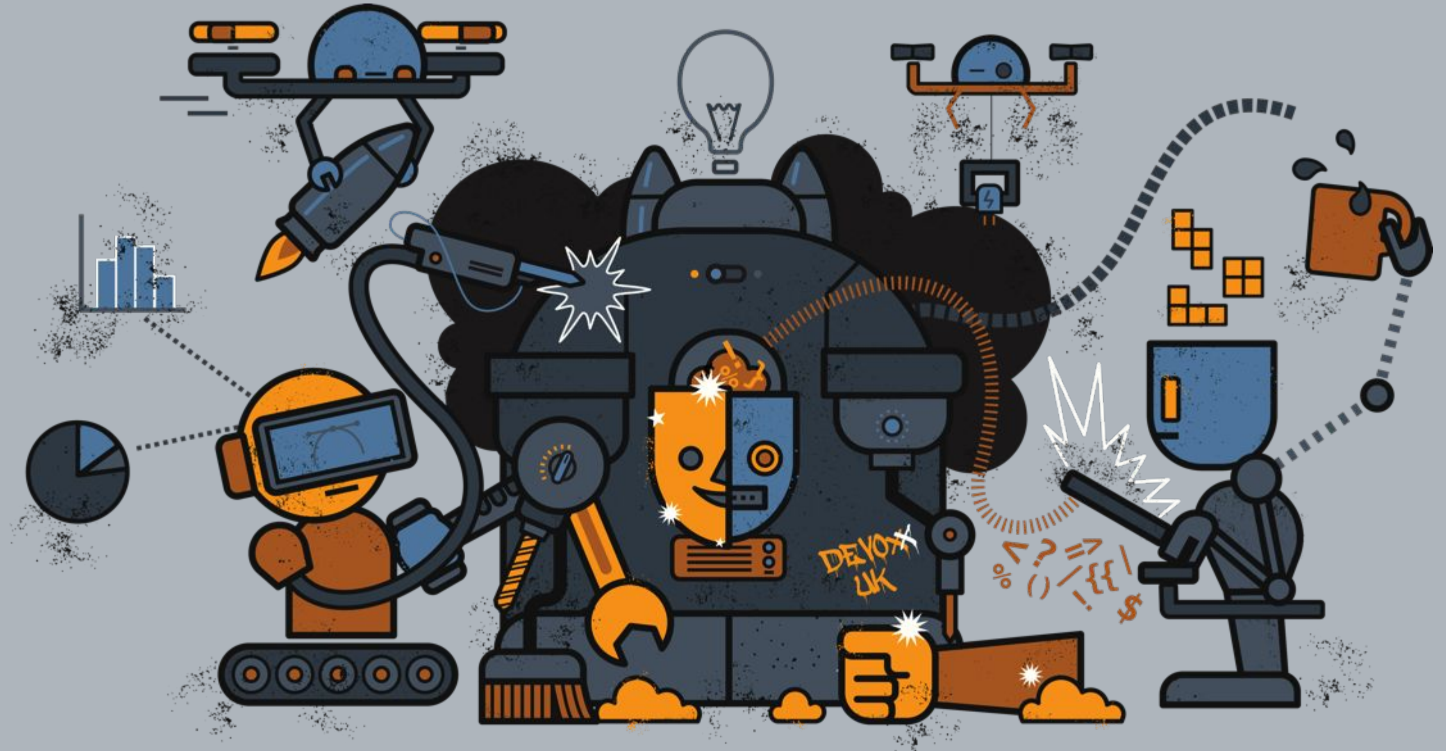
Transactional Actors using STM and Eclipse Vert.x

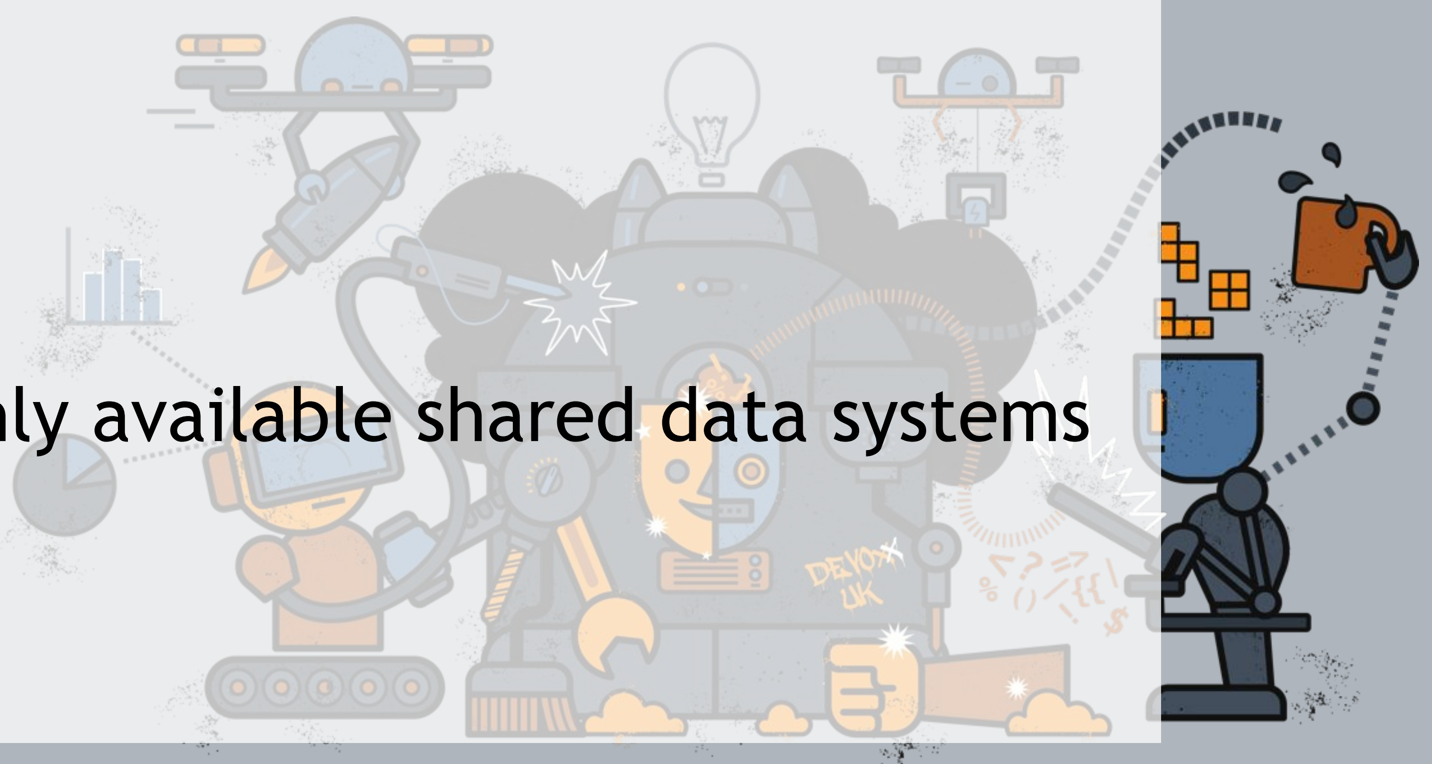
Michael Musgrove
Narayana Team, Red Hat, Inc.
Mark Little
VP, Red Hat, Inc.
May 2017

