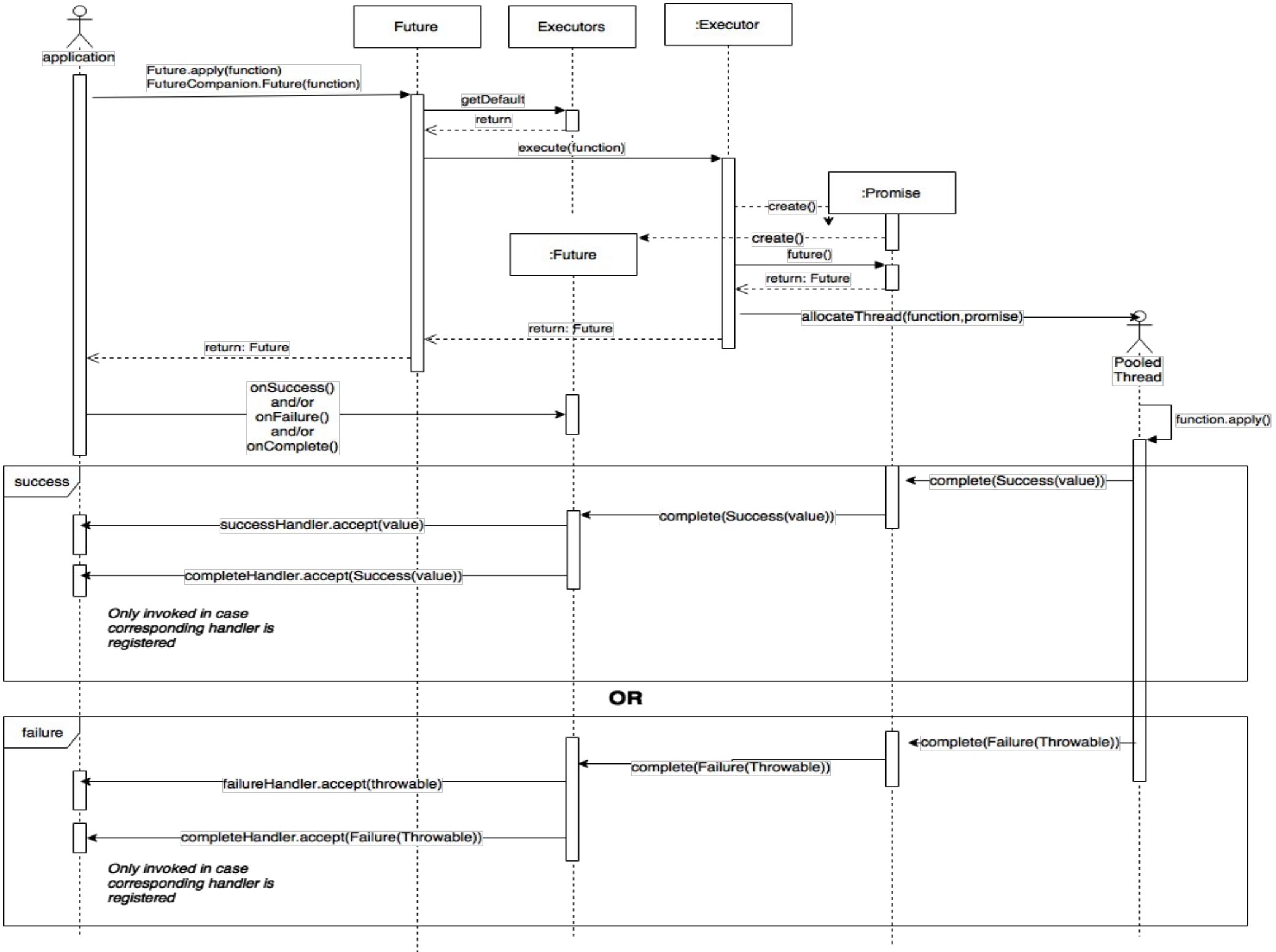
Akka advanced

Дополнительные возможности Akka

Future

- Future – стандартная абстракция которая позволяет запускать параллельное вычисление а также работать с “обещанием” его результата.
- Результат получается либо асинхронно с помощью callback, либо синхронно с помощью блокирующего вызова
- Выполнение Future производится в отдельном потоке который контролируется с помощью ExecutionContext
- Akka имеет как свою реализацию Future, так и может работать со стандартной реализацией java8 - CompletionStage


application
Future
Executors
:Executor
Future.apply(function)
FutureCompanion.Future(function)
getDefault
return
execute(function)
:Promise
create()
:Future
create()
future()
return: Future
allocateThread(function,promise)
Pooled Thread
return: Future
return: Future
onSuccess()
and/or
onFailure()
and/or
onComplete()
function.apply()
success
complete(Success(value))
complete(Success(value))
successHandler.accept(value)
completeHandler.accept(Success(value))
Only invoked in case corresponding handler is registered
OR
failure
complete(Failure(Throwable))
complete(Failure(Throwable))
failureHandler.accept(throwable)
completeHandler.accept(Failure(Throwable))
Only invoked in case corresponding handler is registered

Как создать Future ?

