REACT NATIVE

Database

Database on the App

- When you are develop the app, at one point, you will need to store information permanently for an application (it does not gone after the forced close and re-open)
- For faster loading time, Line app use a local database to remember all of your conversations.
- For caching propose, Facebook app use a local database to record all of the previously loaded news feed.

AsyncStorage

```
AsyncStorage.setItem('myKey', myValue);
```

- AsyncStorage is a simple, unencrypted, asynchronous, persistent, key-value storage system that is global to the app
- On iOS, AsyncStorage is backed by native code that stores small values in a serialized dictionary and larger values in separate files
- On Android, AsyncStorage will use either RocksDB or SQLite based on what is available.

https://reactnative.dev/docs/asyncstorage
https://github.com/react-native-async-storage/async-storage

How to use it?

- Install AsyncStorage with the following commands below:
  - npm i **@react-native-async-storage/async-storage**
  - react-native link @react-native-community/async-storage

```
import AsyncStorage from '@react-native-async-storage/async-storage';
```

- AsyncStorage uses key-value pairs so to save data, e.g. the following example.

```
const storeData = async (value) => {
  try {
    await AsyncStorage.setItem('@storage_Key', value)
  } catch (e) {
    // saving error
  }
}
```

```
const storeData = async (value) => {
  try {
    const jsonValue = JSON.stringify(value)
    await AsyncStorage.setItem('@storage_Key', jsonValue)
  } catch (e) {
    // saving error
  }
}
```

How to use it?

- To loaded the saved data, you can do like this…

```
const getData = async () => {
  try {
    const value = await AsyncStorage.getItem('@storage_Key')
    if(value !== null) {
      // value previously stored
    }
  } catch(e) {
    // error reading value
  }
}
```

```
const getData = async () => {
  try {
    const jsonValue = await
AsyncStorage.getItem('@storage_Key')
    return jsonValue != null ? JSON.parse(jsonValue) : null;
  } catch(e) {
    // error reading value
  }
}
```

- Since loading data is a time consuming task, it is designed to be a asynchronous operation. So ***getItem*** returned the promise, which will invoke the call back function when the read operation is completed.

Example1.js

```
AsyncStorage.setItem('any_key_here', textInputValue);
```

```
AsyncStorage.getItem('any_key_here').then(
  (value) =>
    //AsyncStorage returns a promise so adding
    //a callback to get the value
    setGetValue(value)
  //Setting the value in Text
);
```

Example1.js

```
return (
 <SafeAreaView style={{ flex: 1 }}>
  <View style={styles.container}>
   <Text style={styles.titleText}>AsyncStorage in React Native to Store Data in Session</Text>
   <TextInput
    placeholder="Enter Some Text here"
    value={textInputValue}
    onChangeText={(data) => setTextInputValue(data)}
    underlineColorAndroid="transparent"
    style={styles.textInputStyle}
   />
   <TouchableOpacity onPress={saveValueFunction} style={styles.buttonStyle}>
    <Text style={styles.buttonTextStyle}> SAVE VALUE </Text>
   </TouchableOpacity>
   <TouchableOpacity onPress={getValueFunction} style={styles.buttonStyle}>
    <Text style={styles.buttonTextStyle}> GET VALUE </Text>
   </TouchableOpacity>
   <Text style={styles.textStyle}> {getValue} </Text>
  </View>
 </SafeAreaView>
);
```