DEVOXX United Kingdom



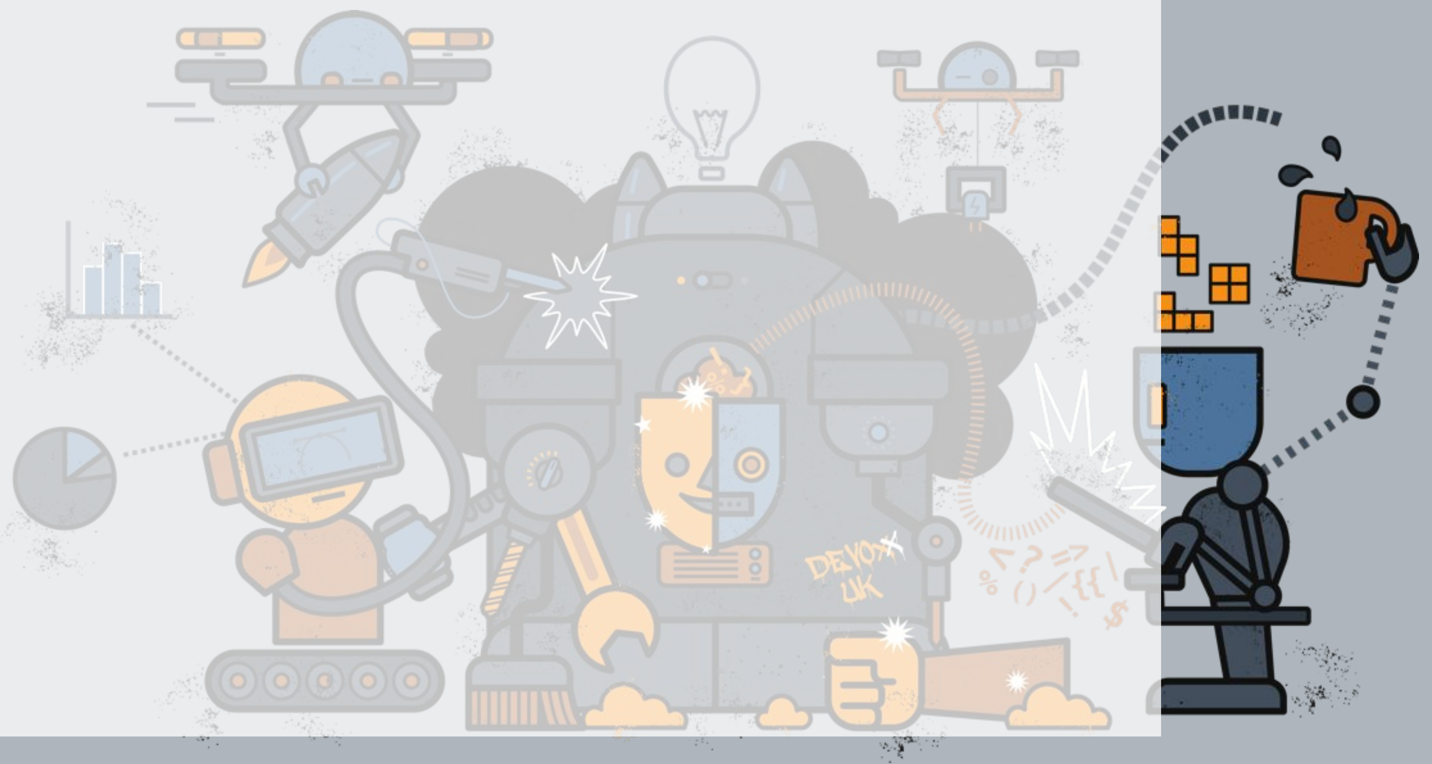
Agenda

- **The Actor Model**
- Transactions and Software Transactional Memory (STM)
- Eclipse Vert.x
- Narayana STM
- Code Walkthrough
- Combining Narayana STM with Vert.x to create highly available shared data systems
First Column



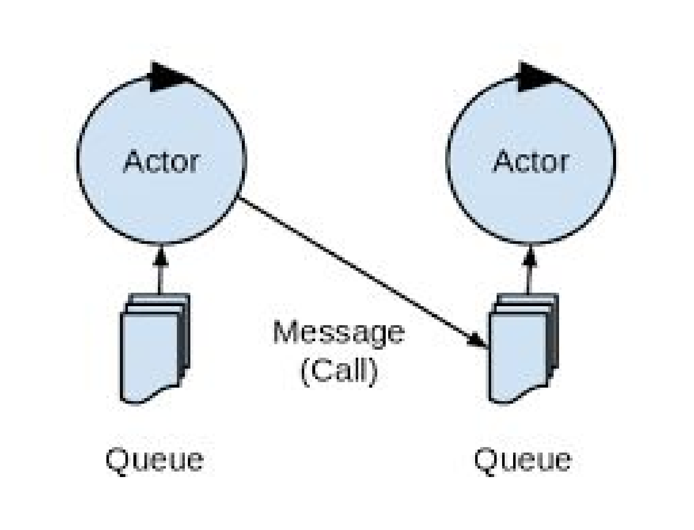
What this talk is NOT

- Not a survey of actors
- Not a survey of transactions
- Not a tutorial on Vert.x
- Not a tutorial on Narayana



The Actor Based Programming Model

- Actors and CSP have been around for decades
  - CSP from Hoare, 1985
  - Actor model from Hewitt et al, 1973
- But popular ways to model primitives for concurrent computations
- Distributed computations communicate via message passing
  - No shared state

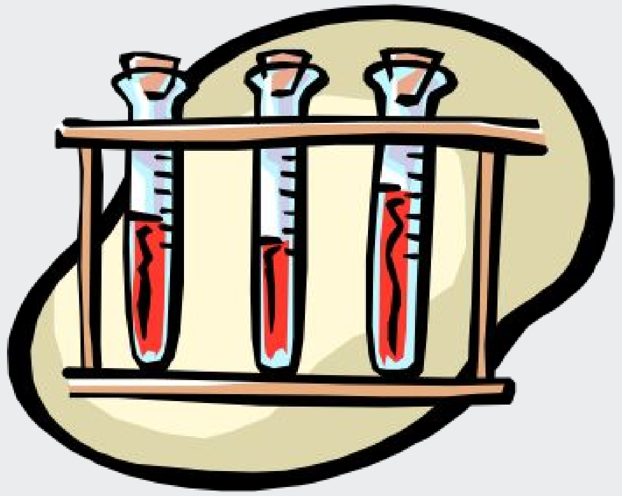


Agenda

- The Actor Model
- **Transactions and Software Transactional Memory (STM)**
- Eclipse Vert.x
- Narayana STM
- Code Walkthrough
- Combining Narayana STM with Vert.x to create highly available shared data systems

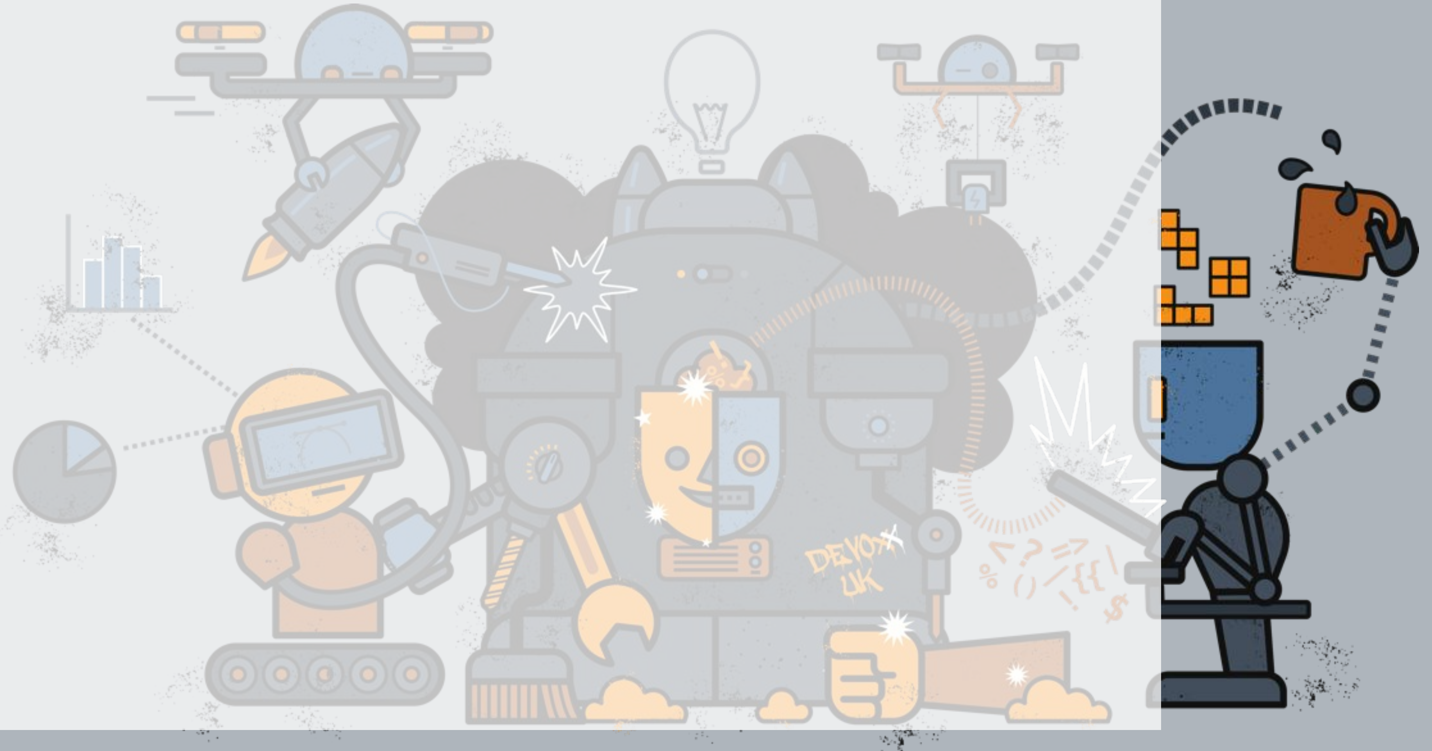
(Distributed) Transactions

- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Two-phase commit
  - Required when there is more than one resource manages (RM) in a transaction
  - Managed by the transaction manager (TM)
  - Uses a familiar two-phase technique:



(Distributed) Transactions

- Durability
  - When a transaction commits, its results must survive failures
    - Must be durably recorded prior to commit
    - System waits for disk ack before acking user
  - If a transaction rolls back, changes are undone
- Isolation
  - Programs can be executed concurrently
    - BUT they appear to execute serially

