KillrChat Exercises Handbook

DuyHai DOAN, Technical Advocate

KillrChat presentation

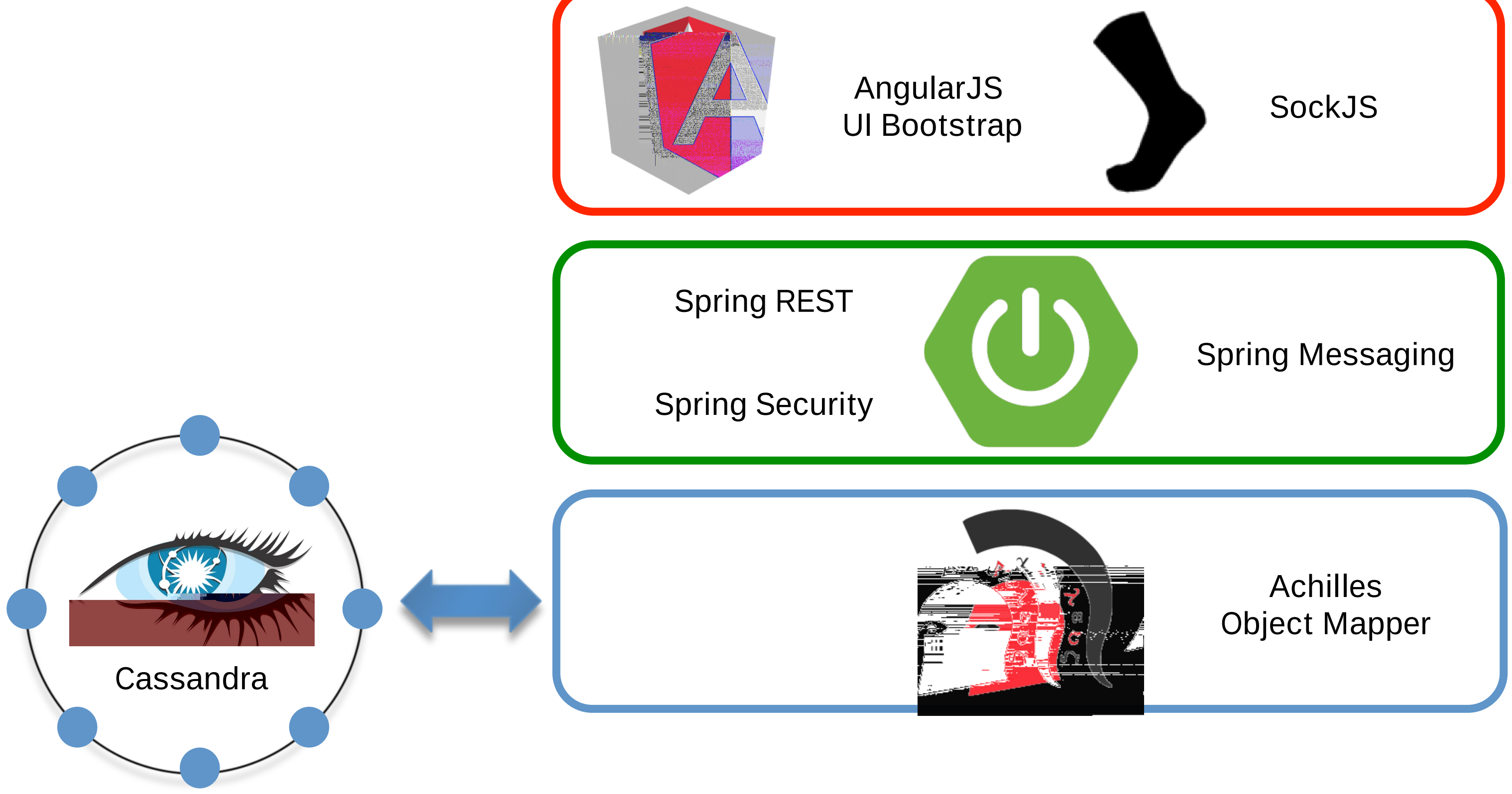
What is **KillrChat** ?