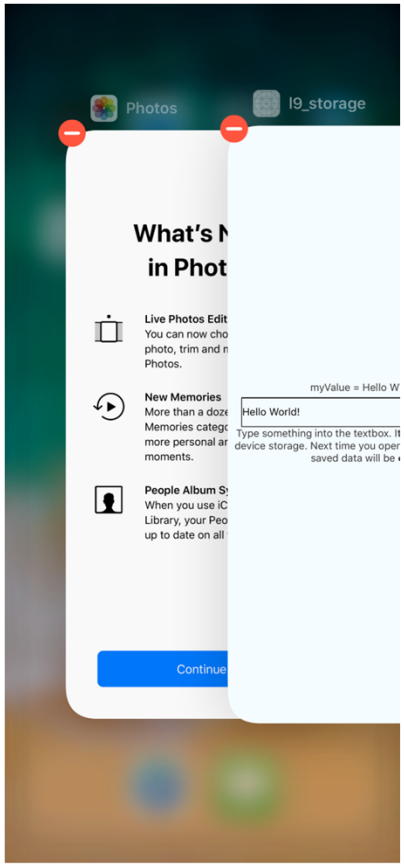
Example1.js

```
const styles = StyleSheet.create({
 container: {
  flex: 1,
  padding: 10,
  backgroundColor: 'white',
 },
 titleText: {
  fontSize: 22,
  fontWeight: 'bold',
  textAlign: 'center',
  paddingVertical: 20,
 },
 textStyle: {
  padding: 10,
  textAlign: 'center',
 },

 // continue…
});
```

```
 // continue
 buttonStyle: {
 fontSize: 16,
 color: 'white',
 backgroundColor: 'green',
 padding: 5,
 marginTop: 32,
 minWidth: 250,
},
buttonTextStyle: {
 padding: 5,
 color: 'white',
 textAlign: 'center',
},
textInputStyle: {
 textAlign: 'center',
 height: 40,
 width: '100%',
 borderWidth: 1,
 borderColor: 'green',
},
```

Simple Text Storage App



Kill the app, by pressing home button twice. Shift+Cmd+H

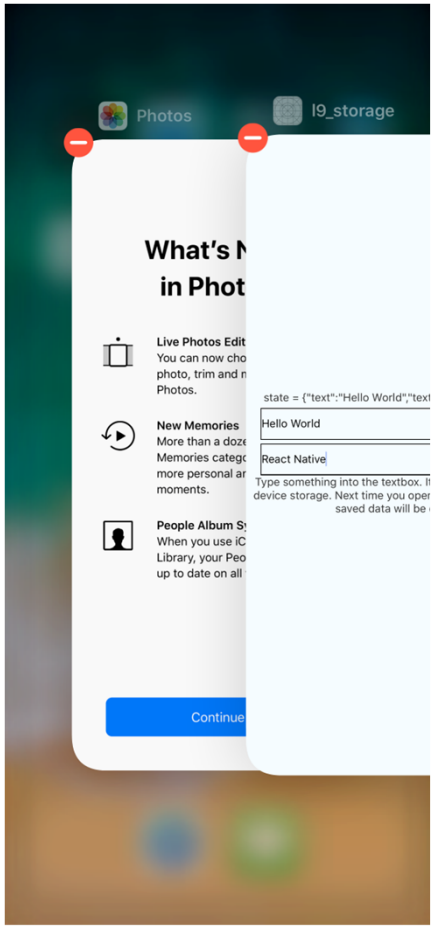
What if.. the whole state is save?

- For faster application development, Instead of we manipulate the field one-by-one, we can save the whole state into the AsyncStorage.
- *setItem()*, and *getItem()* accept only the string arguments. How we can store the whole JSON object?
- Use *JSON.stringify*() and *JSON.parse*() to convert between string and JSON object.

JSON.stringify & JSON.parse

- Object to String
  - JSON.stringify({A:1, B:2}) => “{‘A’:1, ‘B’:2}”
- String to Object
  - JSON.parse(“{‘A’:1, ‘B’:2}”) => {A:1, B:2}

Double Text Storage App



Saving / Loading the whole state.

example2.js

```
constructor(props) {
    super(props);

    this.state = {};
    AsyncStorage.getItem('state').then((myState) => {
        this.setState(JSON.parse(myState));
    });
}

saveState(myValue) {
    AsyncStorage.setItem('state', JSON.stringify(this.state));
}
```