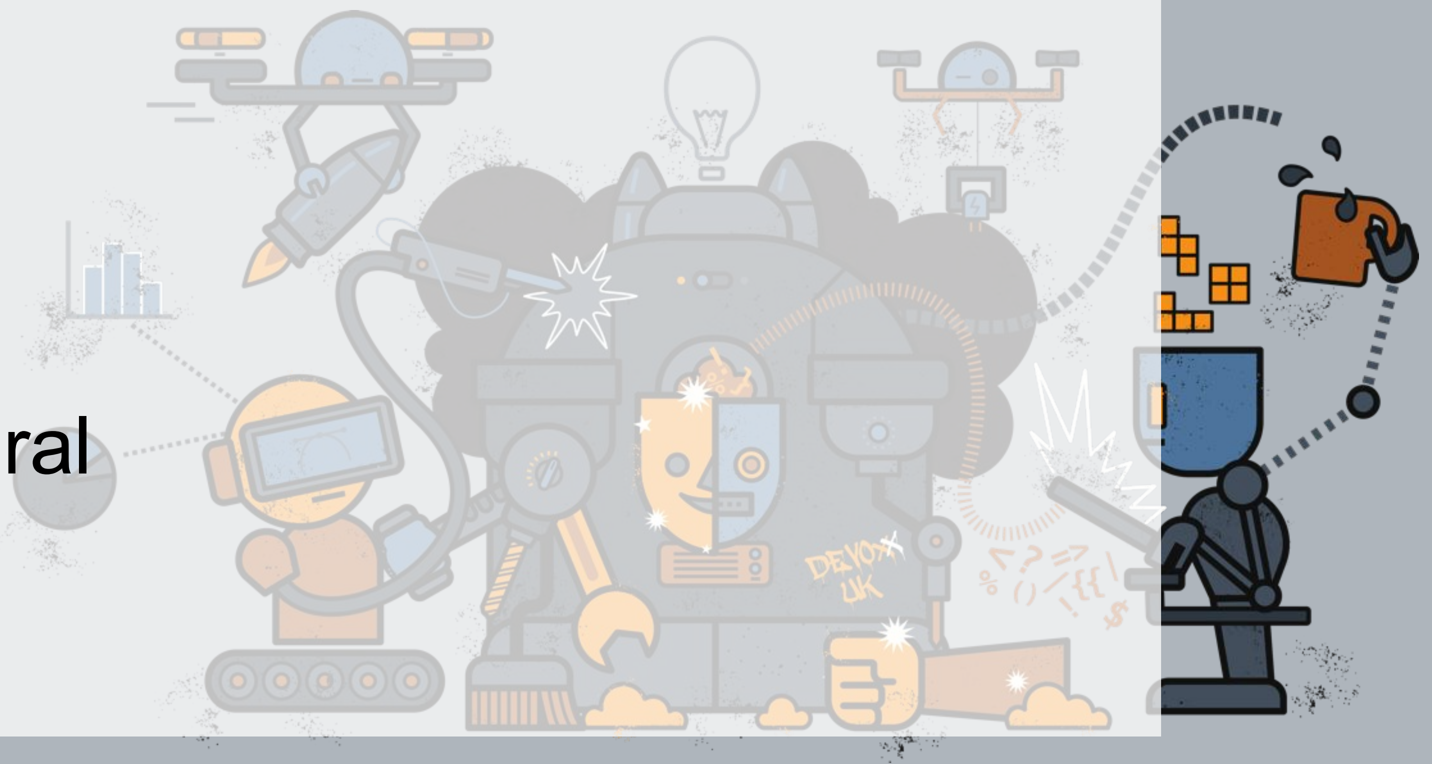


Software Transactional Memory

- Hardware Transactional Memory around since the 1980’s
  - An alternative to lock-based synchronization
- Software Transactional Memory (STM) proposed in 1995
  - Still an active area of research
- STM is about ease of use and reliability
  - Access shared state, either for reading of writing, within atomic blocks
  - All code inside an atomic block executes as if there were no other threads
  - Some implementations can be lock free (optimistic vs pessimistic, timestamp)

Transactions and Actors

- A stateful actor may go through multiple state transactions upon receipt of a message
  - Actors share state through message passing
- Computational failures may occur
- Hardware and software failures may occur
- Consistency of state important
- Composition of actors
- The combination of STM and Actors is fairly natural

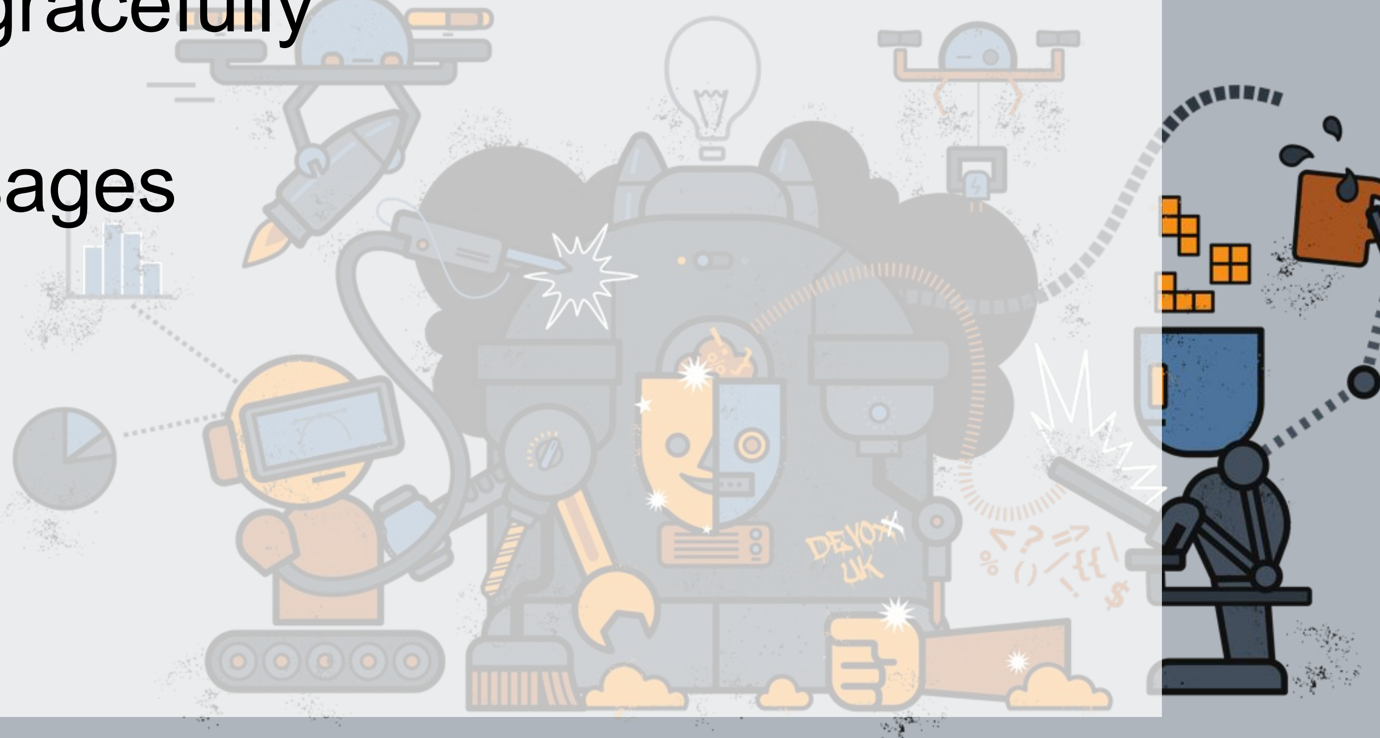


Agenda

- The Actor Model
- Transactions and Software Transactional Memory (STM)
- **Eclipse Vert.x**
- Narayana STM
- Code Walkthrough
- Combining Narayana STM with Vert.x to create highly available shared data systems

Reactive Systems

- **Responsive** - they respond in an acceptable time
- **Elastic** - they scale up and down
- **Resilient** - they are designed to handle failures gracefully
- **Asynchronous** - they interact using async messages
- http://www.reactivemanifesto.org/

