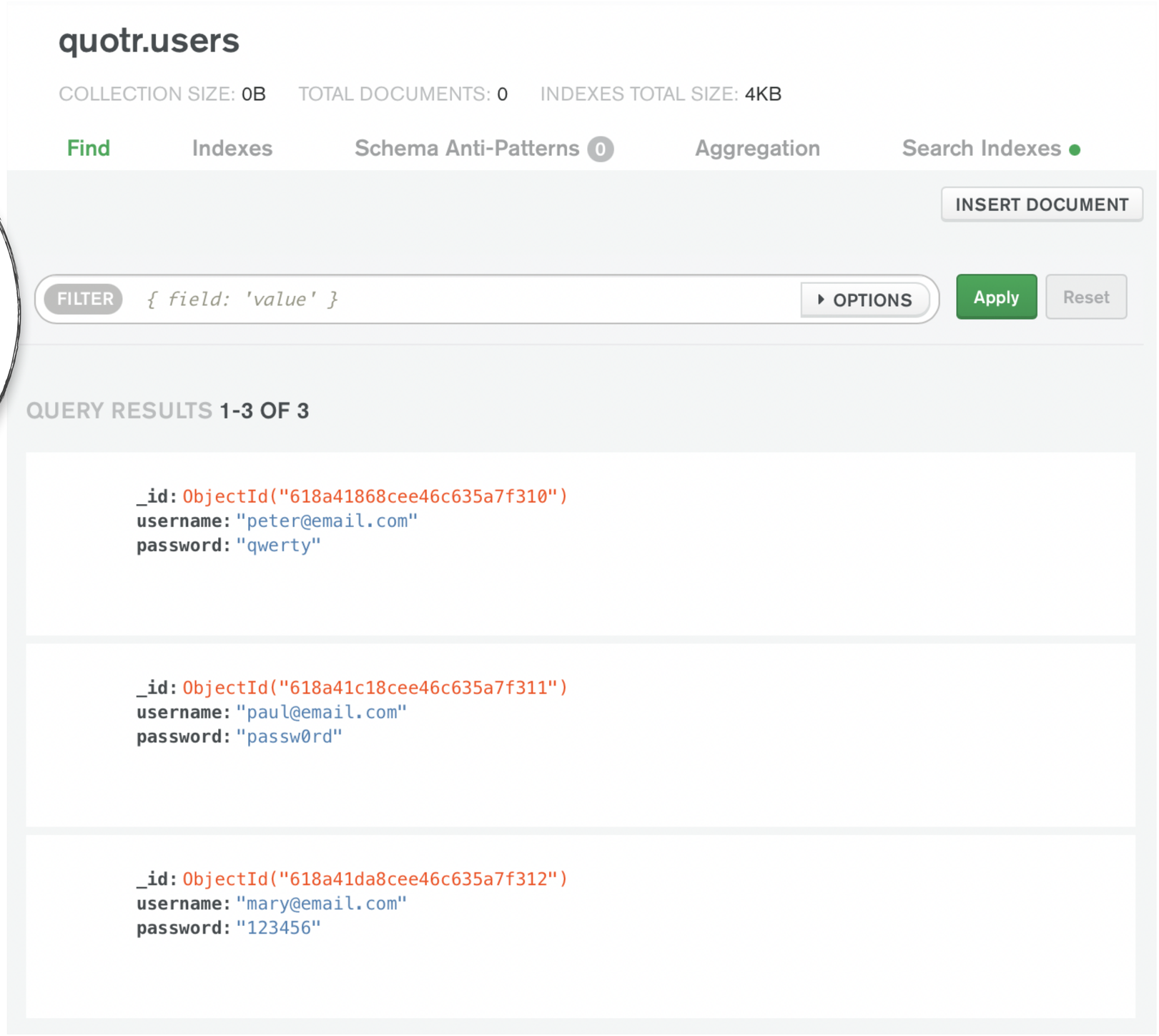
Password hashing

Let's just put the users and their passwords in the database!

- Why is this a bad idea?
- Quick exercise:
  Go to **haveibeenpwned.com** and see if your main email address has ever been "pwned" in a data breach 🙈



Password Security Basics

What is hashing and what do you use it for?

Cryptographic Hashing is computing a (fixed-length) output value from an input value in a way that makes it practically impossible to do the reverse calculation.

What is salting and why is it needed in hashing?

Salting is adding something extra to the input of the hash function to ensure unique hashes even for e.g. duplicate passwords

What is the difference between encryption and hashing?

Encryption is taking some content and encrypting it to hide the content – *with the intent of also doing the reverse, e.g. decrypting it again.*

What does it mean for something to be digitally signed?