Example2.js

```
render() {
    return (
        <View style={styles.container}>
            <Text style={styles.instructions}>
                state={JSON.stringify(this.state)}</Text>
            <TextInput style={styles.textInput}
                onChangeText={(text) => {
                    this.setState({ text: text });
                    // To make sure that the setState is complete before the saveState
                    setTimeout(() => this.saveState(), 0);
                }}
                value={this.state.text} />
            <TextInput style={[styles.textInput, { marginTop: 5 }]}
                onChangeText={(text) => {
                    this.setState({ text2: text });
                    // To make sure that the setState is complete before the saveState
                    setTimeout(() => this.saveState(), 0);
                }}
                value={this.state.text2} />
            <Text style={styles.instructions}>
                Type something into the textbox. It will be saved to the device storage.
                    Next time you open the application, the saved data will be exist
                        </Text>
        </View>
    );
}
```

Removing the Storage

```
AsyncStorage.removeItem('myKey');
```

```
clearState() {
    AsyncStorage.removeItem('state');
    this.setState({ text: '', text2: ''});
}
```

But if we want more complex database to use?

- AsyncStorage is just a key-value pair storage
- In practical application, we need to store complex information much more than key-value pair
- What if we want the database that can query, search, and support encryption with data modeling?

is the answer!

```
const CarSchema = {
    name: 'Car',
    properties: {
        make: 'string',
        model: 'string',
        miles: {type: 'int', default: 0}
    }
};
```

HIENLTH

Realm Database

- A flexible platform for creating offline-first, reactive mobile apps effortlessly.

Simple saving. Fast queries. Save weeks of development time.

Realm Mobile Database is an alternative to SQLite and Core Data. Thanks to its zero-copy design, Realm Mobile Database is much faster than an ORM, and often faster than raw SQLite. Get started in minutes, not hours.

Realm Database

Benefits of a database built for mobile

Offline-first functionality

Make your app work as well offline as it does online.

Fast queries

Even complex queries take nanoseconds, and stay up to date with new data.

Safe threading

Access the same data concurrently from multiple threads, with no crashes.

Cross-platform apps

Use the same database for all your apps, on any major platform.

