Realm mobile platform

by Anton Minashkin

Realm Mobile Platform

Combination of Realm DB on mobile devices and Realm Object Server on the backend

What is Realm DB?

Realm is a replacement for SQLite & Core Data

Fast

“Thanks to its **zero-copy design**, Realm is much faster than an ORM, and is often faster than raw SQLite as well”

Cross-Platform

- iOS
- OS X
- Android
- Xamarin

Easy to Use

Realm is not an ORM on top of SQLite. Instead it uses its own persistence engine, built for simplicity (& speed)

Advanced

- Encryption
- Graph queries
- Easy migrations
- RxJava support

Getting started

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath "io.realm:realm-gradle-plugin:2.3.1"
    }
}

...

apply plugin: 'realm-android'
```

Getting started

```
public class Dog extends RealmObject {
    @Required // Name cannot be null
    private String name;
    private int age;

    // ... Generated getters and setters ...
}
```

Getting started

```
public class Person extends RealmObject {
    @Required // Name is not nullable
    private String name;
    private String imageUrl; // imageUrl is an optional field
    private RealmList<Dog> dogs; // A person has many dogs (a relationship)

    // ... Generated getters and setters ...
}
```

Getting started

```
public class Person extends RealmObject {
    @Required // Name is not nullable
    private String name;
    private String imageUrl; // imageUrl is an optional field
    private RealmList<Dog> dogs; // A person has many dogs (a relationship)

    // ... Generated getters and setters ...
}
```

Getting started

```
Dog dog = new Dog();
dog.setName("Rex");
dog.setAge(1);
Log.v(TAG, "Name of the dog: " + dog.getName());
...

Realm realm = Realm.getDefaultInstance();

realm.beginTransaction();
realm.copyToRealm(dog);
realm.commitTransaction();

...

RealmResults<Dog> puppies = realm.where(Dog.class).lessThan("age", 2).findAll();
```

Auto-Updating Objects

```
realm.beginTransaction();
Dog myDog = realm.createObject(Dog.class);
myDog.setName("Fido");
myDog.setAge(1);
realm.commitTransaction();

Dog myPuppy = realm.where(Dog.class).equals("age", 1).findFirst();
realm.beginTransaction();
myPuppy.setAge(2);
realm.commitTransaction();

myDog.getAge(); // => 2
```

Indexes

- **@Index** and **@PrimaryKey**
- Supported types for **@Index**: **String**, **byte**, **short**, **int**, **long**, **boolean** and **Date**
- Supported types for **@PrimaryKey: String**, **short**, **int** and **long**
- With **@PrimaryKey** it is possible to use **createOrUpdate()**

Fields limitations

no support for **final**, **transient** and **volatile**

Relationships: Many-to-One

```
public class Email extends RealmObject {
    private String address;
    private boolean active;
    // ... setters and getters left out
}

public class Contact extends RealmObject {
    private String name;
    private Email email;
    // ... setters and getters left out
}
```

Relationships: Many-to-Many

```
public class Contact extends RealmObject {
    private RealmList<Email> emails;
    // Other fields...
}
```

Link queries

```
public class Person extends RealmObject {
  private String id;
  private String name;
  private RealmList<Dog> dogs;
  // getters and setters
}

public class Dog extends RealmObject {
  private String id;
  private String name;
  private String color;
  // getters and setters
}
```

Link queries

```
// users => [U1,U2]
RealmResults<User> users = realm.where(User.class)
                                .equalTo("dogs.color", "Brown")
                                .findAll();
```

Writes

- All write operations (**adding**, **modifying**, and **removing** objects) must be wrapped in write transactions
- Write transaction can either be committed or cancelled
- Write transactions **block each other. ANR ALARM!**
- After transaction **all instances of Realm will be notified & updated**
- ACID

Writes

```
realm.beginTransaction();
User user = realm.createObject(User.class); // Create a new object
user.setName("John");
user.setEmail("john@corporation.com");
realm.commitTransaction();
```

Writes: Transaction blocks

```
realm.executeTransaction(new Realm.Transaction() {
      @Override
      public void execute(Realm realm) {
            User user = realm.createObject(User.class);
            user.setName("John");
            user.setEmail("john@corporation.com");
      }
});
```

Asynchronous Transactions

- Runs on background thread
- Callbacks are handled via **Looper**
- Represented by the **RealmAsyncTask**
- Cancel it when component is dying

Asynchronous Transactions