Signing is using encryption to create a digital signature. Everyone can verify the signature, but only the owner of the key can make new signatures.

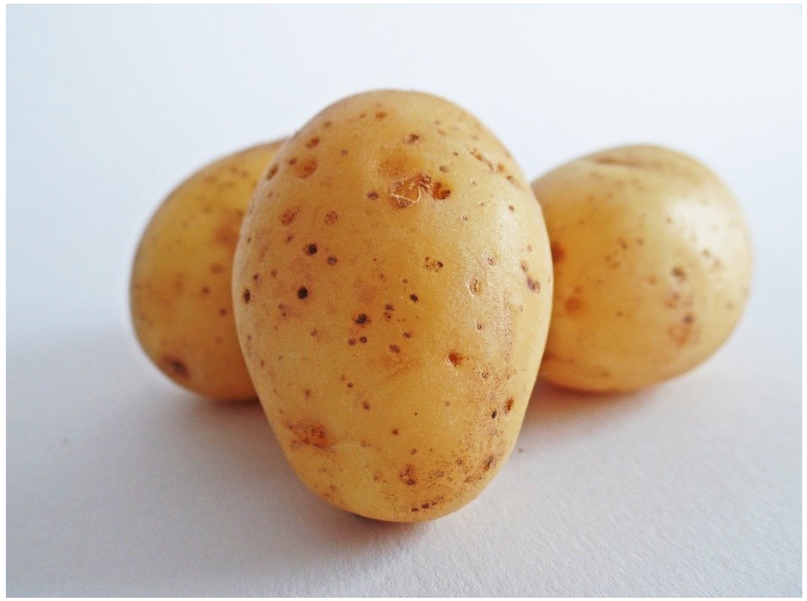
Password hashing

Before we authenticate a user, we need to verify that the user has the correct password.

Passwords on the server should **never be stored in clear text**.

Instead: store the hash of the password. The hash is easily computed from the password, but it is practically impossible to compute the password from the hash.

If two hashes match, you know that they were computed from the same password. This is how your login mechanism should work.



quotr.users

COLLECTION SIZE: 0B TOTAL DOCUMENTS: 0 INDEXES TOTAL SIZE: 4KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

INSERT DOCUMENT

FILTER { field: 'value' } OPTIONS Apply Reset

QUERY RESULTS 1-3 OF 3

```
_id: ObjectId("618a41868cee46c635a7f310")
username: "peter@email.com"
hashed_pw: "$2a$10$8.T3ILvKOsHCgmSmcEYpiuRFOwviJXPzYL532v.dCB.JApIZWqdx2"
```

```
_id: ObjectId("618a41c18cee46c635a7f311")
username: "paul@email.com"
hashed_pw: "$2a$10$Mh37ajACLA2oLkRnn63sOe6y1.txVPskyKLXJwgK1LneqZcyIzsWq"
```

```
_id: ObjectId("618a41da8cee46c635a7f312")
username: "mary@email.com"
hashed_pw: "$2a$10$0WPw.mvPeHtObjtk63p8EeXzMcQK1lRLqBb.Tp59WXD2AH6TTpr5u"
```

Login algorithm

1. Users sends **username** and **password** to server.

2. Server looks up user and retrieves the **stored hash**.

3. Server also computes the hash of the password given.

4. Server compares **stored hash** with **computed hash**. Server verifies that they are equal. If they are, the user sent the correct password!

===

5. We can then save the authentication status in the **session cookie**

If the hash values don't match, the server sends back an **error** in step 5 instead.

!==

Salting a password

To improve security, the **hash value** is actually computed from **password + salt**.

Salt is just some "random value".

The salt is stored with the computed hash.

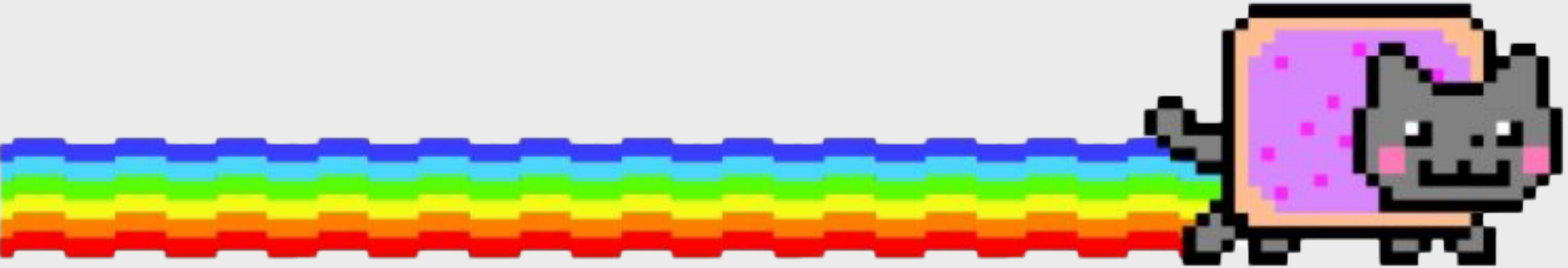
The login algorithm is the same as before, but you have to remember to "add salt" whenever you calculate the hash value.