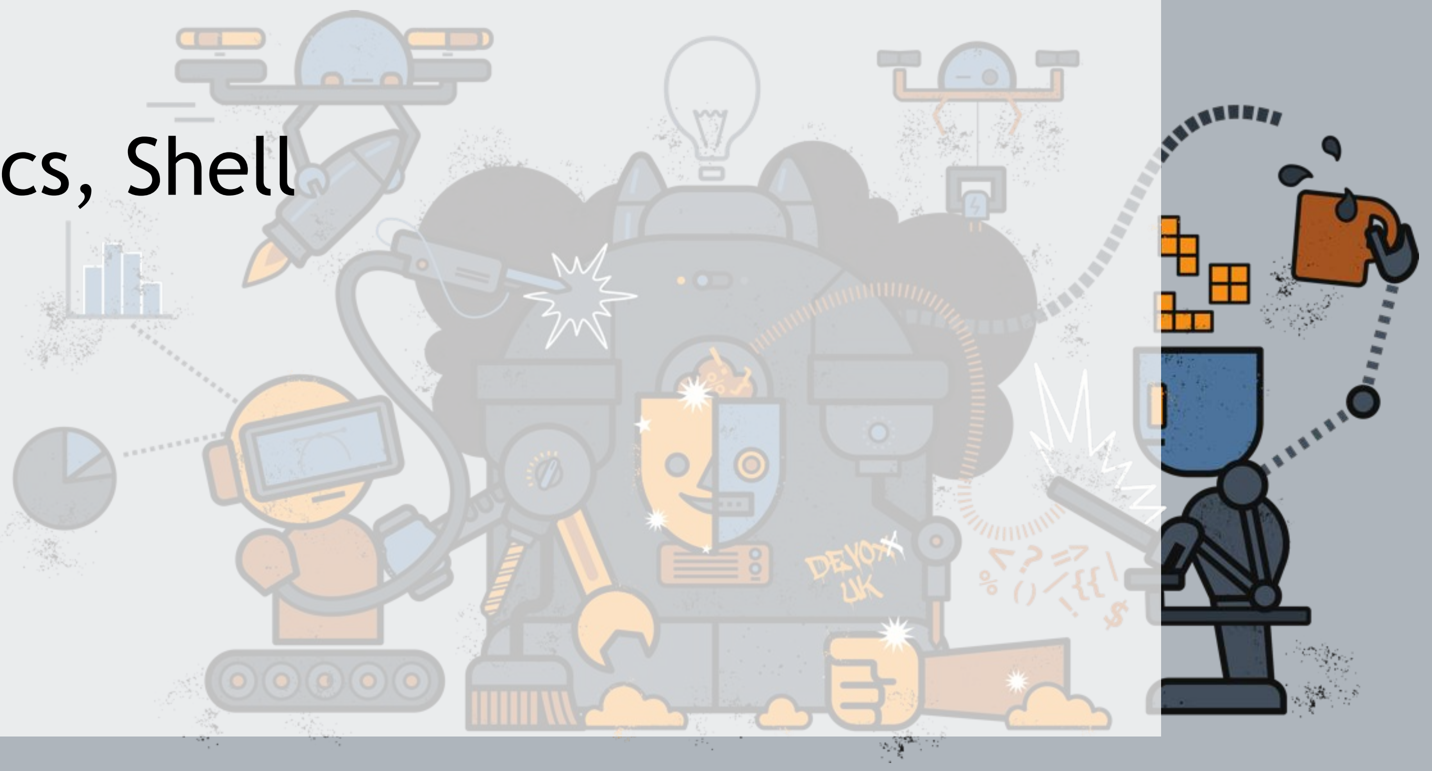


Eclipse Vert.x

- Non-blocking (asynchronous)
- No locking, no concurrency control problems
- Event bus (reactor pattern)
- Actors
- Polyglot JVM
- Added AMQ and Qpid Dispatch Router
- Infinispan/JDG
- Adding transactions and STM
- Spring Boot
- Large user base and community of contributors

What does Vert.x provide?

- TCP, UDP, HTTP 1 & 2 servers and clients (non-blocking) DNS client
- Clustering
- Event bus (messaging)
- Distributed data structures
- (built-in) Load-balancing
- (built-in) Fail-over
- Pluggable service discovery, circuit-breaker Metrics, Shell



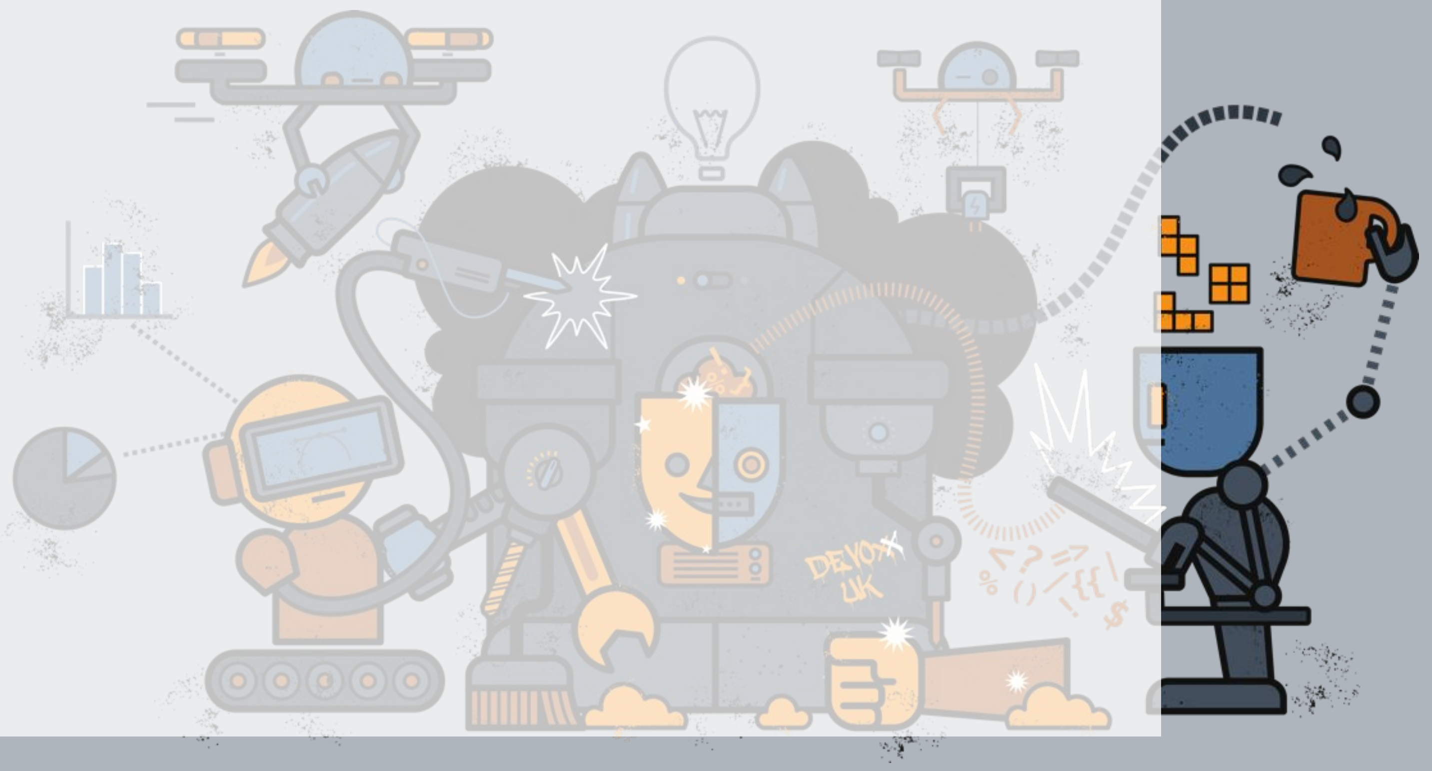
DEVOXX United Kingdom

Agenda

- The Actor Model
- Transactions and Software Transactional Memory (STM)
- Eclipse Vert.x
- **Narayana STM**
- Code Walkthrough
- Combining Narayana STM with Vert.x to create highly available shared data systems

Narayana STM

- Based on Narayana transaction project
- Define state (objects) which can be manipulated within transactions
  - Volatile (recoverable) and persistent (durable)
- Pessimistic and Optimistic models
- Different variants of transactions
  - Top level
  - Nested
  - Nested top level
- Modularity
  - Transaction context on threads
- Annotations