- Future может быть создан от константы
  - Future<String> f1 = Futures.successful("foo");
  - Future<String> a2 = Futures.failed(
    new IllegalArgumentException("Bang!"));
- Оборачивать функцию
  - Future<String> f = future(new Callable<String>() {
    public String call() {
    return "Hello" + "World";
    }
    }, system.dispatcher());
- Оборачивать вызов актора
  - Future<Object> future = Patterns.ask(actor, msg, timeout);

Основные возможности Future

- Методы map, andThen объекта Future позволяют нам выстраивать цепочки асинхронных вычислений

```
Future<String> f1 = Futures.future(()→"Hello World",system.dispatcher());
Future<Integer> f2 = f1.map( new Mapper<String, Integer>() {
                                        public Integer apply(String parameter) {
                                            return parameter.length();
                                        }
                                    }, system.dispatcher());
```

-

Коллекции Future

- Методы traverse,firstCompletedOf, reduce, fold объекта Futures позволяют работать с группой futures

  Например :

- firstCompletedOf – возвращает первый future который был исполнен
  - static <T> Future<T> firstCompletedOf(

    Iterable<Future<T>> futures, ExecutionContext executor

)
- Reduce – возвращает Future с результатом применния функции reduce ко всем результатам future
  - static <T,R> Future<R> reduce(

    Iterable<Future<T>> futures,

    Function2<R,T,R> fun,

    ExecutionContext executor

)

Результат Future

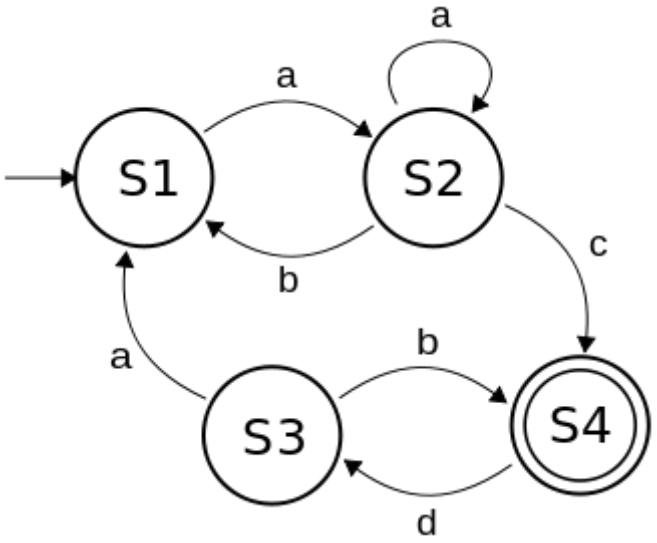
- Мы можем получить результат в результате синхронного вызова value()
  - Integer value = f2.value().get().get();
    - value() возвращает Option<Try<Integer>>
      - Option - класс реализующий монаду “may be”
      - Try – класс контейнер для обработки ошибок
- Также с результатом можно работать с помощью callback

```
future.onSuccess(new OnSuccess<String>() {
  public void onSuccess(String result) {
    if ("bar" == result) {
      //Do something if it resulted in "bar"
    }
  }
}, ec);
```

- Аналогично метод onFailure – обрабатывает с помощью callback ситуацию ошибки (т.е. во время выполнения Future произошла ошибка)
- Метод onComplete – позволяет в одной функции обработать обе ситуации

Finite state machine

- Akka содержит простую реализацию конечных автоматов – FSM.
- Конечный автомат задает граф переходов состояний актора
- Задается набором связей вида
  - State(S) x Event(E) -> Actions (A), State(S')



Реализация FSM в akka

- Создаем enum со списком состояний и определяемся с классом хранилищем состояния
- Наследуем класс актора от AbstractFSM
  - Указываем для него два класса – enum состояний и класс хранилище данных состояния
- С помощью методов базового класса настраиваем граф переходов

Пример простого FSM актора

- Создаем enum с состояниями

```
public static enum SemaphoreStates {
        Unitialized,
        OK,
        Failure,
        Reparing
    }
```

- Указываем что класс использует наш enum и Integer для хранения счетчика

```
public class SemaphoreActor extends AbstractFSM<SemaphoreActor.SemaphoreStates, Integer> {
```

- Задаем начальное состояние

```
{    startWith(SemaphoreStates.Unitialized, 0);
```

- Задаем последовательно для каждого состояния граф переходов из него

```
    when(SemaphoreStates.Unitialized,
        matchEvent(SemaphoreOkEvent.class, Integer.class,
                (e, i) -> goTo(SemaphoreStates.OK))
            .event(SemaphoreFailureEvent.class, Integer.class,
                (e, i) -> goTo(SemaphoreStates.Failure)));
```