Why salting?

Why bother with salt at all? Because it makes figuring out the password from the hash value much harder for hackers.

When using salt, even though the user used the same password on two different websites, the hash values computed on each site will not be the same.

If a password alway hashes to the same value, hackers could utilize **rainbow tables**, meaning precomputed maps of hash 👉 password.



Username	Password	Password Hash
silverlion	Pa55w0rd!	`e3ce84f159b7a1f6d27c01df1c905be7`
moonwalker	ilovemusic1	`0db5e7f8e6c0137d8a68f9de5fe5972a`
emerald123	Secret123!	`d12c4e1962abdbac60a9dcde0250ac71`
blueroses	P@ssw0rd!	`e3ce84f159b7a1f6d27c01df1c905be7`
stargazer	skywatcher1	`68b63f2ab78255c2430a1b77a41f828a`
firefly007	password123	`5f4dcc3b5aa765d61d8327deb882cf99`
cosmicgirl	Gal@xy123	`6aeb0fbfe6c249ed69d6683ea7e6c5c1`
dreamweavr	dr3amC@tch3r	`514b8f2e58a0d01bdaa3e3a9656338e1`
stellarnet	Sunshine42!	`14a6c8df16c33e01e2e876ab4a2c45f5`
nebulaX	P@ssw0rd!	`e3ce84f159b7a1f6d27c01df1c905be7`

Username	Password	Password Hash
silverlion	Pa55w0rd!	e3ce84f159b7a1f6d27c01df1c905be7
moonwalker	ilovemusic1	0db5e7f8e6c0137d8a68f9de5fe5972a
emerald123	Secret123!	d12c4e1962abdbac60a9dcde0250ac71
blueroses	P@ssw0rd!	e3ce84f159b7a1f6d27c01df1c905be7
stargazer	skywatcher1	68b63f2ab78255c2430a1b77a41f828a
firefly007	password123	5f4dcc3b5aa765d61d8327deb882cf99
cosmicgirl	Gal@xy123	6aeb0fbfe6c249ed69d6683ea7e6c5c1
dreamweavr	dr3amC@tch3r	514b8f2e58a0d01bdaa3e3a9656338e1
stellarnet	Sunshine42!	14a6c8df16c33e01e2e876ab4a2c45f5
nebulaX	P@ssw0rd!	e3ce84f159b7a1f6d27c01df1c905be7

Username	Password	Salt	Password Hash
silverlion	Pa55w0rd!	a1b2c3d4e5	78d3630ed5c69ac0d4c4f0b8d5dbf76f
moonwalker	ilovemusic1	f6e5d4c3b2	4f7d8e9c0a56b2135415e96f8741c32a
emerald123	Secret123!	9876543210	35b94ee2f5b6ac8ad7a276208ea9a35f
blueroses	P@ssw0rd!	1a2b3c4d5e	1a23b45c6d78e90f12a34b56c7890de1
stargazer	skywatcher1	5e4d3c2b1a	8a7b6c5d4e3f2a1b0c9876543210fedc
firefly007	password123	0f1e2d3c4b	5f4dcc3b5aa765d61d8327deb882cf99
cosmicgirl	Gal@xy123	dcbazyx321	b84f43e502c6a9dfe76be0a217db70b9
dreamweavr	dr3amC@tch3r	246813579a	77ef10d1c200f708286be48e507c3511
stellarnet	Sunshine42!	1q2w3e4r5t	09233d4174b5e9ca4e16874a7a2df31c
nebulaX	P@ssw0rd!	5t4r3g2a1z	283e1e4e20eb9c6b9f6d1cc1ef19417e

Hashing (and salting) with bcryptjs

Available for node.js:

```
npm install bcryptjs
```

Usage:

- Input: Password (it picks a new salt automatically)
- Output: Hash + Salt

Check incoming password as follows:

1. Hash new incoming password
2. Check hash against hash stored in database
3. If matching, allow login

`bcrypt` takes care of salting automatically. When storing a hash made by `bcrypt`, the salt is automatically appended to the end of the hash value.

You never have to remember to "add salt" with `bcrypt`. It's done for you.