SpringOne Platform

Designing, Implementing, and Using Reactive APIs

Ben Hale (@nebhale)
Paul Harris (@twoseat)

Pivotal

Introduction to Reactive Programming

- Reactive is used to broadly define event-driven systems
- Moves imperative logic to
  - asynchronous
  - non-blocking
  - functional-style code
- Allows stable, scalable access to external systems

Cloud Foundry Java Client

- Cloud Foundry is composed of
  - RESTful APIs
  - Secured by OAuth
  - Transmitting large JSON request and response payloads

- APIs are low level, often requiring orchestration

Cloud Foundry Java Client

- Java has become the preferred language for orchestrating Cloud Foundry
- Complex orchestration is difficult when working against generic payloads

- A strongly typed Java API is required for maintainability

Generic Request and Response Payloads

```
Map<String, Object> request = new HashMap<>();
request.put("name", "test-application");
request.put("application", Paths.get("target/test-application.jar"));
request.put("buildpack", "java-buildpack");

this.operations.applications()
    .create(request)
    .map(response -> response.get("id"));
```

SpringOne Platform

Strongly Typed Request and Response Payloads

```
this.operations.applications()
  .create(CreateApplicationRequest.builder()
    .name("test-application")
    .application(Paths.get("target/test-application.jar"))
    .buildpack("java-buildpack")
    .build())
  .map(CreateApplicationResponse::getId);
```

Cloud Foundry Java Client

- The high-latency network interaction is a good fit for a Reactive Design
- Encourages asynchronous and highly concurrent usage

- Partnering with Project Reactor early influenced both the design of the client and of Reactor

SpringOne Platform

Design

Pivotal

When To Use Reactive

- Networking
  - Latency
  - Failures
  - Backpressure
- Highly Concurrent Operations
- Scalability

What Should a Reactive API Return?

- `Flux<T>`
  - Returns `0` to `N` values
- `Mono<T>`
  - Returns `0` to `1` value

- The Java Client often returns `Monos` containing collections

Using a Flux<T> Response

```java
Flux<Application> listApplications() {...}

Flux<String> listApplicationNames() {
  return listApplications()
    .map(Application::getName);
}

void printApplicationName() {
  listApplicationNames()
    .subscribe(System.out::println);
}
```

Using a Mono<T> Response

```
Flux<Application> listApplications() {...}

Mono<List<String>> listApplicationNames() {
  return listApplications()
    .map(Application::getName)
    .collectAsList();
}

Mono<Boolean> doesApplicationExist(String name) {
  return listApplicationNames()
    .map(names -> names.contains(name));
}
```

Every Method Must Return Something

- Unlike many imperative APIs, `void` is not an appropriate return type
- A reactive flow declares behavior but does not execute it

- Consumers must subscribe to a `Flux` or `Mono` in order to begin execution

Imperative Operation

```
void delete(String id) {
  this.restTemplate.delete(URI, id);
}

public static void main(String[] args) {
  delete("test-id");
}
```

Reactive Operation

```
Mono<Void> delete(String id) {
  return this.httpClient.delete(URI, id);
}

public static void main(String[] args) {
  CountDownLatch latch = new CountDownLatch(1);

  delete("test-id")
    .subscribe(n -> {}, Throwable::printStackTrace, () -> latch.countDown());

  latch.await();
}
```

Scope of a Method

- Methods should be designed to be small and reusable

- Reusable methods can be more easily composed into larger operations
- These methods can be more flexibly combined into parallel or sequential operations

Reusable Methods

```java
Mono<ListApplicationsResponse> getPage(int page) {
  return this.client.applicationsV2()
    .list(ListApplicationsRequest.builder()
      .page(page)
      .build());
}

public static void main(String[] args) {
  getPage(1)
    .map(PaginatedResponse::getTotalPages)
    .flatMap(totalPages -> Flux.range(2, totalPages)
      .flatMap(page -> getPage(page)))
    .subscribe(System.out::println);
}
```