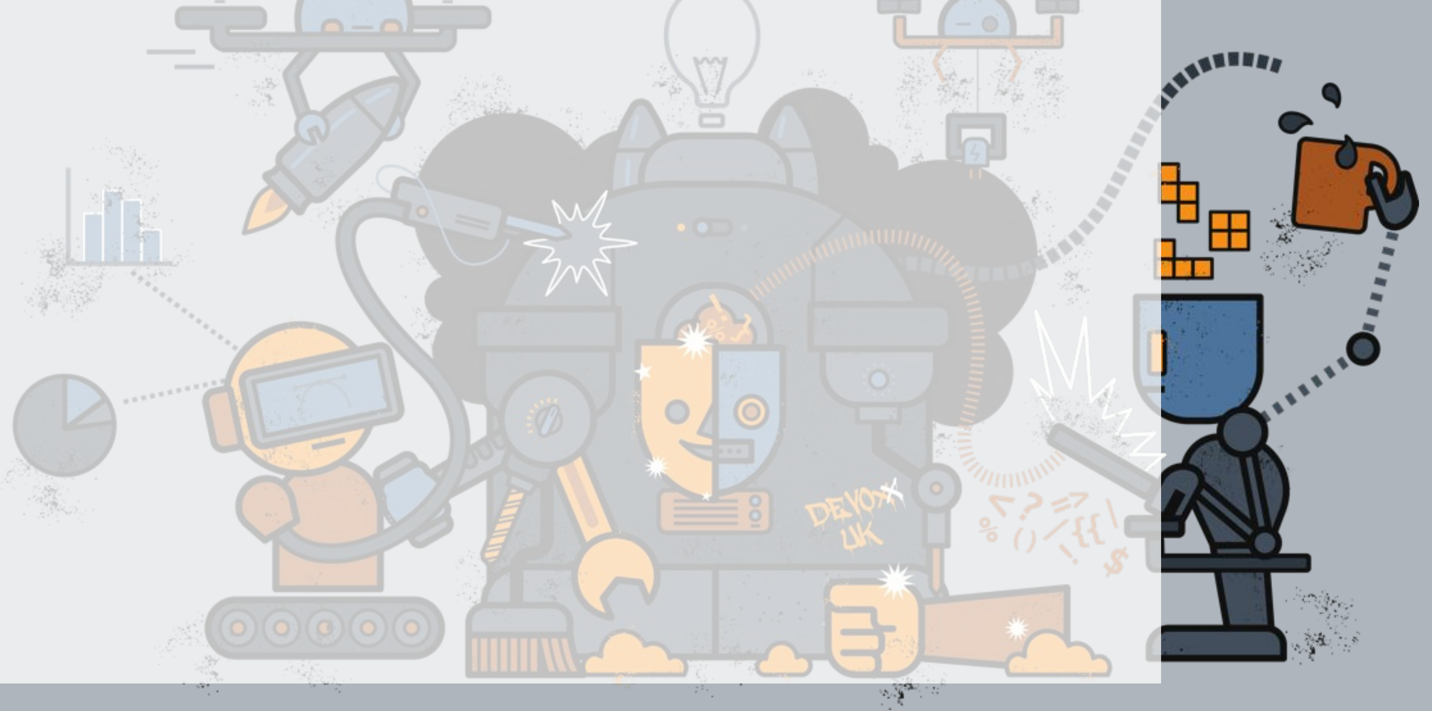
DEVOXX United Kingdom™

STM containers

```java
public class Container<T>
{
  public enum TYPE { RECOVERABLE, PERSISTENT };
  public enum MODEL { SHARED, EXCLUSIVE };

  public Container ();

  public synchronized T create (T member);

  public static final Container<?> getContainer (Object proxy);
}
```

STM Annotations

- @Transactional
  - Implementations of interface are container managed
- @Nested & @NestedTopLevel
  - Container will create a new transaction for each method
- @Optimistic & @Pessimistic
- @ReadLock & @WriteLock
- @State & @NotState
- @TransactionFree

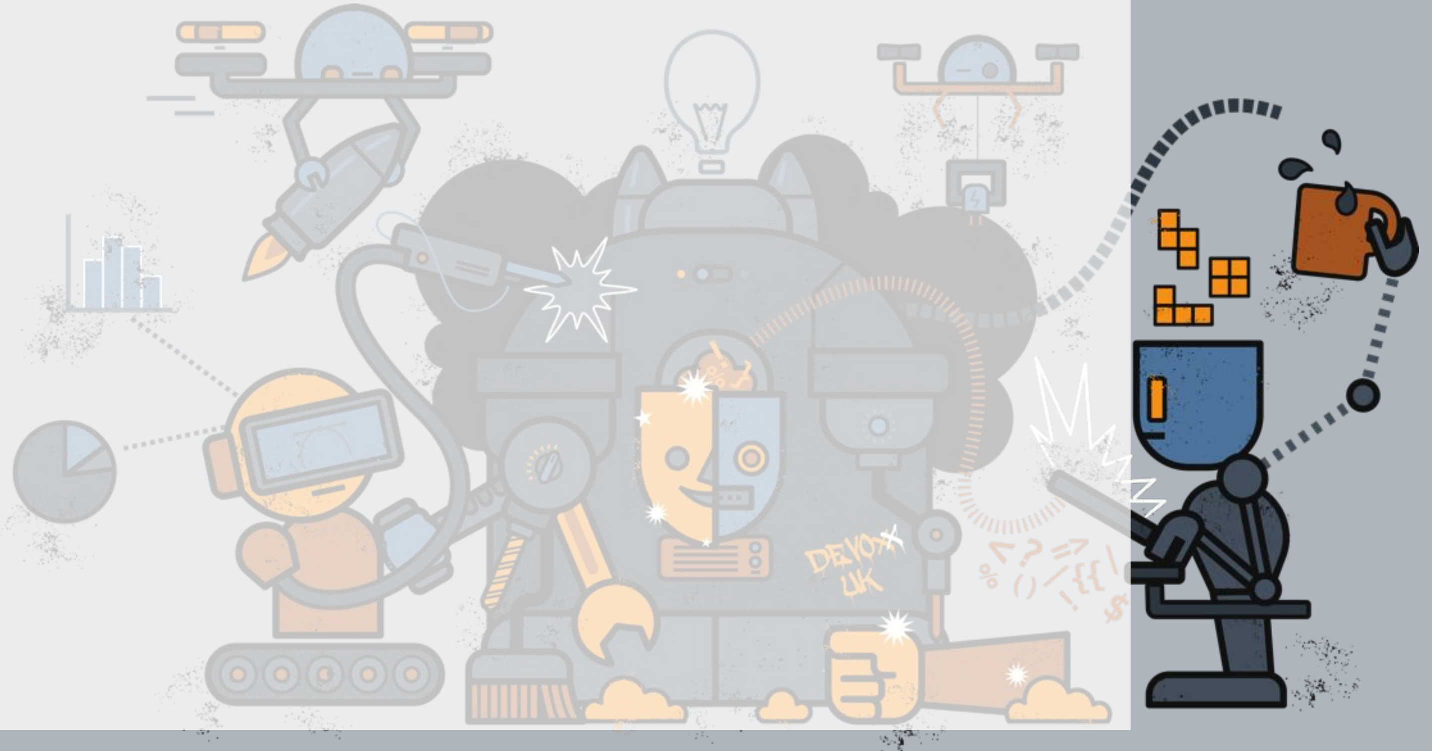


Agenda

- The Actor Model
- Transactions and Software Transactional Memory (STM)
- Eclipse Vert.x
- Narayana STM
- **Code Walkthrough**
- Combining Narayana STM with Vert.x to create highly available shared data systems

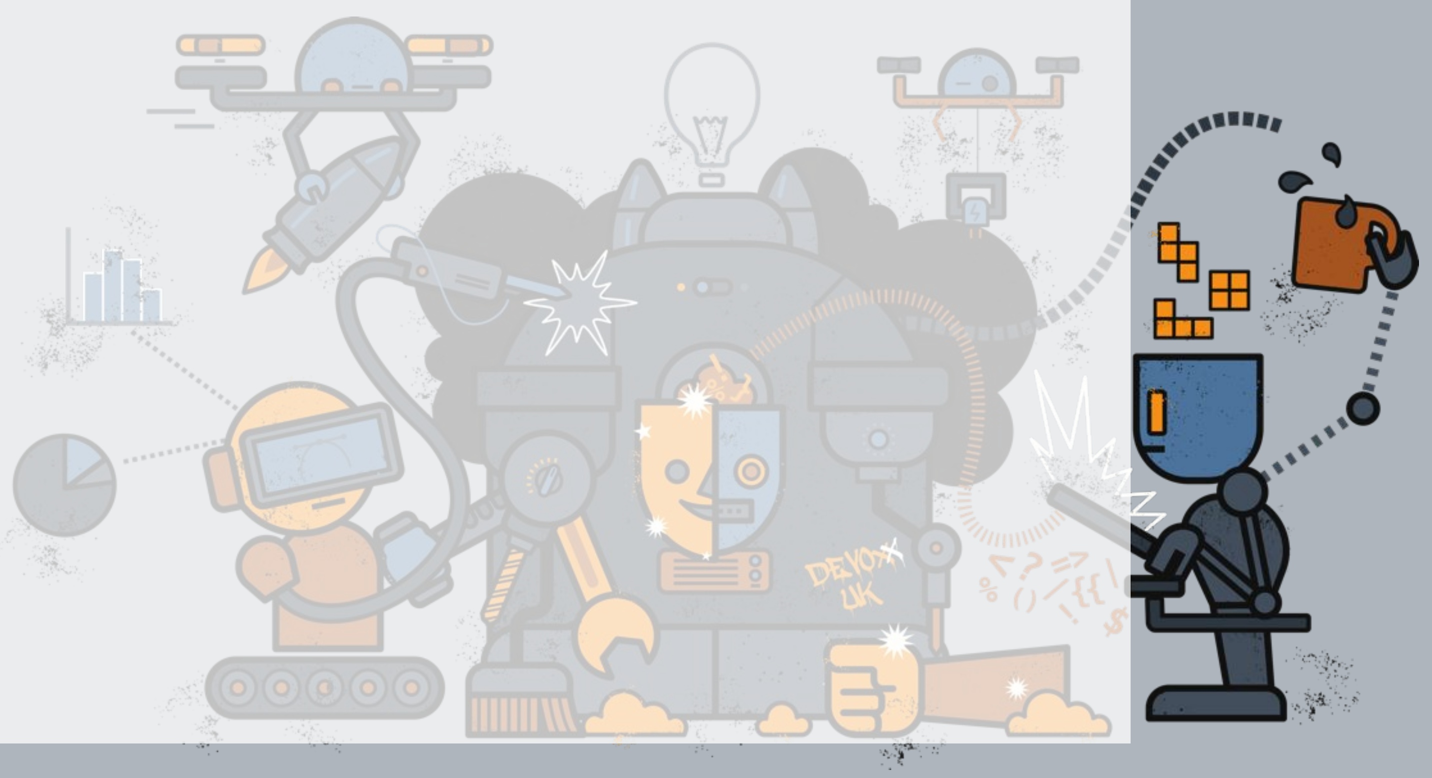
An STM Example: Start out with an Interface

```
public interface StockLevel {
   public int get () throws Exception;
   public void set (int value) throws Exception;
   public void decr (int value) throws Exception;
}
```