```
realm.executeTransactionAsync(new Realm.Transaction() {
            @Override
            public void execute(Realm bgRealm) {
                User user = bgRealm.createObject(User.class);
                user.setName("John");
                user.setEmail("john@corporation.com");
            }
        }, new Realm.Transaction.OnSuccess() {
            @Override
            public void onSuccess() {...}
        }, new Realm.Transaction.OnError() {
            @Override
            public void onError(Throwable error) {...}
        });
```

Queries

- All fetches (including queries) are lazy in Realm, and the data is never copied.
- Realm’s query engine uses a Fluent interface to construct multi-clause queries
- Return **RealmResult<T>**
- Result is always **not null**

Queries

```
// Build the query looking at all users:
RealmQuery<User> query = realm.where(User.class);
// Add query conditions:
query.equalTo("name", "John");
query.or().equalTo("name", "Peter");
// Execute the query:
RealmResults<User> result1 = query.findAll();
// Or alternatively do the same all at once (the "Fluent interface"):
RealmResults<User> result2 = realm.where(User.class)
                                  .equalTo("name", "John")
                                  .or()
                                  .equalTo("name", "Peter")
                                  .findAll();
```

Queries: Conditions

- **between, greaterThan(), lessThan(), greaterThanOrEqualTo()** & **lessThanOrEqualTo()**
- **equalTo()** & **notEqualTo()**
- **contains(), beginsWith() & endsWith()**

Queries: Logical Operators

```
RealmResults<User> r = realm.where(User.class)
                            .greaterThan("age", 10)  //implicit AND
                            .beginGroup()
                                .equalTo("name", "Peter")
                                .or()
                                .contains("name", "Jo")
                            .endGroup()
                            .findAll();
```

Queries: Sorting

```
RealmResults<User> result = realm.where(User.class).findAllSorted("age"); // Sort ascending
RealmResults<User> result = realm.where(User.class).findAllSorted("age", Sort.DESCENDING);
```

Queries: Chaining

```
RealmResults<User> teenagers = realm.where(User.class).between("age", 13, 20).findAll();
User firstJohn = teenagers.where().equalTo("name", "John").findFirst();
```

Queries: Aggregation

```
RealmResults<User> results = realm.where(User.class).findAll();
long   sum     = results.sum("age").longValue();
long   min     = results.min("age").longValue();
long   max     = results.max("age").longValue();
double average = results.average("age");

long   matches = results.size();
```

Queries: RxJava (RxAndroid)

```
Realm realm = Realm.getDefaultInstance();
RealmResults<Person> persons = realm.where(Person.class).findAll();
Person person = persons.first();

Observable<Realm> realmObservable = realm.asObservable();
Observable<RealmResults<Person>> resultsObservable = persons.asObservable();
Observable<Person> objectObservable = person.asObservable();
```

Queries: Asynchronous Queries

```
RealmResults<User> result = realm.where(User.class)
                                 .equalTo("name", "John")
                                 .or()
                                 .equalTo("name", "Peter")
                                 .findAllAsync();
```

Queries: Asynchronous Queries

```
private RealmChangeListener callback = new RealmChangeListener() {
    @Override
    public void onChange() { // called once the query complete and on every update
    // use the result
    }
};

public void onStart() {
    RealmResults<User> result = realm.where(User.class).findAllAsync();
    result.addChangeListener(callback);
}
```

Queries: Asynchronous Queries

```
RealmResults<User> result = realm.where(User.class).findAllAsync();
if (result.isLoaded()) {
  // Results are now available
}
```

Queries: Asynchronous Queries

```
public void onStop () {
    result.removeChangeListener(callback); // remove a particular listener
    // or
    result.removeChangeListeners(); // remove all registered listeners
}
```

Queries: Asynchronous Queries Force Load

```
RealmResults<User> result = realm.where(User.class).findAllAsync();
result.load() // be careful, this will block the current thread until it returns
```

Queries: Asynchronous Queries

Only looper thread!

Dynamic Realm

- No Schema
- No Version
- No Migration

Closing Realm instances

- **Realm** implements **Closeable**
- **Realm** instances are **reference counted**
- If you will not close - native resources will leak

Closing Realm instances