SpringOne Platform

Implement

Pivotal

Sequential and Parallel Coordination

- All significant performance improvements come from concurrency
- But, concurrency is very hard to do properly

- Reactive frameworks allow you to define sequential and parallel relationships between operations
- The framework does the hard work of coordinating the operations

Coordination

```
Mono<ListApplicationsResponse> getPage(int page) {
  return this.client.applicationsV2()
    .list(ListApplicationsRequest.builder()
      .page(page)
      .build());
}

public static void main(String[] args) {
  getPage(1)
    .map(PaginatedResponse::getTotalPages)
    .flatMap(totalPages -> Flux.range(2, totalPages)
      .flatMap(page -> getPage(page)))
    .subscribe(System.out::println);
}
```

Conditional Logic

- Like normal values, errors pass through the flow's operations
- These errors can be passed all the way to consumers, or flows can change behavior based on them
- Flows can transform errors or generate new results based on the error

Error Logic

```java
public Mono<AppStatsResponse> getApp(GetAppRequest request) {
  return client.applications()
    .statistics(AppStatsRequest.builder()
      .applicationId(request.id())
      .build())
    .otherwise(ExceptionUtils.statusCode(APP_STOPPED_ERROR),
      t -> Mono.just(AppStatsResponse.builder().build()));
}
```

Conditional Logic

- It is valid for a flow to complete without sending any elements
  - This is often equivalent to returning `null` from an imperative API
- This completion can be passed all the way to consumers
- Alternatively, flows can switch behavior or generate new elements based on the completion

Empty Logic

```java
public Flux<GetDomainsResponse> getDomains(GetDomainsRequest request) {
  return requestPrivateDomains(request.getId())
    .switchIfEmpty(requestSharedDomains(request.getId));
}
```

Empty Logic

```java
public Mono<String> getDomainId(GetDomainIdRequest request) {
  return getPrivateDomainId(request.getName())
    .otherwiseIfEmpty(getSharedDomainId(request.getName()))
    .otherwiseIfEmpty(ExceptionUtils.illegalState(
      "Domain %s not found", request.getName()));
}
```

Conditional Logic

- Sometimes you want to make decisions not based on errors or emptiness, but on values themselves
- While it’s possible to implement this logic using operators, it often proves to be more complex than is worthwhile
- It’s OK to use imperative conditional statements

Conditional Logic Avoiding Imperative

```java
public Mono<String> getDomainId(String domain, String organizationId) {
  return Mono.just(domain)
    .filter(d -> d == null)
    .then(getSharedDomainIds()
      .switchIfEmpty(getPrivateDomainIds(organizationId))
      .next()
      .otherwiseIfEmpty(ExceptionUtils.illegalState("Domain not found")))
    .otherwiseIfEmpty(getPrivateDomainId(domain, organizationId)
      .otherwiseIfEmpty(getSharedDomainId(domain))
      .otherwiseIfEmpty(ExceptionUtils.illegalState("Domain %s not found", domain)));
}
```

Conditional Logic Using Imperative

```
public Mono<String> getDomainId(String domain, String organizationId) {
  if (domain == null) {
    return getSharedDomainIds()
      .switchIfEmpty(getPrivateDomainIds(organizationId))
      .next()
      .otherwiseIfEmpty(ExceptionUtils.illegalState("Domain not found"));
  } else {
    return getPrivateDomainId(domain, organizationId)
      .otherwiseIfEmpty(getSharedDomainId(domain))
      .otherwiseIfEmpty(ExceptionUtils.illegalState("Domain %s not found", domain));
    }
}
```