- scalable messaging app

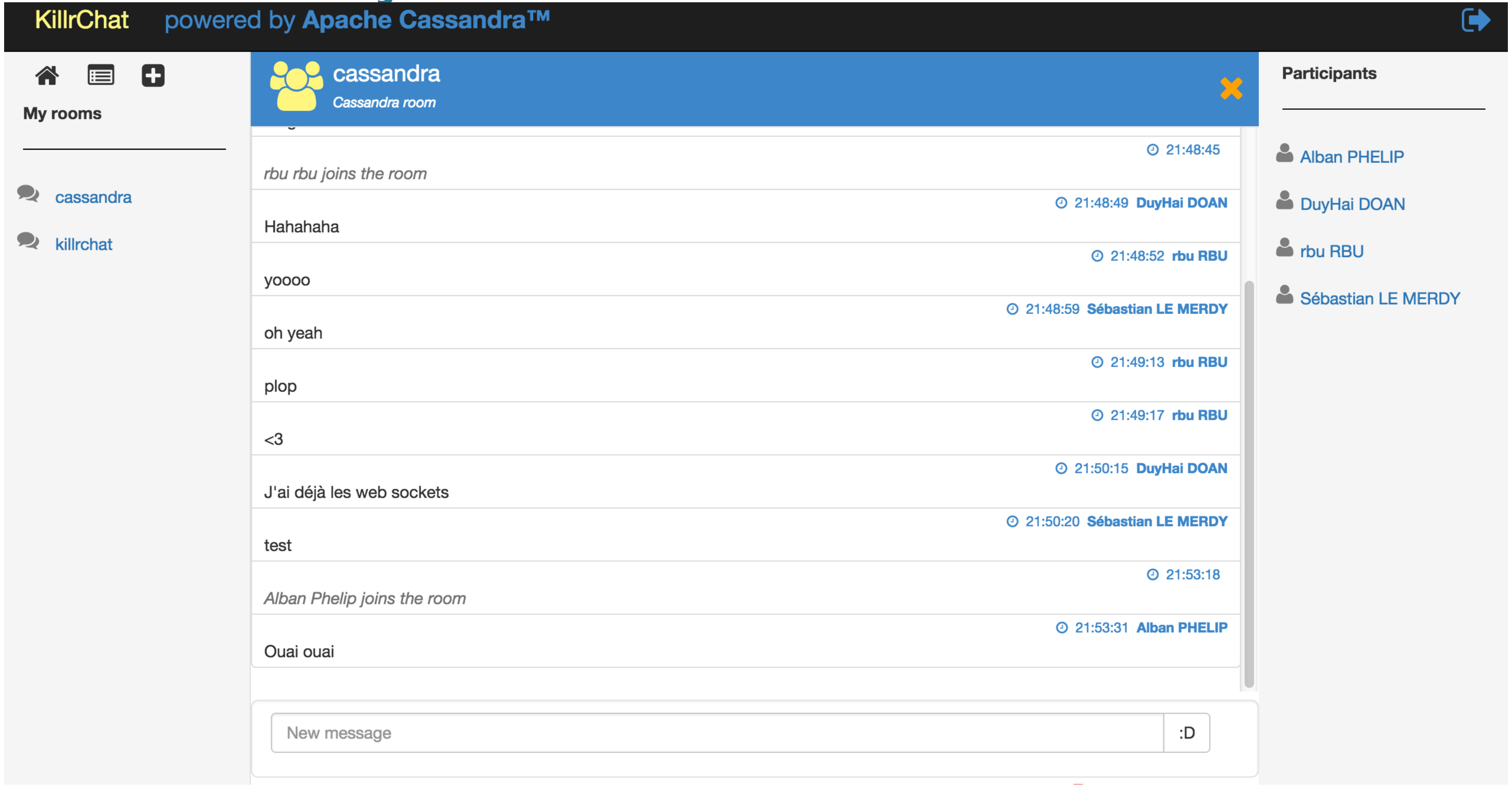
Why **KillrChat** ?

- show real life de-normalization
- DIY exercise
- provide real application for attendees
- highlight Cassandra eco-system

Technology stack



Front end layout



21:48:45
rbu rbu joins the room

21:48:49 **DuyHai DOAN**
Hahahaha

21:48:52 **rbu RBU**
yoooo

21:48:59 **Sébastian LE MERDY**
oh yeah

21:49:13 **rbu RBU**
plop

21:49:17 **rbu RBU**
<3

21:50:15 **DuyHai DOAN**
J'ai déjà les web sockets

21:50:20 **Sébastian LE MERDY**
test

21:53:18
Alban Phelip joins the room

21:53:31 **Alban PHELIP**
Ouai ouai

Exercises outline

TDD style

Implement the services to make tests green

Glue-code and front-end code provided

Getting started

Clone the Git repository

```
git clone https://github.com/doanduyhai/killrchat.git
```

Go into the *'killrchat'* folder and launch tests

```
cd killrchat
mvn clean test
```