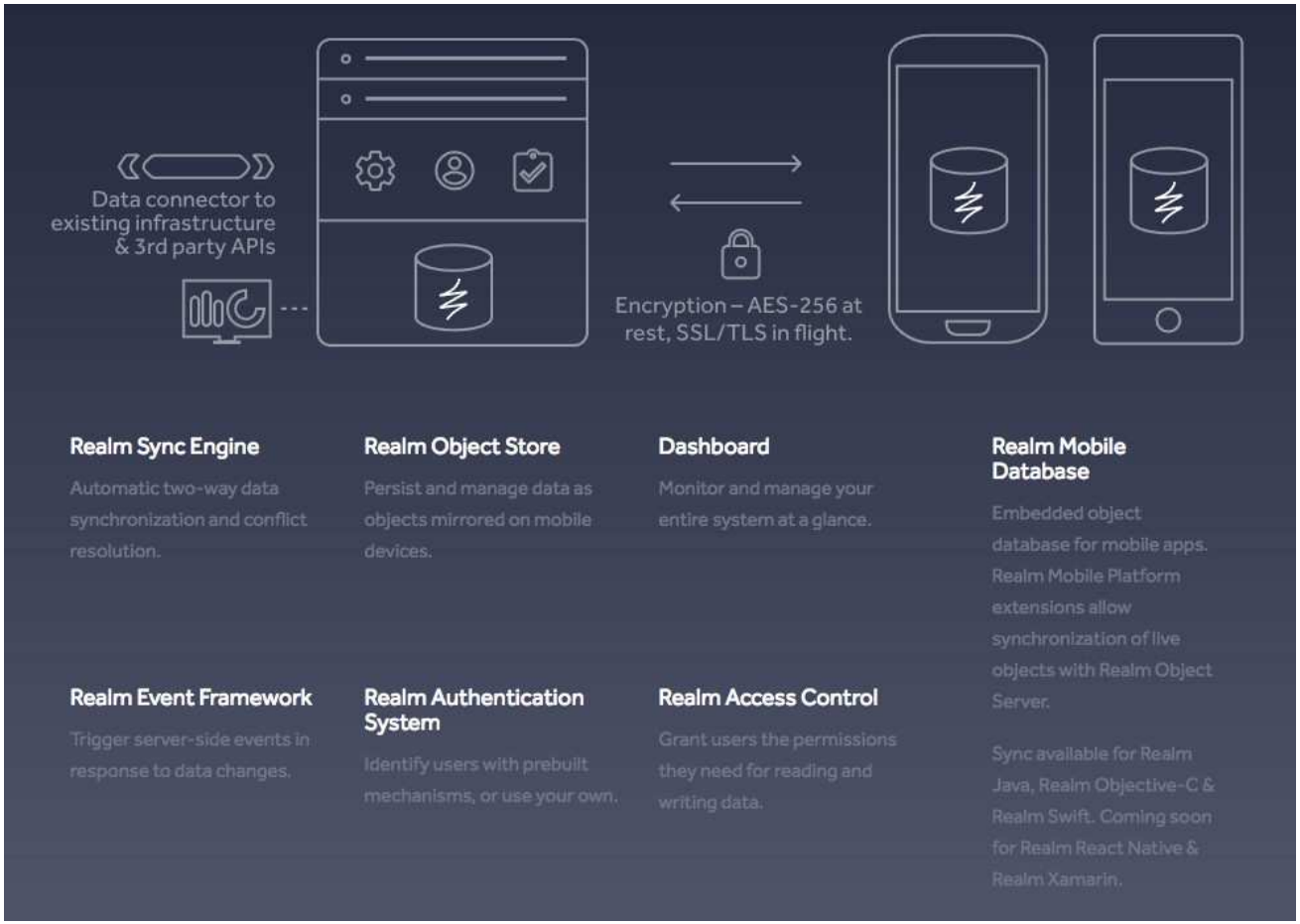
Encryption

Secure your data with transparent encryption and decryption.

Reactive architecture

Connect your UI to Realm, and data changes will appear automatically.