Testing

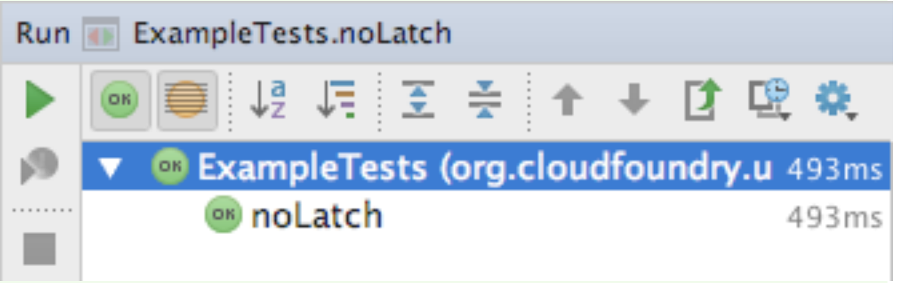
- Most useful flows will be asynchronous
- Testing frameworks are aggressively synchronous, registering results before asynchronous results return
- To compensate for this you must block the main thread and make test assertions in the asynchronous thread

Test Finishing Before Assertions

```
@Test
public void noLatch() {
  Mono.just("alpha")
    .subscribeOn(Schedulers.computation())
    .subscribe(s -> assertEquals("bravo", s));
}
```

Test Finishing Before Assertions

```java
@Test
public void noLatch() {
  Mono.just("alpha")
    .subscribeOn(Schedulers.computation())
    .subscribe(s -> assertEquals("bravo", s));
}
```

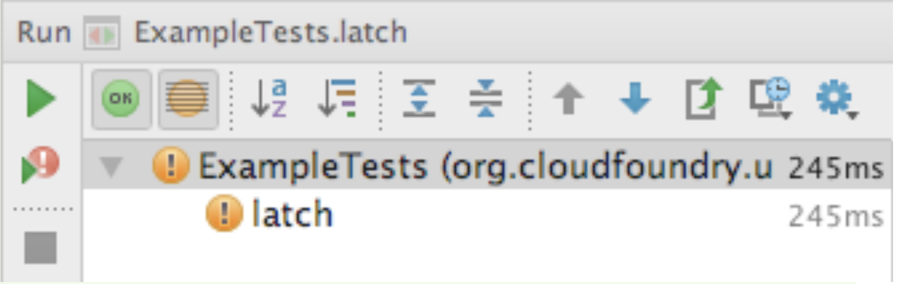


Test Blocking Using CountDownLatch

```java
@Test
public void latch() throws InterruptedException {
  CountDownLatch latch = new CountDownLatch(1);
  AtomicReference<String> actual = new AtomicReference<>();

  Mono.just("alpha")
    .subscribeOn(Schedulers.computation())
    .subscribe(actual::set, t -> latch.countDown(), latch::countDown);

  latch.await();
  assertEquals("bravo", actual.get());
}
```

Test Blocking Using CountDownLatch

```java
@Test
public void latch() throws InterruptedException {
  CountDownLatch latch = new CountDownLatch(1);
  AtomicReference<String> actual = new AtomicReference<>();

  Mono.just("alpha")
    .subscribeOn(Schedulers.computation())
    .subscribe(actual::set, t -> latch.countDown(), latch::countDown);

  latch.await();
  assertEquals("bravo", actual.get());
}
```



Test Subscriber

- Testing a Reactive design requires more than just blocking
  - Multiple value assertion
  - Expected error assertion
  - Unexpected error failure

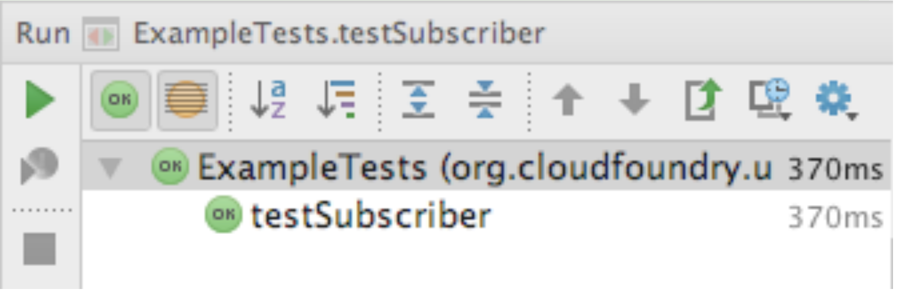
- A `TestSubscriber<T>` can collect this behavior into a single type