Exercise 1

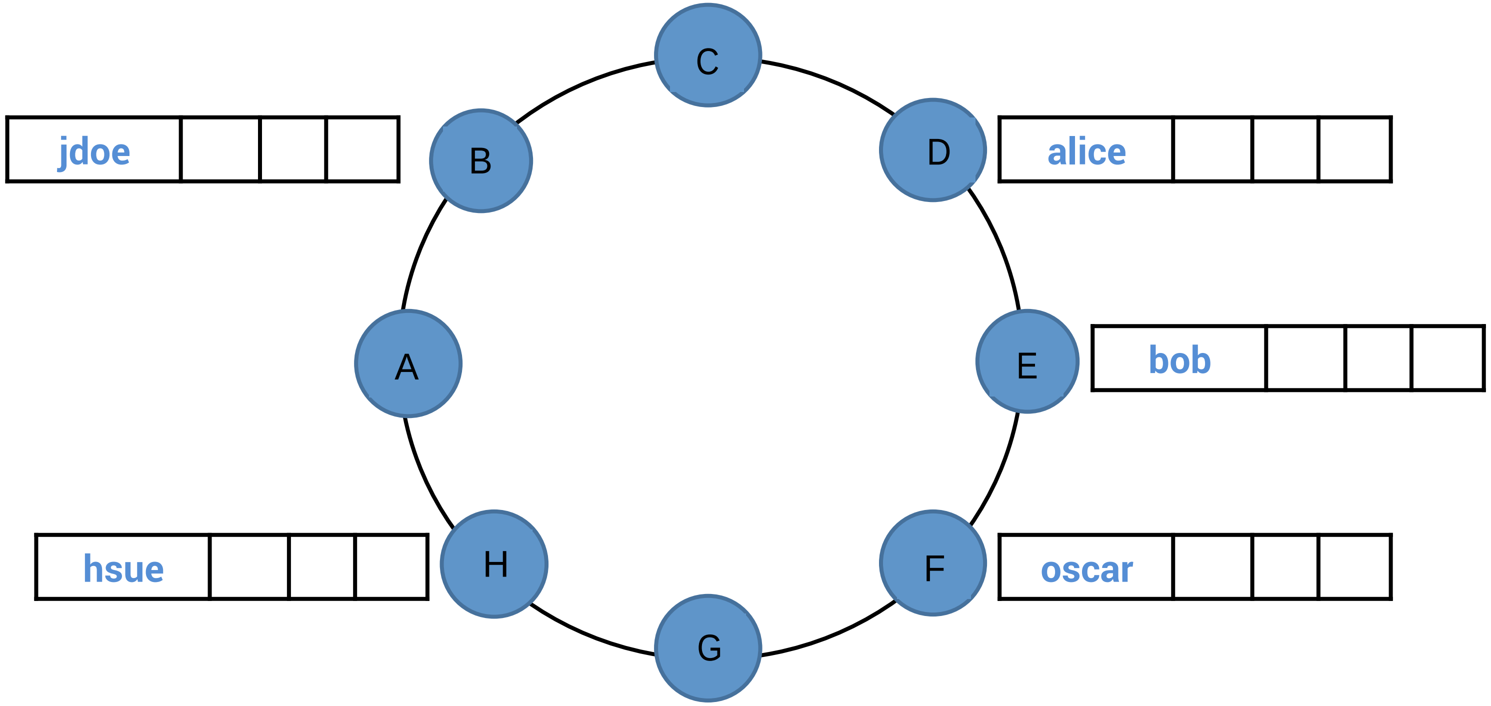
User account management

Specifications

```
git checkout exercise_1_specs
```

Scalability

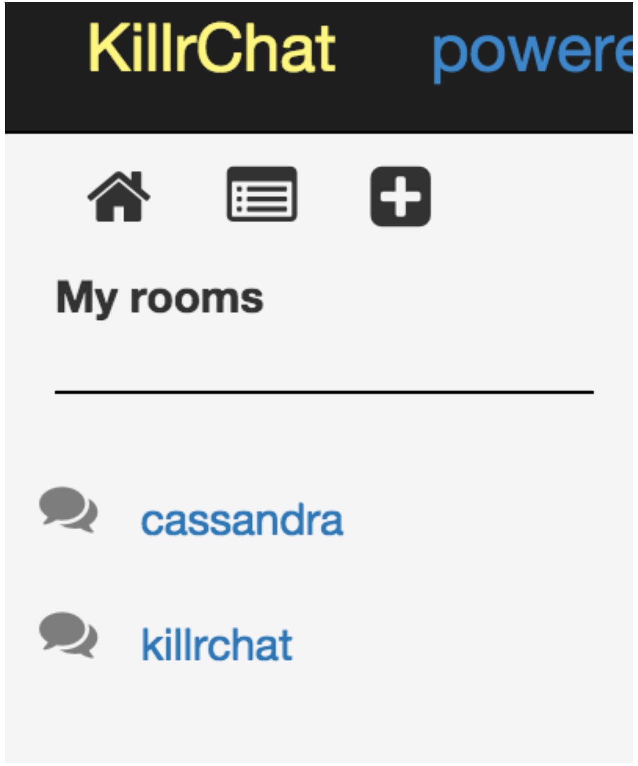
Scaling by login



Data model

```
CREATE TABLE killrchat.users(
        login text,
        pass text,  //password is not allowed because reserved word
        lastname text,
        firstname text,
        bio text,
        email text,
        chat_rooms set<text>,
PRIMARY KEY(login));
```

User’s chat rooms



User's chat rooms data model

How to store chat rooms for an user ?

```
CREATE TABLE killrchat.user_rooms(
        login text,
        room_name text,
PRIMARY KEY((login), room_name));
```

- pros: can store huge room count per user (10^6)
- cons: separated table, needs 1 extra SELECT

User’s chat rooms data model

Best choice

```
CREATE TABLE killrchat.users(
        login text,
        …
        chat_rooms set<text>, //list of chat rooms for this user
PRIMARY KEY(login));
```

- 1 SELECT fetches all data for a given user
- usually, 1 user is not in more that **1000** rooms at a time
- stores only room name

Lightweight Transaction

Avoid creating the same login by 2 different users ?