Realm Database

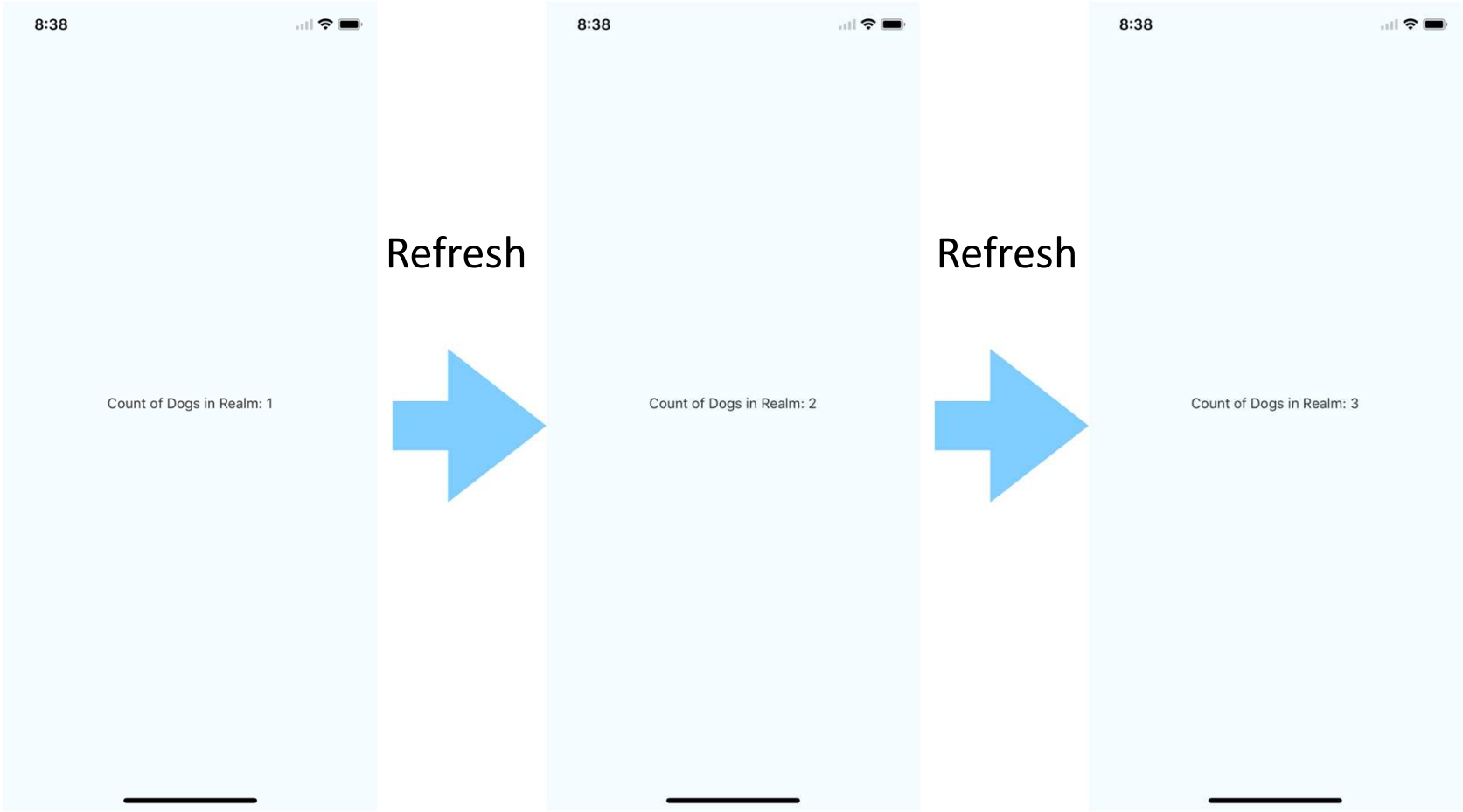


Realm Installation

- Change directories into the new project (cd <project-name>) and add the realm dependency:
  - $|> npm install --save realm
- Next use react-native to link your project to the realm native module.
  - $|> react-native link realm

https://www.npmjs.com/package/realm

Realm Database Basic



realm1.js

```
render() {
    let realm = new Realm({
        schema: [
            {
                name: 'Dog', properties: { name: 'string' }
            }
        ]
    });

    realm.write(() => {
        realm.create('Dog', { name: 'Rex' });
    });

    return(
        <View style={styles.container}>
            <Text style={styles.instructions}>
                Count of Dogs in Realm: {realm.object('Dog').length}
            </Text>
        </View>
    );
}
```

Schema Definition

Create a new Object

Show amount of object saved in DB

Realm Database Model

- Realm data models are defined by the schema information passed into a Realm during initialization.
- The schema for an object consists of the object's name and a set of properties each of which has a name and type as well as the objectType for object and list properties.
- We can also designate each property to be optional or to have a default value.

Realm Database Model

```
const Realm = require('realm');

const CarSchema = {
    name: 'Car',
    properties: {
        make: 'string',
        model: 'string',
        miles: {type: 'int', default: 0}
    }
};

const PersonShema = {
    name: 'Person',
    properties: {
        name: 'string',
        birthday: 'date',
        cars: {type: 'list', objectType: 'Car'},
        picture: {type: 'data', optional: true}
    }
}
```

- When specifying basic properties as a shorthand you may specify only the type rather than having to specify a dictionary with a single entry:

Realm's Basic Property Type

- Realm supports the following basic types: bool, int, float, double, string, data, and date.
  - bool properties map to JavaScript Boolean objects
  - int, float, and double properties map to JavaScript Number objects. Internally 'int' and 'double' are stored as 64 bits while float is stored with 32 bits.
  - string properties map to String
  - data properties map to ArrayBuffer
  - date properties map to Date

Object Properties

- To define the new object, we need to specify the name property of the object schema we are referencing..

```
const PersonShema = {
    name: 'Person',
    properties: {
        //All of the following property definitions are quivalent
        car: {type: 'Car'},
        van: 'Car'
    }
}
```

- When using object properties you need to make sure all referenced types are present in the schema used to open the Realm

```
//CarSchema is needed since PersonChema contains properties of type 'Car'
let realm = new Realm({schema: [CarSchema, PersonShema]});
```

