Future  jection>

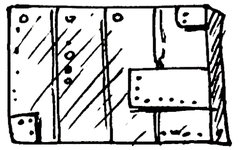
THE CURIOS CASE OF EVENTUAL PROVIDERS

WHAT IS THE USE CASE?

- Loading and processing data from many sources
  - During initial loading of service etc
  - Scatter-Gather with intermediate combining steps
  - Execute independent steps in parallel

THE TOMB OF LORIC

THE OLD TOWER



(TRANSMUTE MITHRIL WALL TO WOOD AND BREAK IT, RETRIEVING THE KEY)

THE BONES OF LORIC

WIND MILL KEY

THE CASTLE KEY

THE BONE DUST OF LORIC

THE POTION OF MITHRIL TRANSMUTATION

POSSIBLE SOLUTIONS?

- Tools for workflow and pipelines?
  - DigDag, Jenkins, BazelBuild, ETLs…?
- Asynchronous execution
  - Event Bus and Handlers
  - Composition of Futures/Promises
- _____
  - (fill in the blanks)

POSSIBLE SOLUTIONS?

- Tools for workflow and pipelines?
  - DigDag, Jenkins, BazelBuild, ETLs…?
  - Maybe, but a bit heavyweight for a micro service, complex configuration
- Asynchronous execution
  - Event Bus and Handlers
  - Composition of Futures/Promises
  - How about callback hell? Maintainability can be better

ASYNC COMPOSITION, NOT THAT BAD

```
CompletableFuture<String> A = CompletableFuture.supplyAsync(() -> "A");
CompletableFuture<String> B = CompletableFuture.supplyAsync(() -> "B");
CompletableFuture<String> C = A.thenCombine(B, (a, b) -> a + b);
CompletableFuture<String> D =
    C.thenApply(c -> c + "D")
        .thenComposeAsync(d -> A.thenApply(a -> a + d));
```

BUT CAN GROW [UNWIELDY]

```
return getCurrentUser()
  .then((user) => {
    const promises = [];
    if (attachFavouriteFood) {
      promises.push(getFood(user.favouriteFoodId)
        .then((food) => {
          user.food = food;
        }));
    }
    if (attachSchool) {
      promises.push(getSchool(user.schoolId)
        .then((school) => {
          user.school = school;
          if (attachFaculty) {
            return getUsers(school.facultyIds)
              .then((faculty) => {
                user.school.faculty = faculty;
              });
          }
        }));
    }

    return Promise.all(promises)
      .then(() => {
        return user;
      });
  });
```

POSSIBLE SOLUTIONS?

- Tools for workflow and pipelines?
  - DigDag, Jenkins, BazelBuild, ETL…?
  - Maybe, but a bit heavyweight for micro service, complex configuration
- Asynchronous execution
  - Event Bus and Handlers
  - Composition of Futures/Promises
  - How about callback hell? Maintainability can be better

- Dependency injection?
  - Threading, error handling, all different

FUTURE DEPENDENCY INJECTION

- DI is for constructing object graphs, not exactly for computing, right?
- What if we instead of constructing A, B, C graph
  would construct Future<A>, Future<B>, Future<C> composition graph
- Benefits
  - Clear, declarative definition of computations
  - Lightweight runtime which can be used in microservices
  - Everything is in comparison

```
B map(A a) {
  return new B(a.a());
}


ListenableFuture<B> map(ListenableFuture<A> a) {
  return Futures.transform(a,
      aResult -> new B(aResult.a()));
}
```

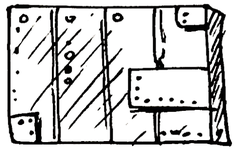
```
C combine(A a, B b) {
  return new C(a.a(), b.b());
}
```

```
ListenableFuture<C> combine(ListenableFuture<A> a, ListenableFuture<B> b) {
  return Futures.transform(Futures.allAsList(Arrays.asList(a, b)),
      (List<Object> input) -> {
        A a = (A) input.get(0);
        B b = (B) input.get(1);
        return new C(a.a(), b.b());
      });
}
```