Продолжение примера

```
    when(SemaphoreStates.OK,
            matchEvent(SemaphoreFailureEvent.class, Integer.class,
                    (e, i) -> goTo(SemaphoreStates.Failure)));
    when(SemaphoreStates.Failure,
            matchEvent(SemaphoreOkEvent.class, Integer.class,
                    (e, i) -> goTo(SemaphoreStates.Reparing).using(1)));
    when(SemaphoreStates.Reparing, FiniteDuration.apply(10, TimeUnit.SECONDS),
            matchEvent(SemaphoreOkEvent.class, Integer.class,
                    (e, i) -> i > REPARE_BARRIER ? goTo(SemaphoreStates.OK) : stay().using(i + 1))
            .event(Arrays.asList(SemaphoreFailureEvent.class, StateTimeout()), Integer.class,
                    (e, i) -> goTo(SemaphoreStates.Failure))
    );
    whenUnhandled(
            matchEvent(SemaphoreStateRequest.class, Integer.class,
                    (e, i) -> stay().replying(new SemaphoreStateReply(stateName()))));
    onTransition(
            (fromState, toState) -> {
                System.out.println("transition from " + fromState + " to " + toState);
            });
}
```

Akka HTTP

- Akka HTTP – отдельный модуль предоставляющий средства для работы с протоколом HTTP.
- Доступны api сервера и клиента
- Разработка приложения происходит в инфраструктуре Akka
  - HTTP запросы и ответы представляют собой сообщения
  - Реализация полностью асинхронна
  - Доступны два вида серверного API :
    - Высокоуровневое
    - Низкоуровневое

Инициализация сервера

```
ActorSystem system = ActorSystem.create("routes");
final Http http = Http.get(system);
final ActorMaterializer materializer = ActorMaterializer.create(system);
MainHttp instance = new MainHttp(system);
final Flow<HttpRequest, HttpResponse, NotUsed> routeFlow =
instance.createRoute(system).flow(system, materializer);
final CompletionStage<ServerBinding> binding = http.bindAndHandle(
        routeFlow,
        ConnectHttp.toHost("localhost", 8080),
        materializer
);
System.out.println("Server online at http://localhost:8080/\nPress RETURN to stop...");
System.in.read();
binding
        .thenCompose(ServerBinding::unbind)
        .thenAccept(unbound -> system.terminate());
```

High level API

- Идея высокоуровневого серверного api состоит в том чтобы задать дерево разбора HTTP запросов с помощью DSL
- Далее дерево “компилируется” в набор акторов с помощью materializer
- Полученный набор акторов запускается в akka system
- Минимальный пример

```
Route route =
  get(
    () -> complete("Received GET")
  ).orElse(
    () -> complete("Received something else")
  )
```

Создание Route

- Общий вид одного “кирпичика” создания route

```
directiveName(arguments [, ...], (extractions [, ...]) -> {
  ... // inner route
})
```

- Основные директивы :
  - Набор альтернатив : route
  - Фрагмент http url : path
  - Методы http : get,post
  - Query parameter : parameter
  - Body запроса : entity

Пример создания route

```
route(
        path("semaphore", () ->
            route(
                get( () -> {
                    Future<Object> result = Patterns.ask(testPackageActor,
SemaphoreActor.makeRequest(), 5000);
                    return completeOKWithFuture(result, Jackson.marshaller());
                })),
        path("test", () ->
            route(
                post(() ->
                    entity(Jackson.unmarshaller(TestPackageMsg.class), msg -> {
                        testPackageActor.tell(msg, ActorRef.noSender());
                        return complete("Test started!");
                    }))),
        path("put", () ->
            get(() ->
                parameter("key", (key) ->
                    parameter("value", (value) ->
                    {
                        storeActor.tell(new StoreActor.StoreMessage(key, value), ActorRef.noSender());
                        return complete("value saved to store ! key=" + key + " value=" + value);
                    }))),
```

Результат работы Route

- В крайнем узле дерева разбора http запроса происходит обработка запроса и возврат результата обработки

```
(value) → {
                storeActor.tell(new StoreActor.StoreMessage(key, value),
ActorRef.noSender());
                return complete("value saved to store ! key=" + key + " value=" + value);
               }
```

- complete возвращает результат немедленно
- completeOKWithFuture возвращает future содержащий ответ (также мы можем указать marshaller для упаковки результата)
- Также доступны варианты в которых мы указываем с помощью Future статус, составной объект HttpResponse и т.д.

Scheduler

- Предназначен для запуска повторяющихся или отложенных действий
- Может работать в двух вариантах – отправка сообщения актору или выполнение java кода

```
system.scheduler().schedule(
        Duration.create(50, TimeUnit.MILLISECONDS),
        Duration.create(10, TimeUnit.SECONDS),
        testPackageActor,
        new PrintAllResultsToLogMsg(),
        system.dispatcher(),
        ActorRef.noSender()
        );
```