☞ use Lightweight Transaction

```
INSERT INTO killrchat.users(room_name, …)
VALUES (‘jdoe’, …) IF NOT EXISTS ;
```

Expensive operation

☞ do you create a new account every day ?

Let's code!

Tasks

- annotate UserEntity
- implement UserService

Solution

```
git checkout exercise_1_solution
```

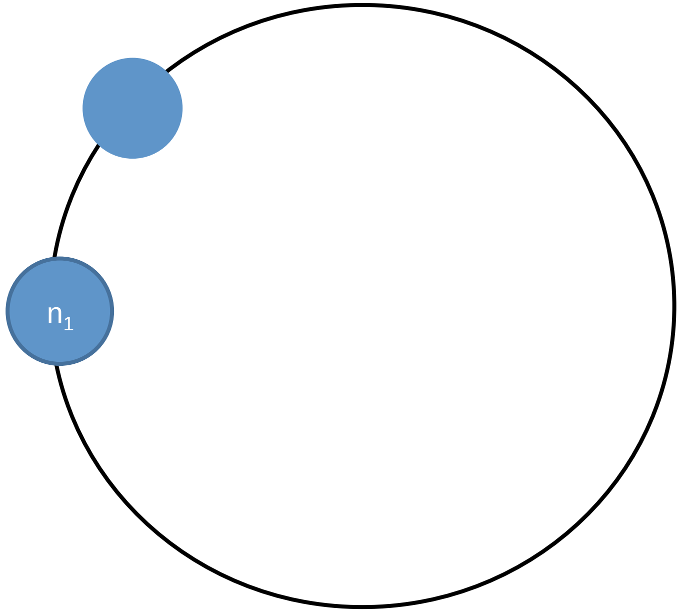
Exercise 2

Chat room management

Specifications

git checkout *exercise_2_specs*

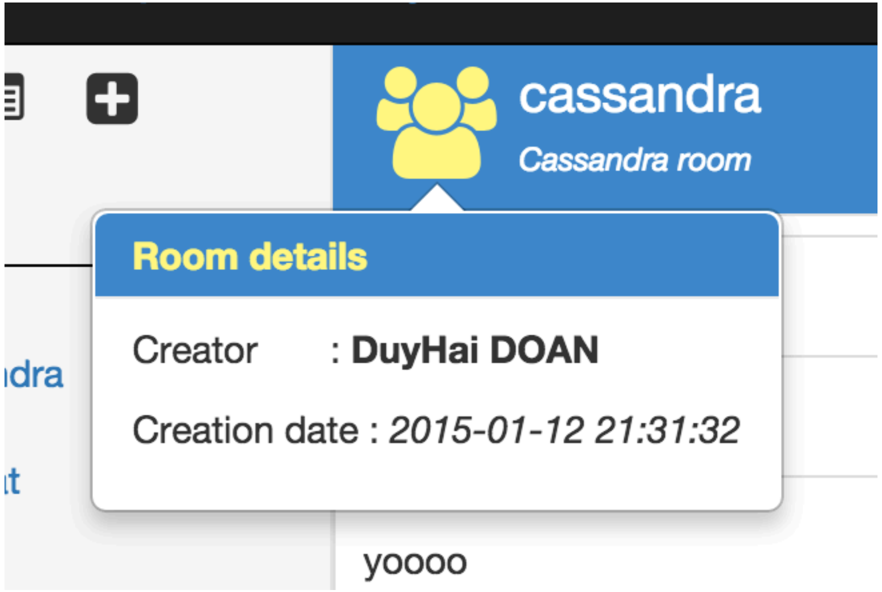
Scalability



Data model

```
CREATE TABLE killrchat.chat_rooms(
        room_name text,
        name text,                       // same as room_name
        creation_date timestamp,
        banner text,
        creator text,                    // de-normalization
        creator_login text,
        participants set<text>,          // de-normalization
PRIMARY KEY(room_name));
```