Example: Implement the Interface

```
public class StockLevelImpl implements StockLevel {

  public int get () throws Exception {
     return state;
  }

  public void set (int value) throws Exception {
     state = value;
  }

  public void decr (int value) throws Exception {
     state -= value;
  }

  private int state;
}
```

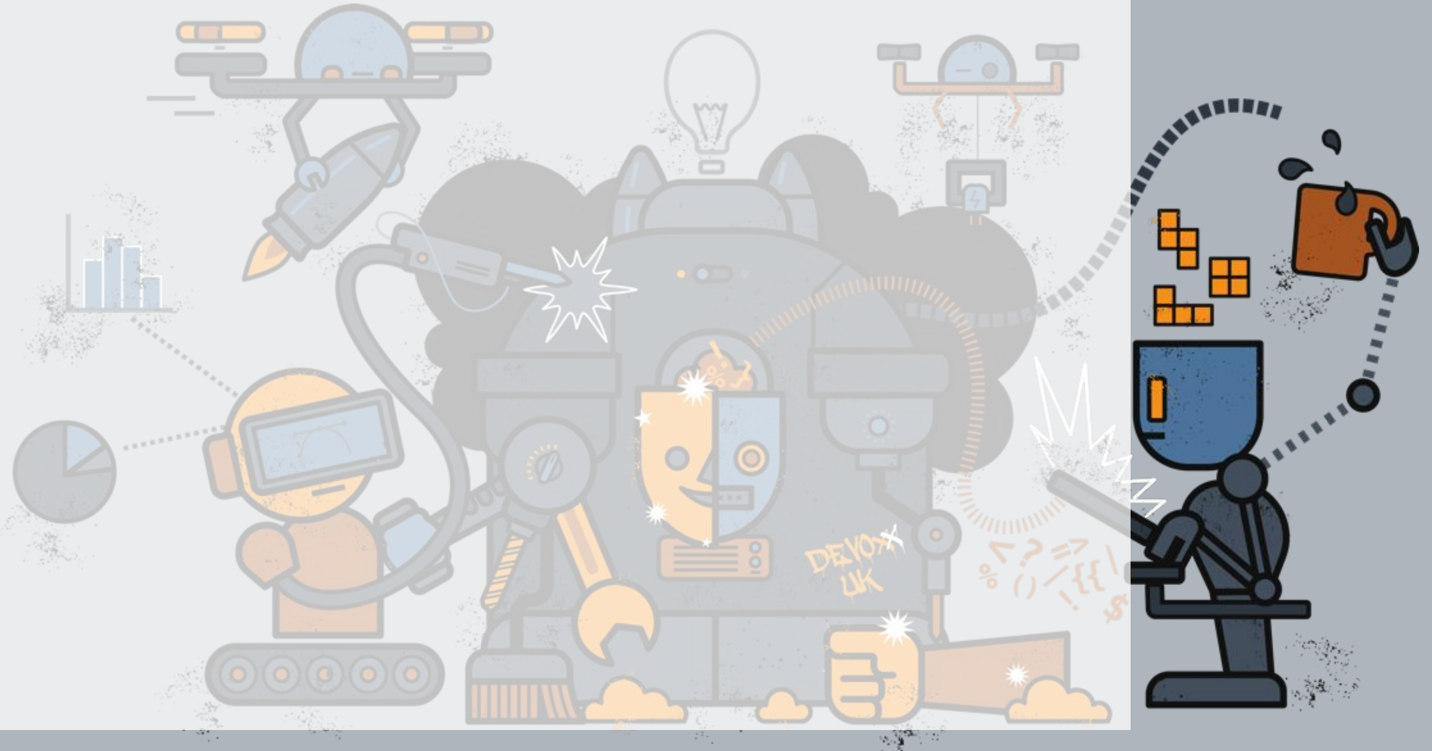


Example: Make the interface transactional

- Transactional objects must be instrumented so that the underlying STM implementation is able to differentiate them from non-transactional objects. Do this by annotating the interface with:

```
@Transactional
public interface StockLevel {
   public int get () throws Exception;
   public void set (int value) throws Exception;
   public void decr (int value) throws Exception;
}
```

}



DEVOXX United Kingdom

Example: Define locking semantics

- Next, to ensure that the transactional object is free from conflicts when used in a concurrent environment we must indicate which methods read or mutate state using @ReadLock and @WriteLock annotations:

```
public class StockLevelImpl implements StockLevel {
   @ReadLock
   public int get () throws Exception { return state; }
   @WriteLock
   public void set (int value) throws Exception { state = value; }
   @WriteLock
   public void decr (int value) throws Exception { state -= value; }

   @State    private int state;
}
```

Example: Use the STM object

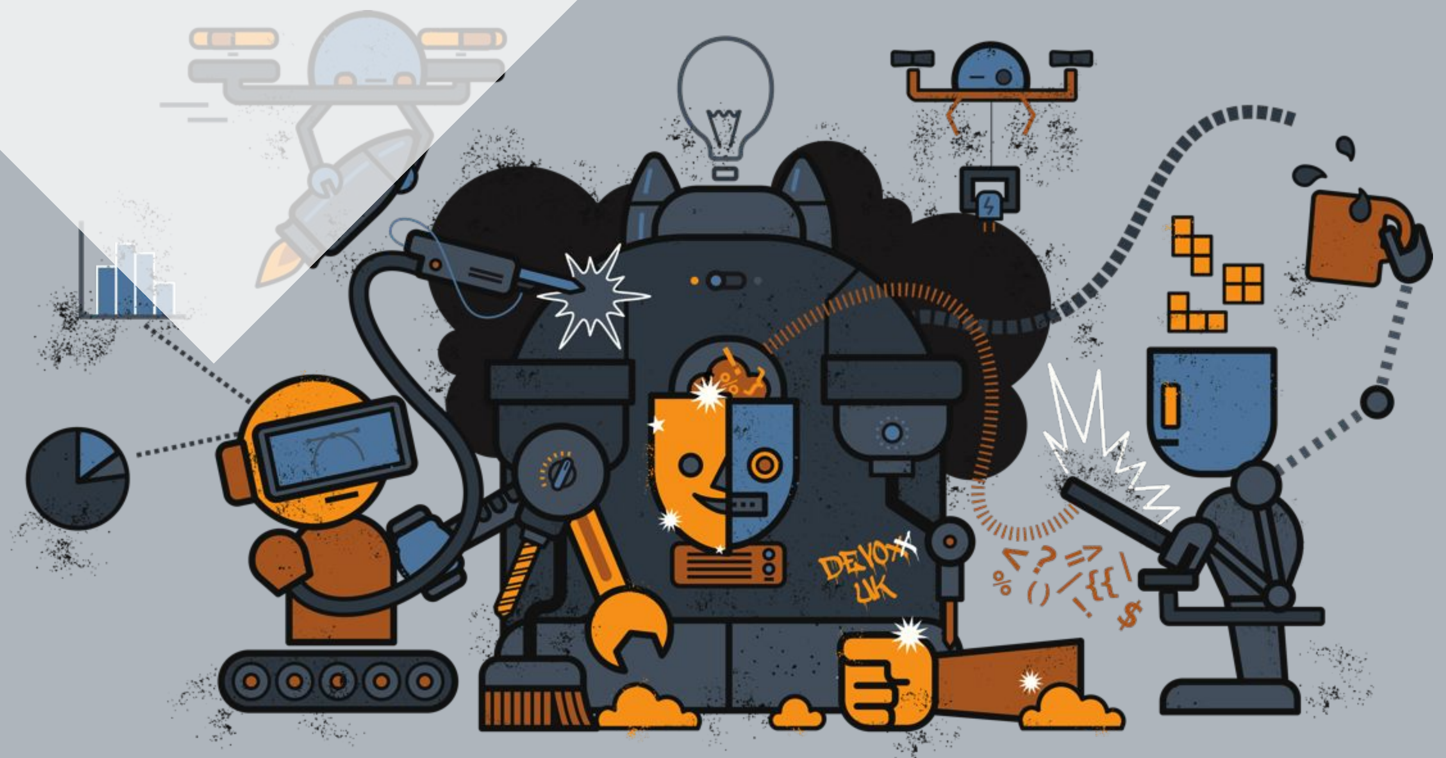
- Now we create instances of the transactional object (via an STM container) and use them inside transactions:

```
Container<StockLevel> container = new Container<>();
StockLevelImpl stock = new StockLevelImpl();

StockLevel obj = container.create(stock);

// update the STM object inside a transaction:
AtomicAction a = new AtomicAction(); // or use annotations to define transaction boundaries

a.begin();
obj.set(1234);
a.commit();
```

