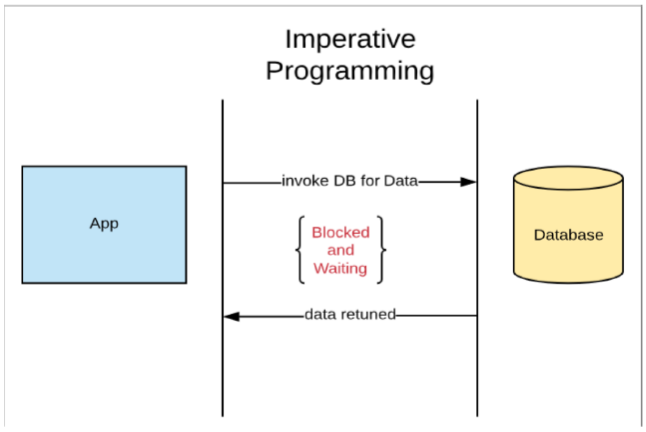
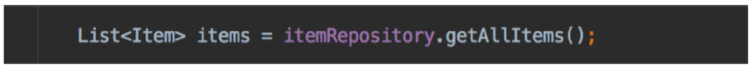
Imperative Programming:

```
List<Item> items = itemRepository.getAllItems();
```

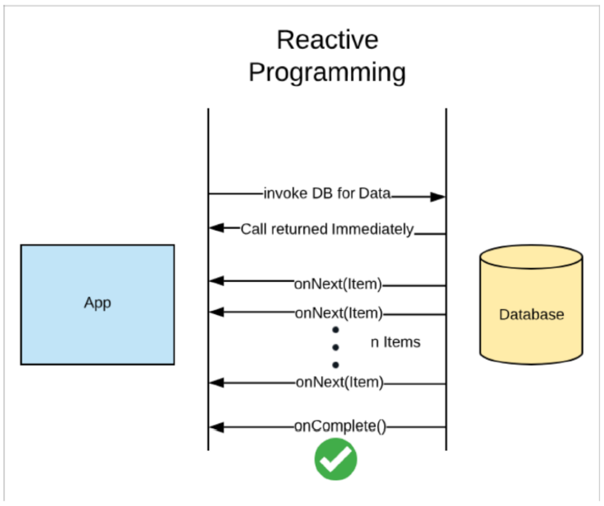


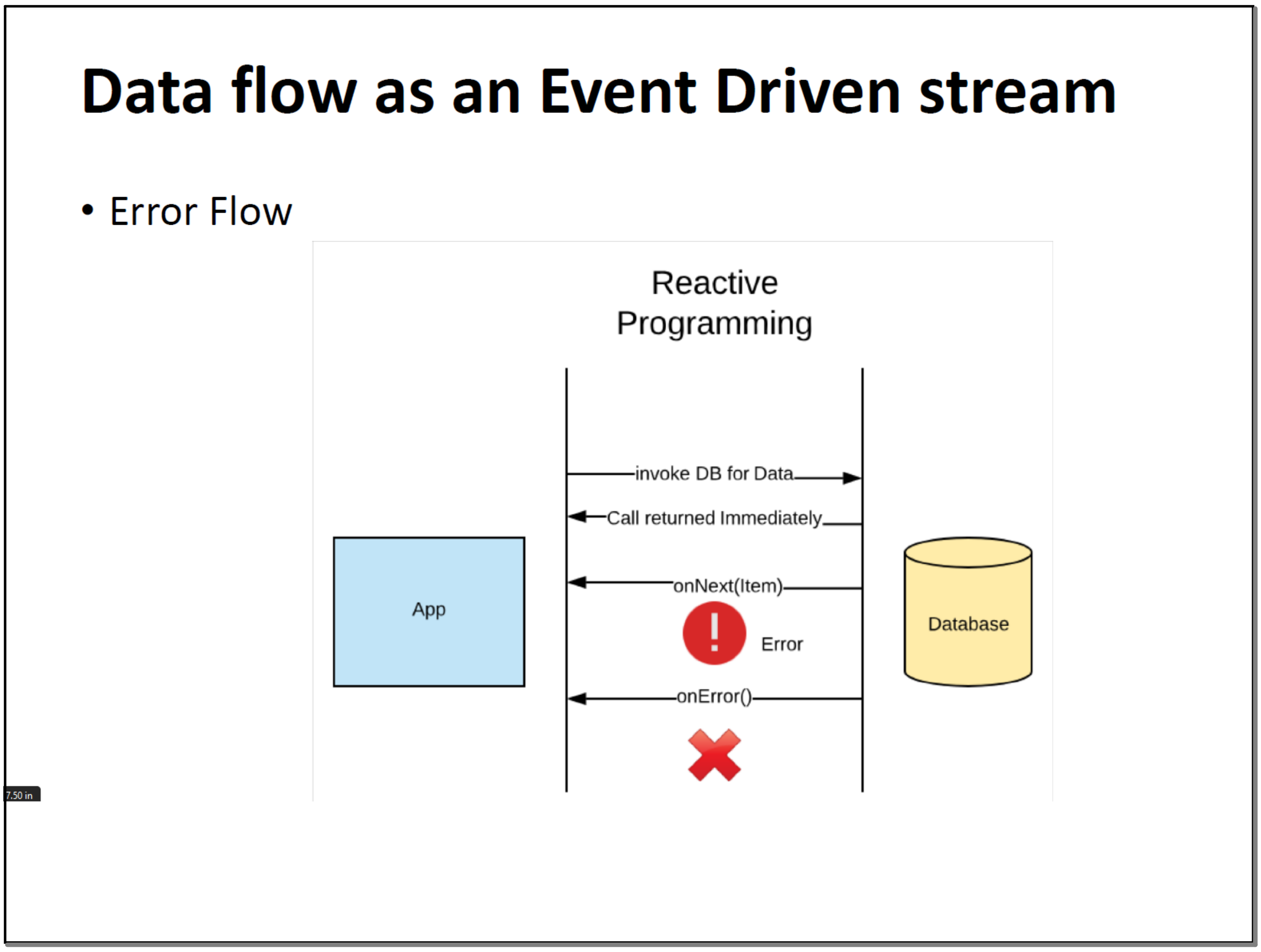
- Synchronous and blocking communication Model.