Room details



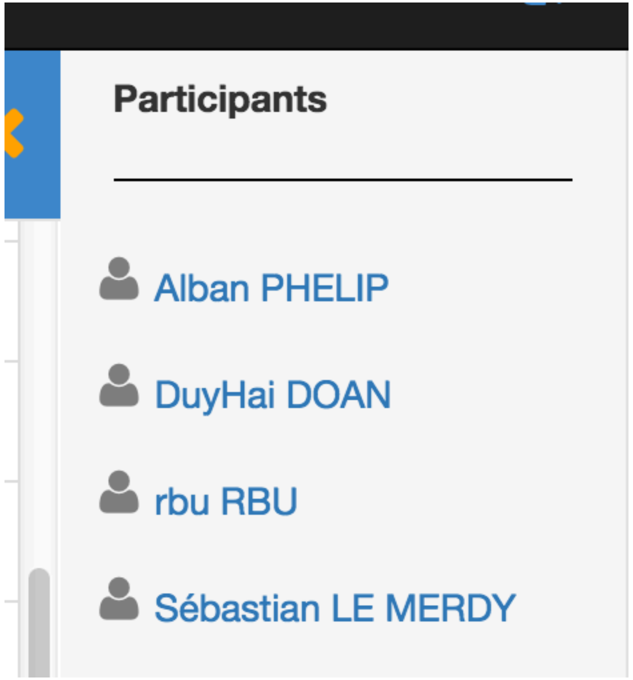
Room details

Creator : **DuyHai DOAN**

Creation date : *2015-01-12 21:31:32*

yoooo

Room participants


Participants

Alban PHELIP

DuyHai DOAN

rbu RBU

Sébastian LE MERDY

De-normalization

```
CREATE TABLE killrchat.chat_rooms(
        room_name text,
        …
        creator text,                   // JSON blob {login: …, firstname: …, lastname: …}
        …
        participants set<text>,         // JSON blob {login: …, firstname: …, lastname: …}
PRIMARY KEY(room_name));
```

Lightweight Transaction

Avoid creating the same room by 2 different users ?

☞ use Lightweight Transaction

```
INSERT INTO killrchat.chat_rooms(room_name, …)
VALUES (‘games’, …) IF NOT EXISTS ;
```

Listing all rooms

How to list all existing rooms ?

- limit to first 100 rooms
- rooms ordered by their token (hash of room_name)

Full text search ?

- possible with *'gam*'* sematics
- Lucene integration otherwise (DSE)

Let’s code!

DATASTAX

Tasks

- **ChatRoomEntity** already given with proper annotations
- Implement first methods in **ChatRoomService**

Solution

```
git checkout exercise_2_solution
```

Exercise 3

Participants management

Room deletion

Specifications

```
git checkout exercise_3_specs
```

Participant joining

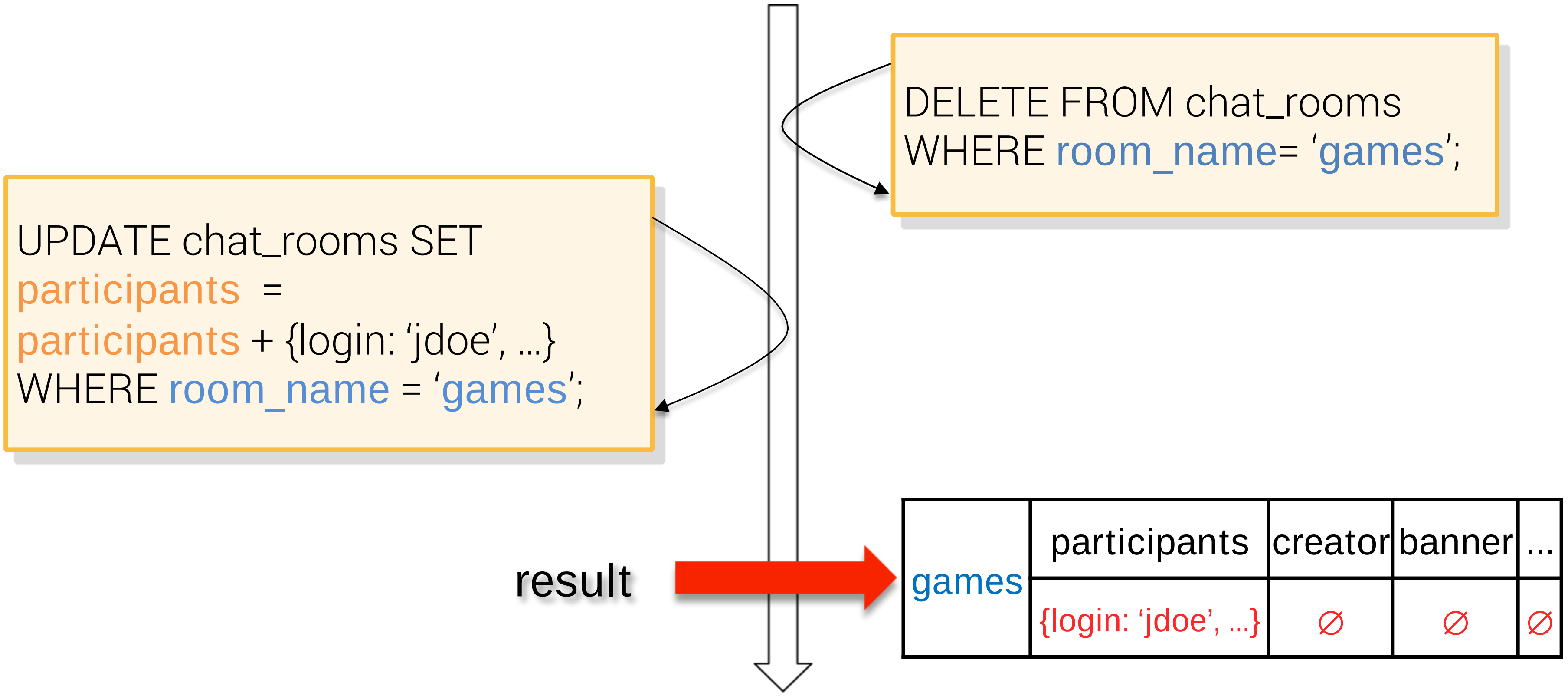
Adding new participant

```
UPDATE killrchat.chat_rooms SET participants  = participants + {…}
WHERE room_name = 'games';
```

❓

What if the creator deletes the room at the same time ?