Test Using TestSubscriber<T>

```java
@Test
public void testSubscriber() throws InterruptedException {
  TestSubscriber<String> testSubscriber = new TestSubscriber<>();

  Flux.just("alpha", "bravo")
    .subscribeOn(Schedulers.computation())
    .subscribe(testSubscriber
      .assertEquals("alpha")
      .assertEquals("bravo"));

  testSubscriber.verify(ofSeconds(5));
}
```

Test Using TestSubscriber<T>

```
@Test
public void testSubscriber() throws InterruptedException {
  TestSubscriber<String> testSubscriber = new TestSubscriber<>();

  Flux.just("alpha", "bravo")
    .subscribeOn(Schedulers.computation())
    .subscribe(testSubscriber
      .assertEquals("alpha")
      .assertEquals("bravo"));

  testSubscriber.verify(ofSeconds(5));
}
```



SpringOne Platform

Use

Pivotal

Countdown Latches

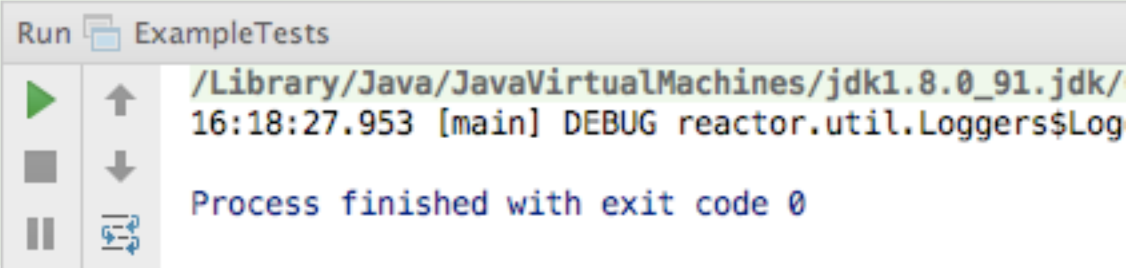
- Reactive frameworks can only coordinate their own operations
- Many execution environments have processes that outlast individual threads
  - Reactive programming plays nicely here
- Some execution environments aggressively terminate the process before any individual thread

main() Finishing Before Subscriber<T>

```
public static void main(String[] args) {
  Mono.just("alpha")
    .delaySubscription(Duration.ofSeconds(1))
    .subscribeOn(Schedulers.computation())
    .subscribe(System.out::println);
}
```

main() Finishing Before Subscriber<T>

```java
public static void main(String[] args) {
  Mono.just("alpha")
    .delaySubscription(Duration.ofSeconds(1))
    .subscribeOn(Schedulers.computation())
    .subscribe(System.out::println);
}
```

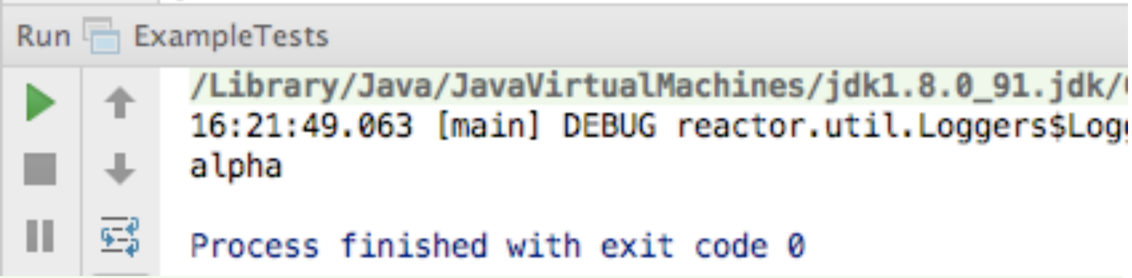


main() Waiting for Subscriber<T>

```
public static void main(String[] args) throws InterruptedException {
  CountDownLatch latch = new CountDownLatch(1);

  Mono.just("alpha")
    .delaySubscription(Duration.ofSeconds(1))
    .subscribeOn(Schedulers.computation())
    .subscribe(System.out::println, t -> latch.countDown(),
               latch::countDown);

    latch.await();
}
```