LIVE CODING

THE TOMB OF LORIC

THE OLD TOWER



(TRANSMUTE MITHRIL WALL TO WOOD AND BREAK IT, RETRIEVING THE KEY)

THE BONES OF LORIC

WIND MILL KEY

THE CASTLE KEY

THE BONE DUST OF LORIC

THE POTION OF MITHRIL TRANSMUTATION

```java
@Value.Immutable
interface TombOfLoric {}

@Value.Immutable
interface OldTower {}

@Value.Immutable
interface BonesOfLoric {}

@Value.Immutable
interface WindmillKey {}

@Value.Immutable
interface Bonedust {
  @Value.Parameter
  BonesOfLoric bones();
}

@Value.Immutable
interface PotionOfMithrill {
  @Value.Parameter
  Bonedust component();
}
```

```java
@Singleton
public class Barbican {

  @Eventually.Provides
  public TombOfLoric tomb() {
    return ImmutableTombOfLoric.of();
  }

  @Eventually.Provides
  public OldTower tower() {
    return ImmutableOldTower.of();
  }

  @Eventually.Provides
  public WindmillKey millKey(OldTower oldTower) {
    tickProgress(4, "millKey from the " + oldTower);
    return ImmutableWindmillKey.of();
  }

  @Eventually.Provides
  public BonesOfLoric bones(TombOfLoric tombOfLoric) {
    tickProgress(5, "bones from the " + tombOfLoric);
    return ImmutableBonesOfLoric.of();
  }

  @Eventually.Provides
  public Bonedust bonedust(WindmillKey key, BonesOfLoric bones) {
    tickProgress(5, "bonedust milled on the windmill opened using key " + key);
    return ImmutableBonedust.of(bones);
  }

  @Exposed
  @Eventually.Provides
  public PotionOfMithrill potion(Bonedust bonedust) {
    tickProgress(3, "potion");
    return ImmutablePotionOfMithrill.of(bonedust);
  }
```

```java
//… CONTINUATION
  public static void main(String... args) {
    // For parallel execution
    ExecutorService executor = Executors.newCachedThreadPool();

    Injector injector = new EventualModules.Builder()
        .add(Barbican.class)
        .executor(executor)
        .joinInjector();

    PotionOfMithrill result = injector.getInstance(PotionOfMithrill.class);

    System.out.println(result);
    executor.shutdown();
  }

  private static void tickProgress(int times, String millKey) {
    for (int i = 0; i < times; i++) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
      }
      int percentage = (int) (((i + 1) / (float) times) * 100);
      System.out.printf("%d%% %s\n", percentage, millKey);
    }
  }
}
```

```
>> (Notice parallel execution for millKey and bones
25% millKey from the OldTower{}
20% bones from the TombOfLoric{}
50% millKey from the OldTower{}
40% bones from the TombOfLoric{}
60% bones from the TombOfLoric{}
75% millKey from the OldTower{}
80% bones from the TombOfLoric{}
100% millKey from the OldTower{}
100% bones from the TombOfLoric{}
20% bonedust milled on the windmill opened using key WindmillKey{}
40% bonedust milled on the windmill opened using key WindmillKey{}
60% bonedust milled on the windmill opened using key WindmillKey{}
80% bonedust milled on the windmill opened using key WindmillKey{}
100% bonedust milled on the windmill opened using key WindmillKey{}
33% potion
66% potion
100% potion
PotionOfMithrill{component=Bonedust{bones=BonesOfLoric{}}}
```

LIBRARIES

- Dagger 2
  https://google.github.io/dagger/producers.html

- Eventual Providers for Guice
  https://github.com/immutables/eventual

THE FUTURE OF DEPENDENCY INJECTION?

- Combine with Async/Future (as just shown)
- Combine with Observables/Flow (RxJava etc)
- ????
- Q+A