```
protected Void doInBackground(Void... params) {
    Realm realm = null;
    try {
        realm = Realm.getDefaultInstance();
        // ... Use the Realm instance
    } finally {
        if (realm != null) {
            realm.close();
        }
    }

    return null;
}
```

Closing Realm instances

```
public class MyThread extends Thread {
    private Realm realm;
    public void run() {
        Looper.prepare();
        try {
            realm = Realm.getDefaultInstance();
            //... Setup the handlers using the Realm instance
            Lopper.loop();
        } finally {
            if (realm != null) {
                realm.close();
            }
        }
    }
}
```

Closing Realm instances

IF API > 19

```
try (Realm realm = Realm.getDefaultInstance()) {
    // No need to close the Realm instance manually
}
```

Threading

- Object and Queries are auto-updated
- **Realm**, **RealmObject** or **RealmResults** instances cannot be passed across threads

More features

- JSON
- Notifications
- Encryption
- Migrations

What is Realm Mobile Platform

What is Realm Mobile Platform

- Offline first. Stored locally in the Realm DB
- Each device can fully function when offline
- The Realm Mobile Platform handles the complexities of network state and conflict management
- Can be deployed on-premises or in the public cloud (like AWS, Azure, and other popular options)
- Can be integrated with any other 3rd party service

Components

- Realm Sync Engine
- Realm Object Store
- Dashboard
- Realm Event Framework
- Realm Authentication System
- Realm Access Control

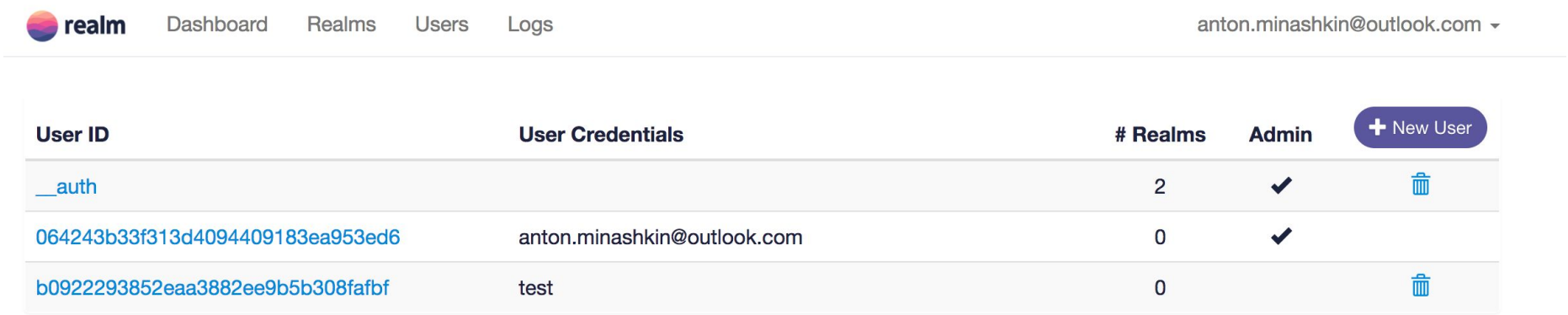
Realm Object Server

The Realm Object Server provides automatic data sync, simple authentication, access controls, event handling, and more. Deploy it anywhere, and easily integrate with existing systems.

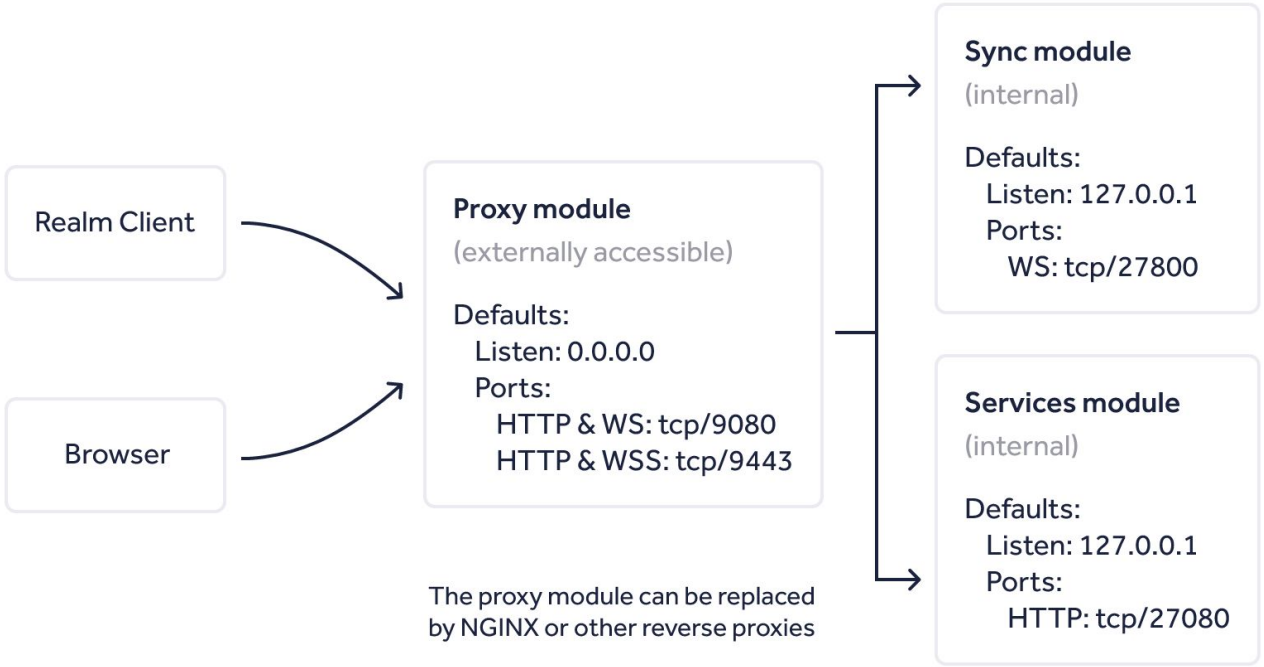
How to install