Concurrent delete/update



```
DELETE FROM chat_rooms
WHERE room_name= ‘games’;
```

```
UPDATE chat_rooms SET
participants  =
participants + {login: ‘jdoe’, …}
WHERE room_name = ‘games’;
```

	participants	creator	banner	...
games	{login: ‘jdoe’, …}	∅	∅	∅

Participant joining

Solution

☞ use Lightweight Transaction

```
UPDATE killrchat.chat_rooms SET participants  = participants + {…}
WHERE room_name = 'games' IF name = 'games';
```

Why not use **UPDATE … IF EXISTS** ?

Participant joining

Solution

☞ use Lightweight Transaction

```
UPDATE killrchat.chat_rooms SET participants  = participants + {…}
WHERE room_name = 'games' IF name = 'games';
```

Why not use **UPDATE … IF EXISTS** ?

- syntax not yet available
- use column **name** = **room_name** = **partition key** as condition
- see https://issues.apache.org/jira/browse/CASSANDRA-8610

Participant leaving

Removing participant (no **read-before-write**)

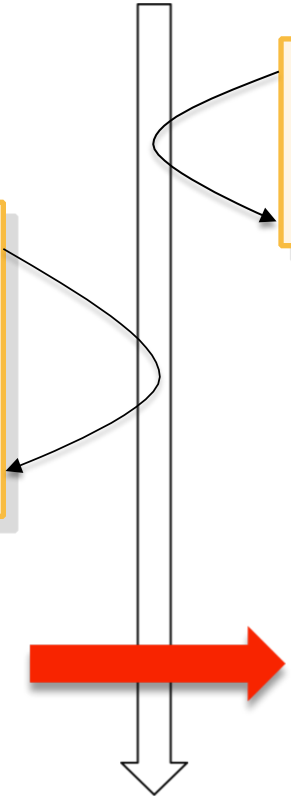
```
UPDATE killrchat.chat_rooms SET participants  = participants - {…}
WHERE room_name = ‘games’;
```

❓

What if the creator deletes the room at the same time ?

- we’ll create a tombstone
- tombstone will be garbage-collected by compaction

Concurrent delete/update



```
DELETE FROM chat_rooms
WHERE room_name= 'games';
```

```
UPDATE chat_rooms SET
participants  =
participants - {login: 'jdoe', ...}
WHERE room_name = 'games';
```

result

	participants	creator	banner	...
games	∅	∅	∅	∅

Deleting room

What if participant leaving at the same time ?

- not a problem, tombstone will be garbage

What if participant joining at the same time ?

☞ use Lightweight Transaction

Only room creator can delete room, no one else!

☞ use Lightweight Transaction

Deleting room

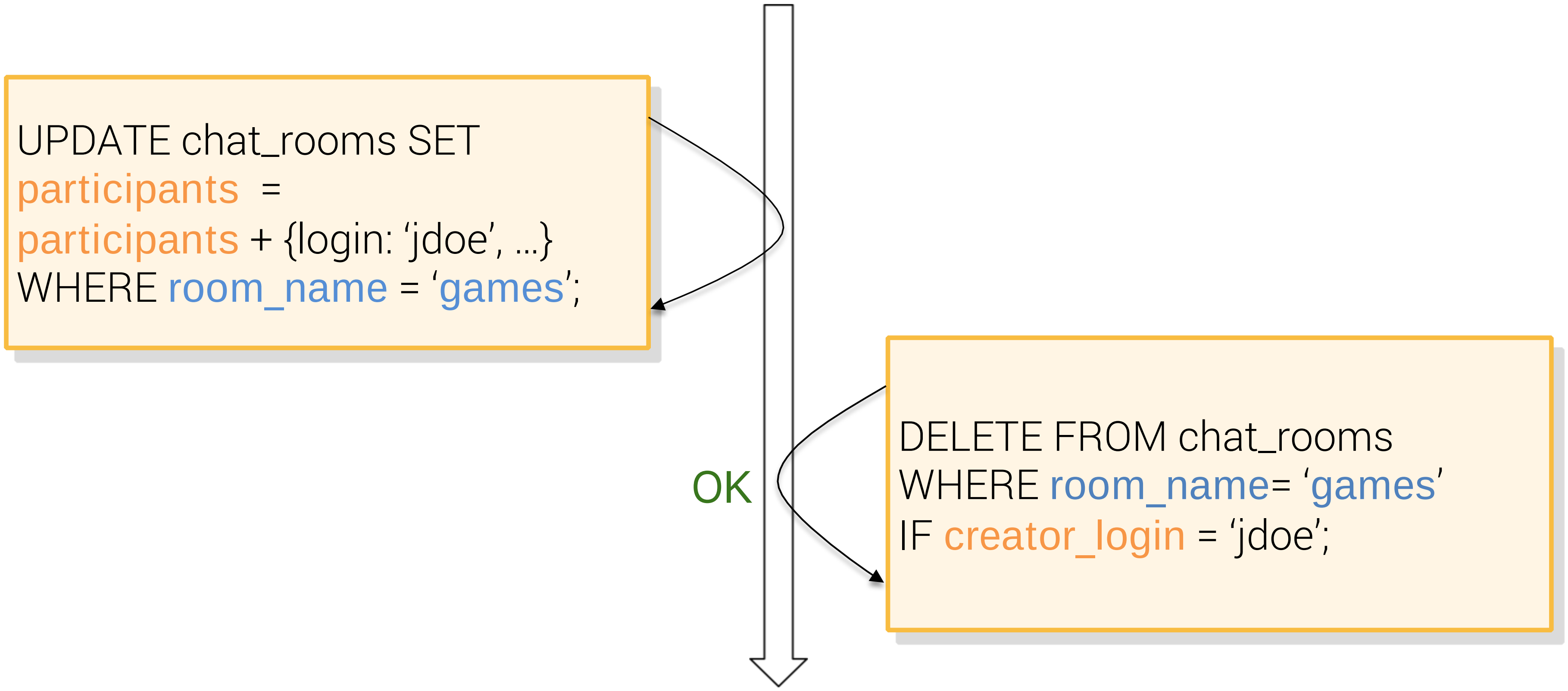
Solution

```
DELETE killrchat.chat_rooms
WHERE room_name = ‘games’
IF creator_login = <current_user_login>;
```

