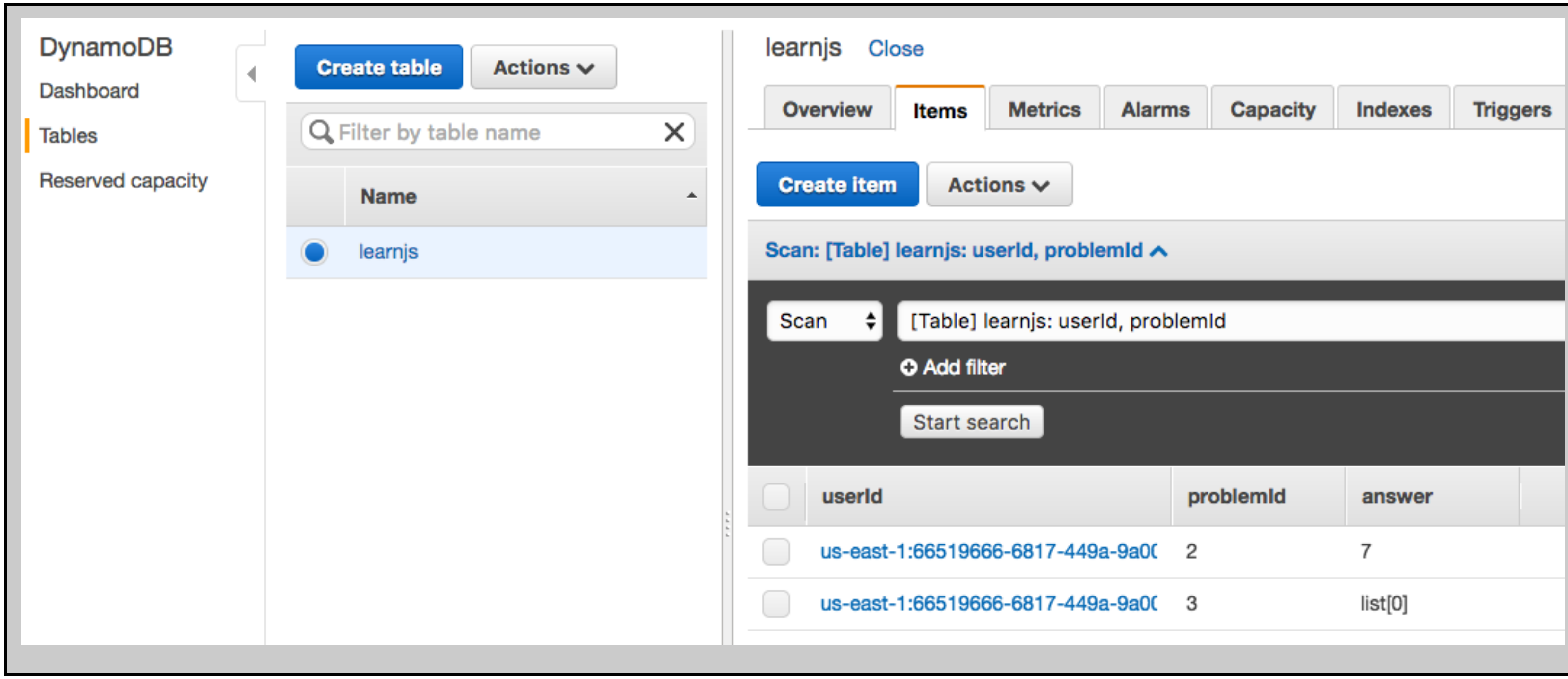
- To load a previously saved correct answer, we add the following code to the problemView() view function

```
learnjs.fetchAnswer(number).then(function(data) {
  if (data.Item) {
    answer.val(data.Item.answer);
  }
});
```

- This code is a brilliant example of closures and promises
  - The answer DOM element is captured in a closure
  - We call fetch and do not really care if the view gets updated or not
    - IF the call succeeds, then the promise will make sure that the answer DOM element is updated at some point "later"

Viewing the Table

- The documents being stored in our table can be viewed via the AWS Console



Summary

- In this chapter, we have touched on a number of topics
  - Amazon's DynamoDB
    - a distributed document database with configurable read/write capacity
    - configurable read semantics: consistent or eventually consistent
    - flexible document storage, no schema imposed on attributes
      - with the exception of identifying the attributes that serve as the primary key and range key
  - Use of promises to read/write DynamoDB documents with error handling
- Next Time: Implementing Microservices in Amazon's Lambda