```
# Setup Realm's PackageCloud repository

curl -s https://packagecloud.io/install/repositories/realm/realm/script.deb.sh | sudo bash

# Update the repositories

sudo apt-get update

# Install the Realm Object Server

sudo apt-get install realm-object-server-developer

# Enable and start the service

sudo systemctl enable realm-object-server

sudo systemctl start realm-object-server
```

Dashboard


realm
Dashboard
Realms
Users
Logs
anton.minashkin@outlook.com
User ID
User Credentials
Realms
Admin
New User
__auth
2
064243b33f313d4094409183ea953ed6
anton.minashkin@outlook.com
0
b0922293852eaa3882ee9b5b308fafbf
test
0

Network architecture



Backend Services

HTTP Service (port: 27080)

Sync Service (port: 27800)

Administering The Server

sudo systemctl status realm-object-server

sudo systemctl start realm-object-server

sudo systemctl stop realm-object-server

sudo systemctl restart realm-object-server

Proxy Module

The included reverse proxy module dispatches all incoming traffic to the appropriate backend service. This proxy is capable of handling both HTTP, WebSocket, HTTPS and Secure WebSocket traffic (but exclusively traffic destined for the Realm Object Server, this is not a general purpose proxy).

Conflict resolving

At a very high level the rules are as follows:

- **Deletes always wins**. If one side deletes an object it will always stay deleted, even if the other side has made changes to it later on.
- **Last update wins**. If two sides has updated the same property, the value will end up as the last updated.
- **Inserts in lists are ordered by time**. If two items are inserted at the same position, the item that was inserted first will end up before the other item.

Access Control

Realm Object Server includes access control mechanisms to restrict which users are allowed to sync against which Realm files. There are two concepts: Authentication and Authorization. In order to grant some user access to a Realm, Realm Object Server authenticates the user to verify the identity of the user, and authorizes that the user has the correct permissions to access the Realm.

Other stuff

- Load Balancing (EE)
- High Availability
- Offline-First
- Failover (EE)

Backup

- The **manual backup** system is included in all versions of Realm Mobile Platform via a command line utility. It can be triggered during server operations to create a copy of the data in the Object Server.
- The **continuous backup system**, available only in the Enterprise Edition, is a backup server running alongside the main process in the Realm Object Server. It continuously watches for changes across all synchronized Realms and sends changes to one or more backup clients.

Event Handling (PE & EE)

This functionality is provided in the server-side Node.js SDK via a global event listener API which hooks into the Realm Object Server, allowing you to observe changes across Realms.

Event Handling (PE & EE)

```
{
    "name": "MyApp",
    "version": "0.0.1",
    "main": "index.js",
    "author": "Your Name",
    "description": "My Cool Realm App",
    "dependencies": {
        "realm": "file:realm-1.0.0-professional.tgz"
    }
}
```

Event Handling (PE & EE)