Advantages

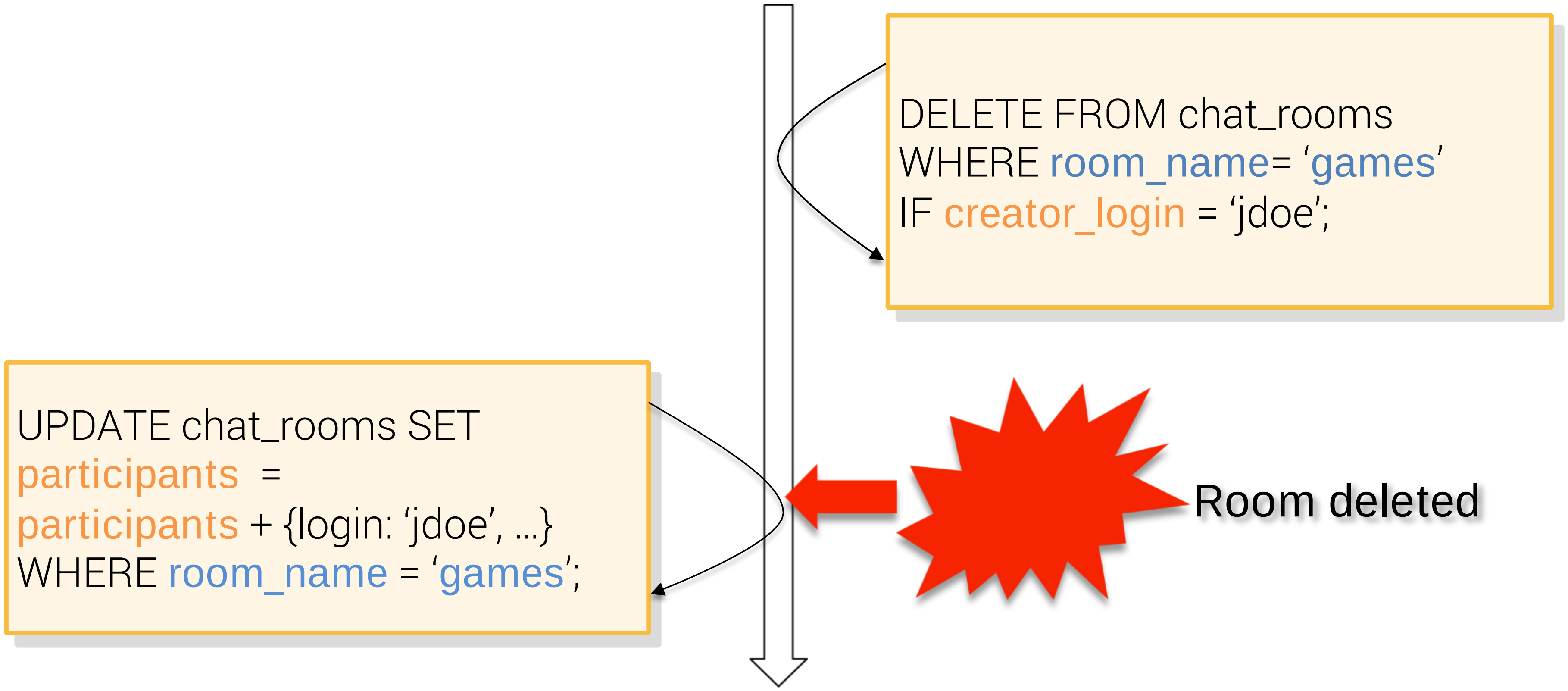
- current user login coming from Security context, no cheating !
- slow but **how often do you delete rooms ?**

Concurrent delete/update


```
UPDATE chat_rooms SET
participants  =
participants + {login: ‘jdoe’, …}
WHERE room_name = ‘games’;
```

```
DELETE FROM chat_rooms
WHERE room_name= ‘games’
IF creator_login = ‘jdoe’;
```

Concurrent delete/update



```
DELETE FROM chat_rooms
WHERE room_name= 'games'
IF creator_login = 'jdoe';
```

```
UPDATE chat_rooms SET
participants  =
participants + {login: 'jdoe', …}
WHERE room_name = 'games';
```

Let’s code!