Agenda

- The Actor Model
- Transactions and Software Transactional Memory (STM)
- Eclipse Vert.x
- Narayana STM
- Code Walkthrough
- **Combining Narayana STM with Vert.x to create highly available shared data systems**

Demos

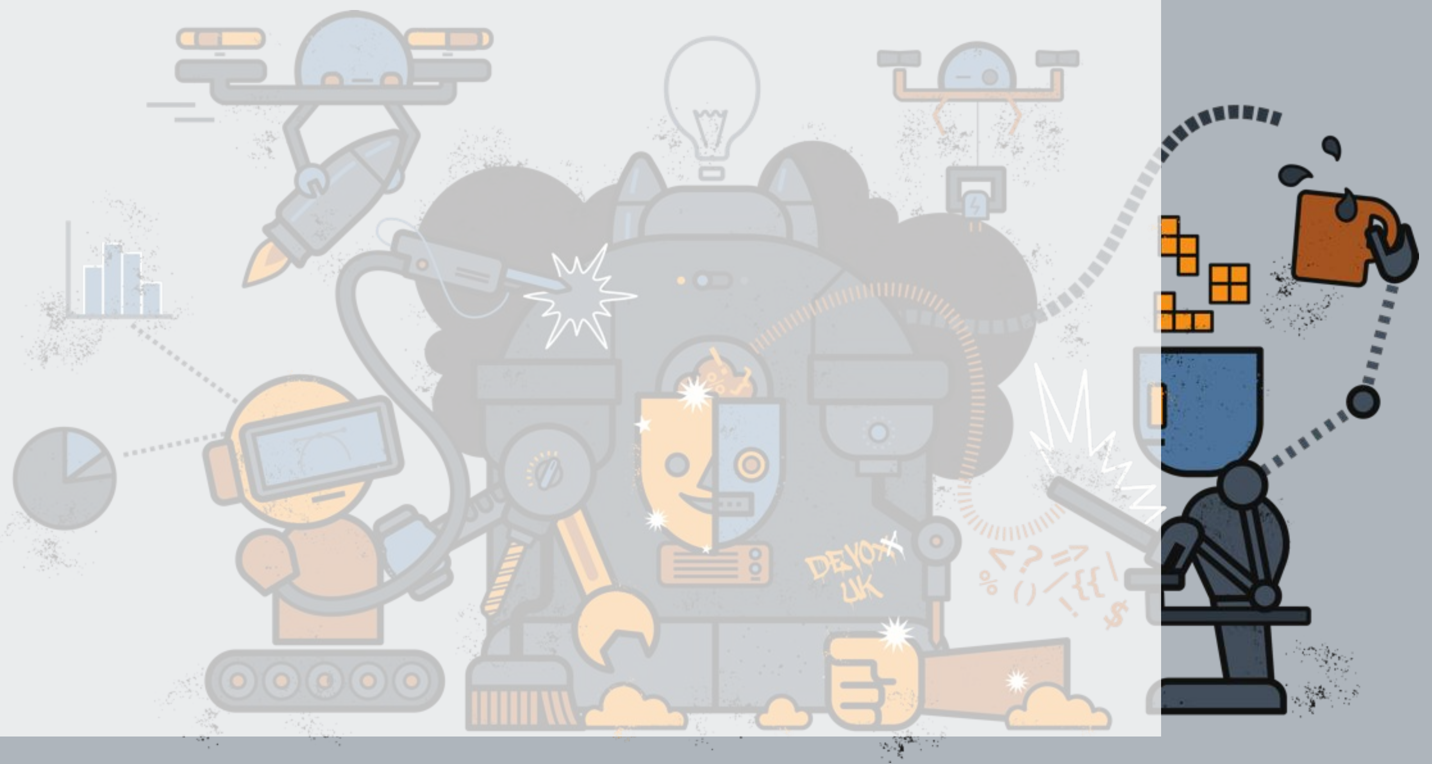


Use Case 1: Scaling with more threads

- Characteristics:
  - RECOVERABLE and EXCLUSIVE
  - creates multiple Vert.x instances each instance using a handle to the same STM object
- What it's good for:
  - vertical scaling where adding better h/w is an option in order to support more threads in one JVM

Use Case 1: Scaling with more threads

- This example shows a Theatre booking service:
  - an STM object is used to maintain all Theatre bookings
    the STM object is volatile
  - multiple Vert.x instances each listening on the same HTTP endpoint
  - each Vert.x instance shares the same STM object
  - all vertx instances run in the same address space
    concurrency is managed by the STM runtime
  - shows vertical scaling

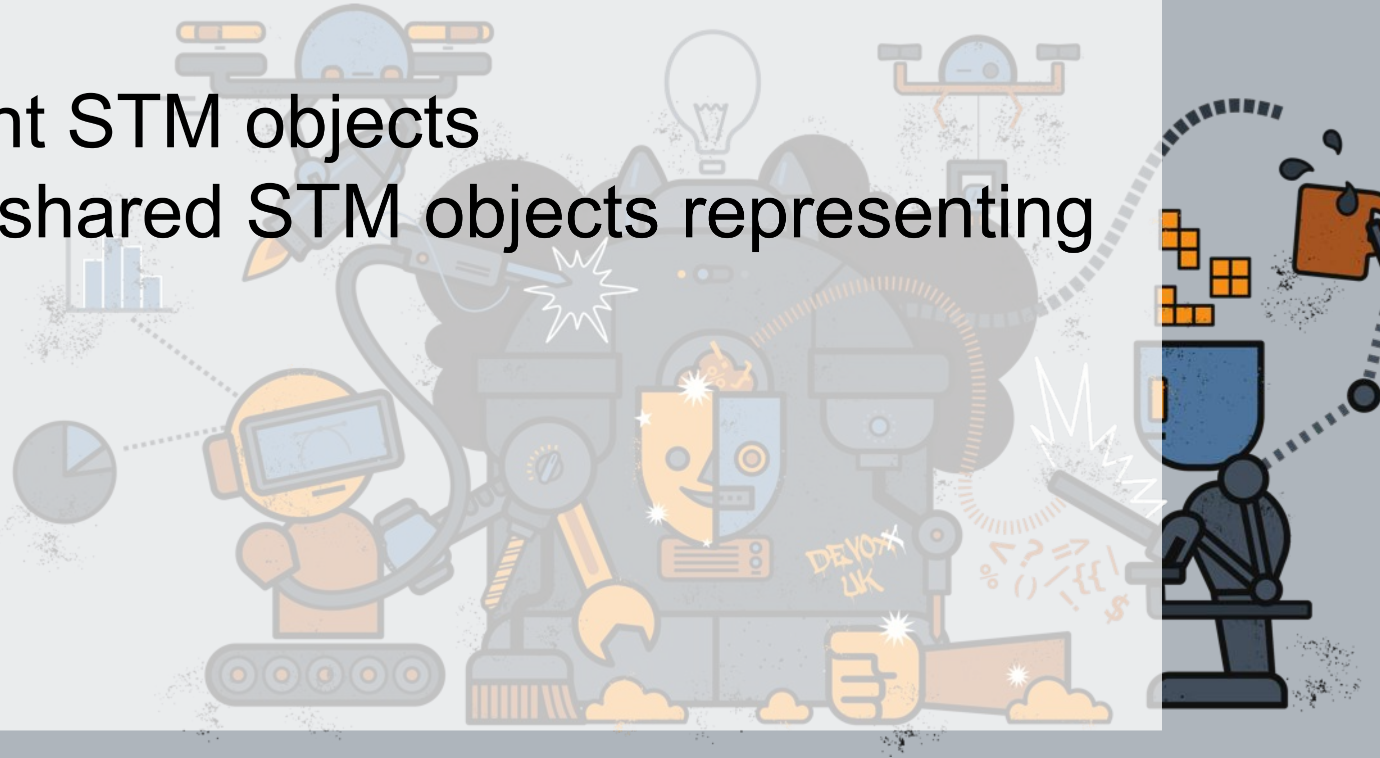


Use Case 2: Scaling with more JVMs

- Similar to use case 1 but uses persistent STM objects spread load across different JVMs:
- Characteristics:
  - PERSISTENT and SHARED
  - theatre service verticles running in different JVMs sharing the same STM object
  - each JVM hosting multiple vert.x instances each instance using a handle to the same STM object
- What it's good for:
  - horizontal scaling by using better hardware so that more threads can do be used to service the workload;

Use Case 3: Sharing different STM objects

- Characteristics:
  - RECOVERABLE and EXCLUSIVE
  - trip service verticle (multiple instances) using STM objects to maintain theatre and taxi bookings
  - providing HTTP endpoints for making trip, theatre and taxi bookings.
- What it's good for:
  - composing transactional operations across different STM objects
- The trip service fulfils booking requests by updating shared STM objects representing the theatre and taxi booking services respectively.