main() Waiting for Subscriber<T>

```java
public static void main(String[] args) throws InterruptedException {
  CountDownLatch latch = new CountDownLatch(1);

  Mono.just("alpha")
    .delaySubscription(Duration.ofSeconds(1))
    .subscribeOn(Schedulers.computation())
    .subscribe(System.out::println, t -> latch.countDown(),
               latch::countDown);

    latch.await();
}
```



Blocking flows

- It's very common to bridge between imperative and reactive APIs

- In those cases blocking while waiting for a result is appropriate.

Bridging Imperative and Reactive using `block()`

```
Mono<User> requestUser(String name) {...}

@Override
User requestUser(String name) {
  return getUser(name)
    .block();
}
```

Bridging Imperative and Reactive using `block()`

```
Flux<User> requestUsers() {...}

@Override
List<User> listUsers() {
  return requestUsers()
    .collectList()
    .block();
}
```

Error Handling

- Because errors are values they must be handled, they do not just show up
- Subscribe has 0...3 parameters
- Release latches on error

Error Handling

```
public static void main(String[] args) throws InterruptedException {
  CountDownLatch latch = new CountDownLatch(1);

  Flux.concat(Mono.just("alpha"), Mono.error(new IllegalStateException()))
    .subscribe(System.out::println, t -> {
      t.printStackTrace();
      latch.countDown();
    }, latch::countDown);

  latch.await();
}
```

Composable Method References

- Reactive programming is susceptible to callback hell, but it doesn't have to be
- Extracting behavior into private methods can be even more valuable than in imperative programming
- Well named methods, and method reference syntax, lead to very readable flows

Composable Method References

```java
@Override
public Flux<ApplicationSummary> list() {
  return Mono
    .when(this.cloudFoundryClient, this.spaceId)
    .then(function(DefaultApplications::requestSpaceSummary))
    .flatMap(DefaultApplications::extractApplications)
    .map(DefaultApplications::toApplicationSummary);
}
```

Point Free

- You may have noticed that our code samples use a *very* compact style
  - Point Free (http://bit.ly/Point-Free)
- It helps the developer think about composing functions (high level), rather than shuffling data (low level)
- You don't have to use, but we find that most people prefer it (eventually)

Point Free Style

```java
Mono<Void> deleteApplication(String name) {
  return PaginationUtils
    .requestClientV2Resources(page -> this.client.applicationsV2()
      .list(ListApplicationsRequest.builder()
        .name(name)
        .page(page)
        .build()))
    .single()
    .map(applicationResource -> applicationResource.getMetadata().getId())
    .then(applicationId -> this.client.applicationsV2()
      .delete(DeleteApplicationRequest.builder()
        .applicationId(applicationId)
        .build()));
}
```

SpringOne Platform

Summary

Pivotal

Designing, Implementing and Using Reactive APIs

- Design
  - When to use
  - Return Mono/Flux
  - Reusable Methods
- Implement
  - Sequential and Parallel
  - Conditional logic
  - Testing
- Use
  - Countdown Latches
  - Blocking flows
  - Error Handling
  - Composable methods
  - Point Free Style

SpringOne Platform

Learn More. Stay Connected.

https://github.com/cloudfoundry/cf-java-client

@springcentral
spring.io/blog

@pivotal
pivotal.io/blog

@pivotalcf
http://engineering.pivotal.io