Accessing Object Properties

- When accessing object properties, you can access nested properties using normal property syntax

```
realm.write(() => {
    var nameString = person.car.name;
    person.car.miles = 1100;

    //create a new Car by setting the property to valid
    person.van = { make: 'Fold', model: 'Transit' };

    //set both properties to the same car instance
    person.car = person.van;
});
```

Accessing Object Properties

```
const CarSchema = {
    name: 'Car',
    properties: {
        make: 'string',
        model: 'string',
        miles: {type: 'int', default: 0}
    }
};

const PersonShema = {
    name: 'Person',
    properties: {
        name: 'string',
        birthday: 'date',
        cars: {type: 'list', objectType: 'Car'},
        picture: {type: 'data', optional: true}
    }
}
```

List Properties

- For list properties you must specify the property type as list as well as the objectType

```
const PersonSchema = {
    name: 'Person',
    properties: {
        cars: {type: 'list', objectType: 'Car' }
    }
};
```

Accessing List Property

- When accessing list properties a List object is returned. List has methods very similar to a regular JavaScript array. The big difference is that **any changes made to a List are automatically persisted to the underlying Realm**.

```
let carList = person.cars;

//Add new cars to the list
realmm.write(() => {
    carList.push({ make: 'Honda', model: 'Accord', miles: 100});
    carList.push({ make: 'Toyota', model: 'Prius', miles: 200});
});

let secondCard = carList[1].model; //access using an array index
```

Optional Properties

- Properties can be declared as optional or non- optional by specifying the optional designator in your property definition:

```
const PersonSchema = {
    name: 'Person',
    properties: {
        name: {type: 'string'},                    //required property
        birthday: {type: 'data', optional: true},  //optional property

        //object properties are always optional
        car: {type: 'car'}
    }
};
```

Setting Optional Properties

```
let realm = new Realm({schema: [PersonShema, CarSchema]});

realm.write(() => {
    //optional properties can be set null or undefined at creation
    let charlie = realm.create('Person', {
        name: 'Chalie',
        birthday: new Date(1995, 11, 25),
        car: null
    });
});

//optional properties can be set to 'null', 'indefined',
//or to a new non-null value
charlie.birthday = undefined;
charlie.car = { make: 'Honda', model: 'Accord', miles: 10000 };
```

Default Property

- Default property values can be specified by setting the default designator in the property definition. To use a default value, leave the property unspecified during object creation.

```
const CarSchema = {
    name: 'Car',
    properties: {
        make: {type: 'string'},
        model: {type: 'string'},
        drive: {type: 'string', default: 'fwd'},
        make: {type: 'int', default: 0}
    }
};

realm.write(() => {
    //Since 'miles' is left out it defaults to '0, and since
    //'drive' is specified, it overrides the default value
    realm.create('Car', {make: "Honda", model: 'Accord', drive: 'awd'});
})
```

Index Property

- You can add an indexed designator to a property definition to cause that property to be indexed. This is supported for int, string, and bool property types:

```
var BookSchema= {
    name: 'Book',
    properties: {
        name: { type: 'string', indexed: true},
        price: 'float'
    }
};
```

- Indexing a property will greatly speed up queries where the property is compared for equality at the cost of slower insertions.

PrimaryKey Properties

- You can specify the primaryKey property in an object model for string and int properties.
- Declaring a primary key allows objects to be looked up and updated efficiently and enforces uniqueness for each value.
- Once an object with a primary key has been added to a Realm the primary key cannot be changed.
- Note that, Primary key properties are automatically indexed.