Tasks

- Implement remaining methods in **ChatRoomService**

Solution

```
git checkout exercise_3_solution
```

Exercise 4

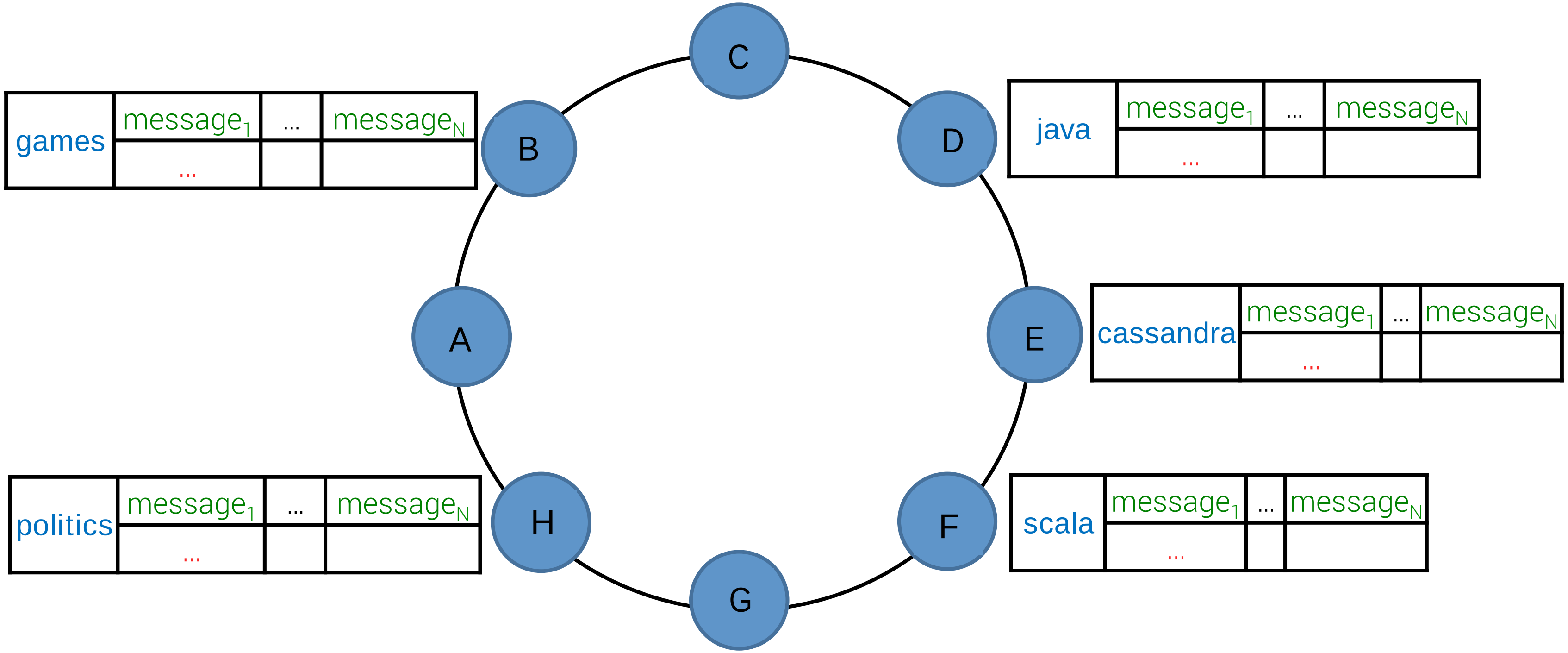
Chat messages management

Specifications

```
git checkout exercise_4_specs
```

Scalability

Scaling messages by room name



Data model

```
CREATE TABLE killrchat.chat_room_messages(
        room_name text,
        message_id timeuuid,
        content text,
        author text,                    // JSON blob {login: …, firstname: …, lastname: …}
        system_message boolean,
        PRIMARY KEY((room_name), message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);
```

Data model

Clustering column message_id order by DESC

- latest messages first
- leverage the new row cache in Cassandra 2.1

Improvements

- current data model limits messages count to ≈ 500×10^6
- bucketing by day is the right design

```
PRIMARY KEY((room_name, day), message_id) //day format yyyyMMdd
```

Let’s code!

Tasks

- **MessageEntity** already given with proper annotations
- Implement methods in **MessageService**

Solution

```
git checkout exercise_4_solution
```

DATASTAX

Q & R

Thank You

@doanduyhai

duy_hai.doan@datastax.com

https://academy.datastax.com/