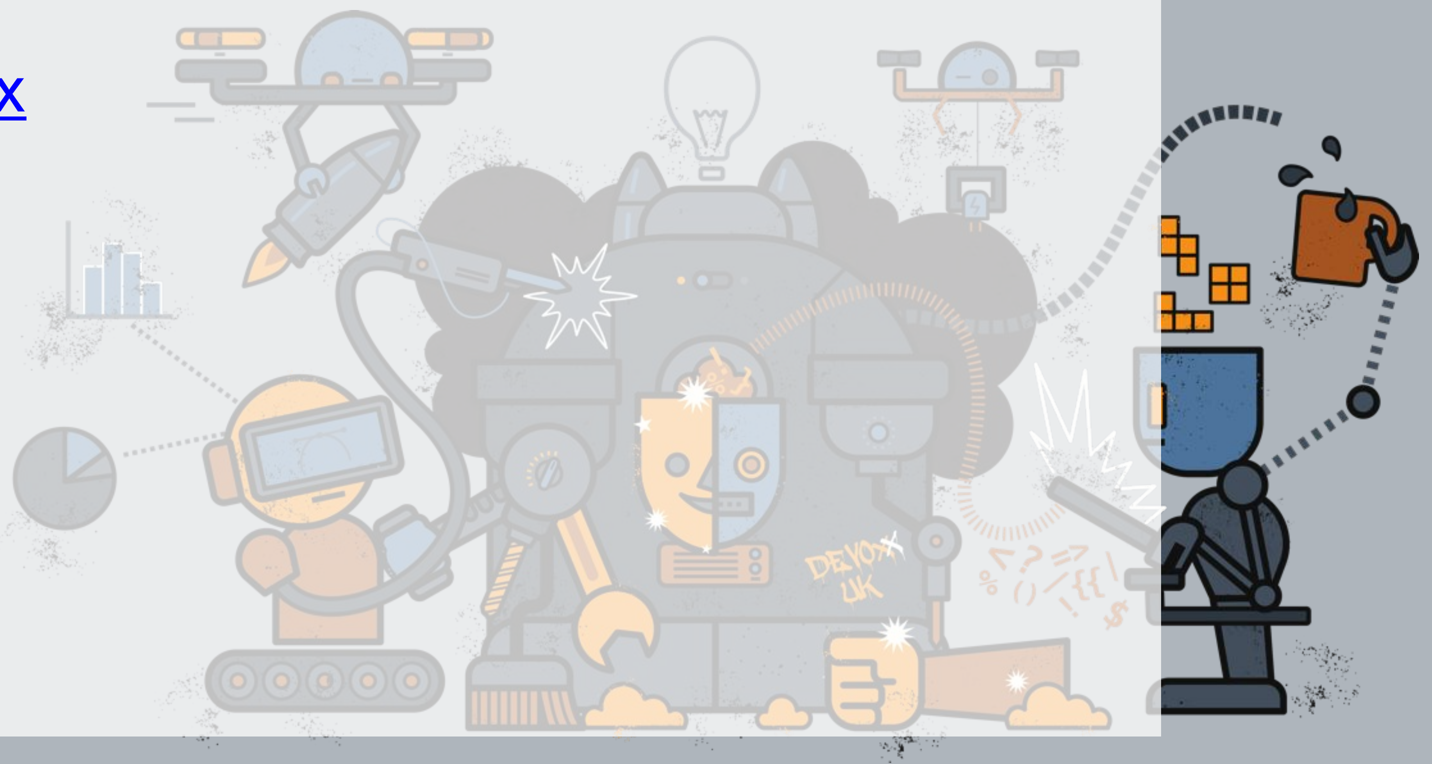
DEVOXX United Kingdom

Use Case 4: Stress Testing

- None of the previous use cases demonstrate contention of the STM objects
- The final use case will:
  - Start use case 1 (theatre booking example)
  - and then make lots of concurrent trip bookings and validate that the expected number of bookings are made.

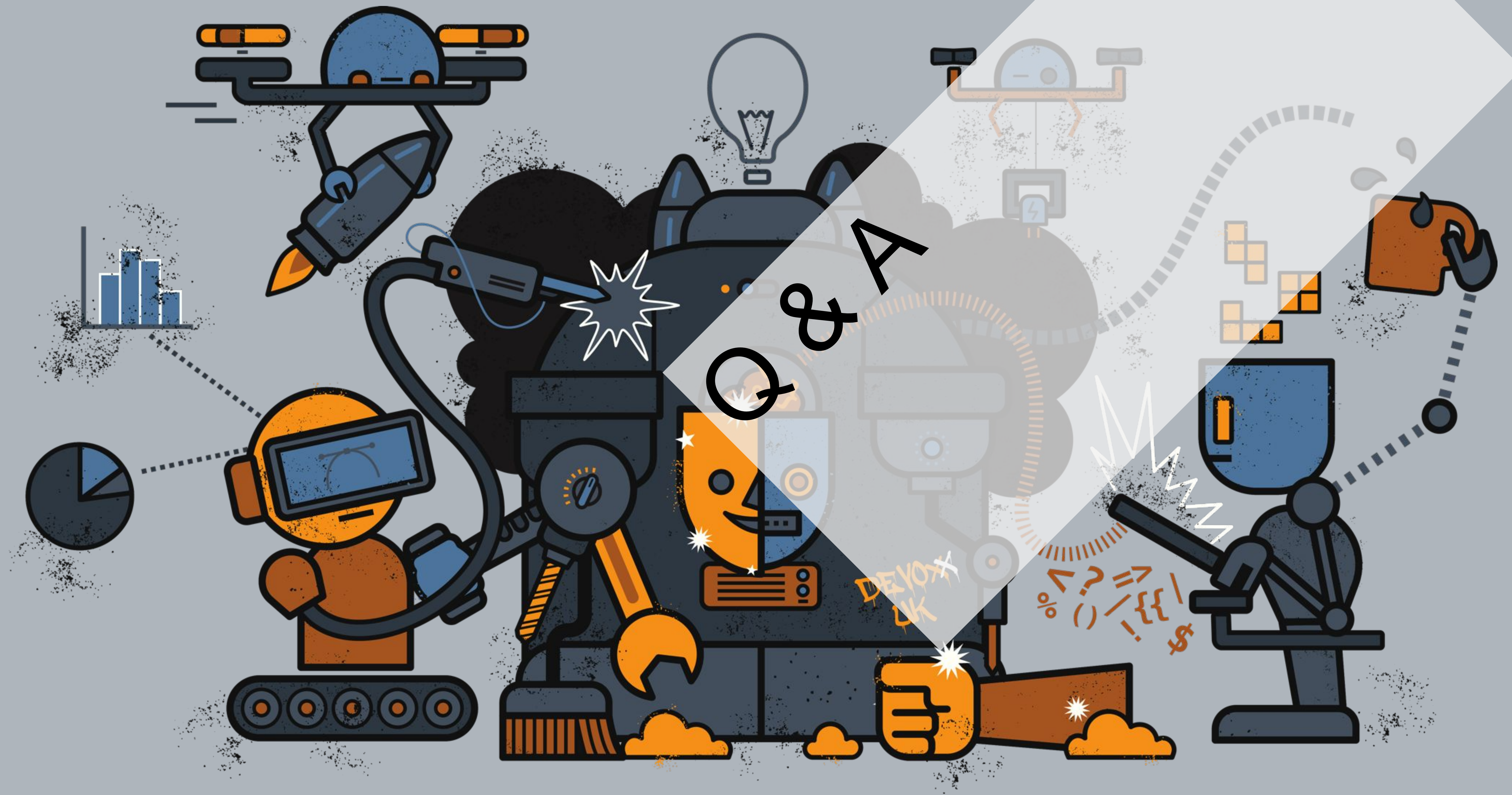
Summary

- More information available from narayana.io
  - Forums
  - Blogs
  - IRC
- Demo source
  - https://github.com/jbosstm/quickstart/tree/master/STM/vertx
- STM source
  - https://github.com/jbosstm/narayana/tree/master/STM



DEVOXX United Kingdom

Q & A



DEVOXX United Kingdom