PrimaryKey Properties

```
var BookSchema= {
    name: 'Book',
    primaryKey: 'id',
    properties: {
        id: 'int',       //primary key
        title: 'string',
        price: 'float'
    }
};
```

Writes: Creating Objects

- Objects are created by using the *create* method.

```
let realm = new Realm({ schema: [CarSchema] });
realm.write(() => {
    realm.create('car', { make: 'Honda', model: 'Accord', drive: 'awd'});
});
```

- Nested Objects can create recursively by specifying JSON value of each child property.

```
let realm = new Realm({ schema: [PersonSchema, CarSchema] });
realm.write(() => {
    realm.create('Person', {
        name: 'Joe',
        //Nested objects are created recusively
        car: { make: 'Honda', model: 'Accord', drive: 'awd'}
    });
});
```

Writes: Updating Objects

- You can update any object by setting its properties within a write transaction.

```
realm.write(() => {
    car.miles = 1100;
});
```

- If the model class includes a primary key, you can have Realm intelligently update or add objects based off of their primary key values. This is done by passing true as the third argument to the create method:

```
realm.write(() => {
    //create a book object
    realm.create('Book', { id: 1, title: 'Receipes', price: 35 });

    //update book with new price keyed off the id
    realm.create('Book', {id: 1, price: 55}, true);
});
```

Writes: Deleting Objects

- Objects can be deleted by calling the delete method within a write transaction.

```
realm.write(() => {
    //create a book object
    let book = realm.create('Book', { id: 1, title: 'Recipes', price: 35 });

    //delete the book
    realm.delete(book);

    //delete multiple books by passing in a 'Results', 'List'
    //or JavaScript 'Array'
    let allBooks = realm.object('Book');
    realm.delete(allBooks); //delete all books
});
```

Queries

- Queries allow you to get objects of a single type from a Realm, with the option of filtering and sorting those results.
- All queries (including queries and property access) are lazy in Realm. Data is only read when objects and properties are accessed. This allows you to represent large sets of data in a performant way.
- When performing queries you are returned a Results object. Results are simply a view of your data and are not mutable.

Queries: Get All Objects

- The most basic method for retrieving objects from a Realm is using the objects method on a Realm to get all objects of a given type:

```
let dogs = realm.objects('Dog'); //retrieves all Dogs from the realm
```

Filtering

- You can get a filtered Results by calling the filtered method with a query string.
- For example, the following would to retrieve all dogs with the color tan and names beginning with ‘B’

```
let dogs = realm.object('Dog');
let tanDogs = dogs.filtered('color = "tan" AND name BEGINSWITH "B"' );
```

Filtering Support Operations

- Basic comparison operators ==, !=, >, >=, <, and <= are supported for numeric properties.
- ==, BEGINSWITH, ENDSWITH, and CONTAINS are supported for string properties.
- String comparisons can be made case insensitive by appending [c] to the operator: ==[c], BEGINSWITH[c] etc.
- Filtering by properties on linked or child objects can by done by specifying a keypath in the query eg car.color == 'blue'.

Sorting

- Results allows you to specify a sort criteria and order based on a single or multiple properties.
- For example, the following call sorts the returned cars from the example above numerically by miles:

```
let hondas = realm.objects('Car').filtered('maked = "Honda"');

//Sort Hondas by mileage
let sortedHondas = hondas.sorted('miles');
```

Auto-Updating Result

- Results instances are live, auto-updating views into the underlying data, which means results never have to be re- fetched. Modifying objects that affect the query will be reflected in the results immediately.
- This applies to all Results instances, included those returned by the objects, filtered, and sorted methods.

```
let hondas = realm.objects('Car').filtered('maked = "Honda"');
//hondas.length = 0

realm.write(() => {
    realm.create('Car', { make: 'Honda', model: 'RSX'});
});
//hondas.length = 1
```

Size Limiting

- Realm are lazy, performing this paginating behavior isn't necessary at all, as Realm will only load objects from the results of the query once they are explicitly accessed.
- If for UI-related or other implementation reasons you require a specific subset of objects from a query, it's as simple as taking the Results object, and reading out only the objects you need.

```
let cars = realm.objects('Car');

//get first 5 car objects
let firstCars = cars.slice(0, 5);
```

Realm API Doc

- Realm API Reference
  - https://realm.io/docs/javascript/latest/api/
- Realm Tutorial
  - https://realm.io/docs/javascript/latest/#getting-%20started

Thank you!