```
function handleChange(changeEvent) {
  ...
  var realm = changeEvent.realm;
  var coupons = realm.objects('Coupon');
  for (var couponIndex in couponIndexes) {
...
    }
  }
}
...
// register the event handler callback
Realm.Sync.addListener(SERVER_URL, adminUser, NOTIFIER_PATH, 'change', handleChange);
```

Enabling Realm Mobile Platform

```
realm {
  syncEnabled = true;
}
```

...which will cause additional lib downloading

User

- User is the **α** & **Ω**
- User – that is associated with a synchronized Realm
- User can be authenticated to a shared Realm via a username/password scheme or through a number of 3rd party authentication methods

User

User requires 2 objects:

- A URL (as a string) of a Realm Authentication Server
- Credentials for an authentication mechanism

Creating a Credential

Password/Login:

```
SyncCredentials myCredentials = SyncCredentials.usernamePassword(username, password, true);
```

Google:

```
String token = "..."; // a string representation of a token obtained by Google Login API
SyncCredentials myCredentials = SyncCredentials.google(token);
```

Facebook

```
String token = "..."; // a string representation of a token obtained by Facebook Login API
SyncCredentials myCredentials = SyncCredentials.facebook(token);
```

Creating a Credential

Custom Auth:

```
String token = "..."; // a string representation of a token obtained from your authentication server
Map<String, Object> customData = new HashMap<>();
SyncCredentials myCredentials = SyncCredentials.custom(
            token,
            'myauth',
            customData,
    );
```

Login User

```
String authURL = "http://my.realm-auth-server.com:9080/auth";
SyncUser user = SyncUser.login(myCredentials, authURL);
```

Login User

```
SyncUser user = getUserFromLogin();
String serverURL = "realm://my.realm-server.com:9080/~/default";
SyncConfiguration configuration = new SyncConfiguration.Builder(user, serverURL).build();
```

Where is my User?

```
SyncUser user = SyncUser.currentUser();
String userJson = user.toJson();

SyncUser user = SyncUser.fromJson(userJson);

...

SyncManager.setUserStore(userStore)
```

Client Access Control

- **mayRead** indicates that the user is allowed to read from the Realm,
- **mayWrite** indicates that the user is allowed to write to the Realm,
- **mayManage** indicates that the user is allowed to change the permissions for the Realm.

Client Access Control

```
SyncUser user = SyncUser.currentUser();
String realmURL = "realm://ros.example.com/~/default"; // The remote Realm URL on which to apply the changes
String anotherUserID = "other-user-id"; // The user ID for which these permission changes should be applied
Realm realm = user.getManagementRealm();
realm.executeTransaction(new Realm.Transaction() {
        @Override public void execute(Realm realm) {
                Boolean mayRead = true; // Grant read access
                Boolean mayWrite = null; // Keep current permission
                boolean mayManage = false; // Revoke management access
                PermissionChange change = new PermissionChange(realmUrl, anotherUserID, mayRead, mayWrite, mayManage);
                realm.insert(change);
        }
});
```

Client Access Control

```
SyncUser user = SyncUser.currentUser();

Realm realm = user.getManagementRealm();

final String permissionId = "permission-id";

PermissionChange change = realm.where(PermissionChange.class).equalTo("id", permissionId).findFirst();
change.addChangeListener(new RealmChangeListener<PermissionChange>() {

        @Override public void onChange(PermissionChange change) {

                if (change.getId().equals(permissionId)) {

                Integer status = change.getStatusCode();

                if (status == 0) { // Handle success

                        realm.close();

                }

                else if (status > 0) { // Handle error }

                }

        }

});
```

Error Handling

```
SyncConfiguration configuration = new SyncConfigurtion.Builder(user, serverURL)
        .errorHandler(new Session.ErrorHandler() {
                void onError(Session session, ObjectServerError error) {
                        // do some error handling
                }
        })
        .build();
```

Error Handling

```
SyncManager.setDefaultSessionErrorHandler(myErrorHandler);
```

When to use?

- PoC
- Mobile first apps
- Real time data sync apps

What to read

https://realm.io/docs/

https://realm.io/news/

https://github.com/realm

Twitter: @AntonMinashkin

Email: anton.minashkin@outlook.com


I HAZ A QUESTION