Data flow as an Event Driven stream



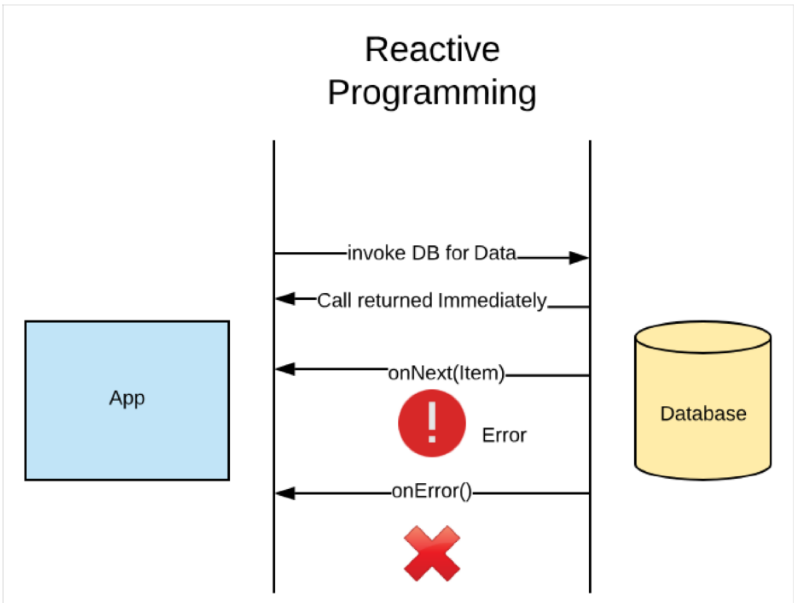
```java
List<Item> items = itemRepository.getAllItems();
```





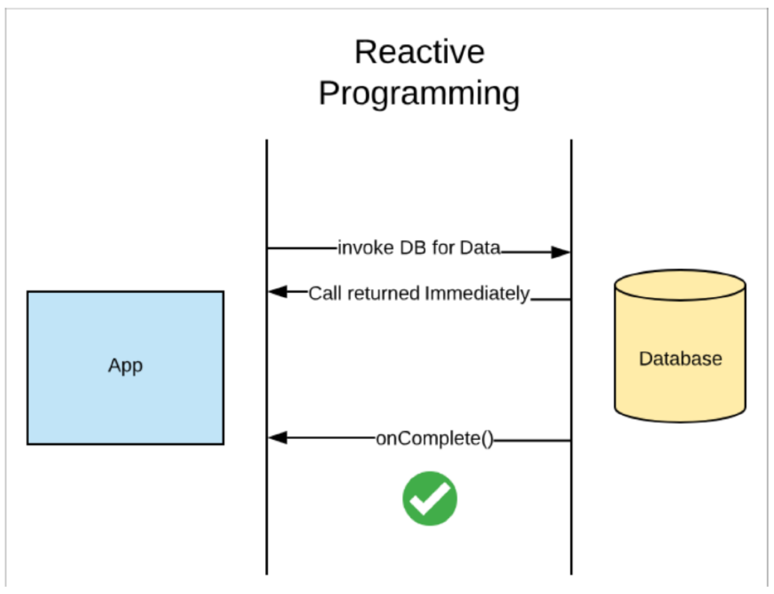
Data flow as an Event Driven stream

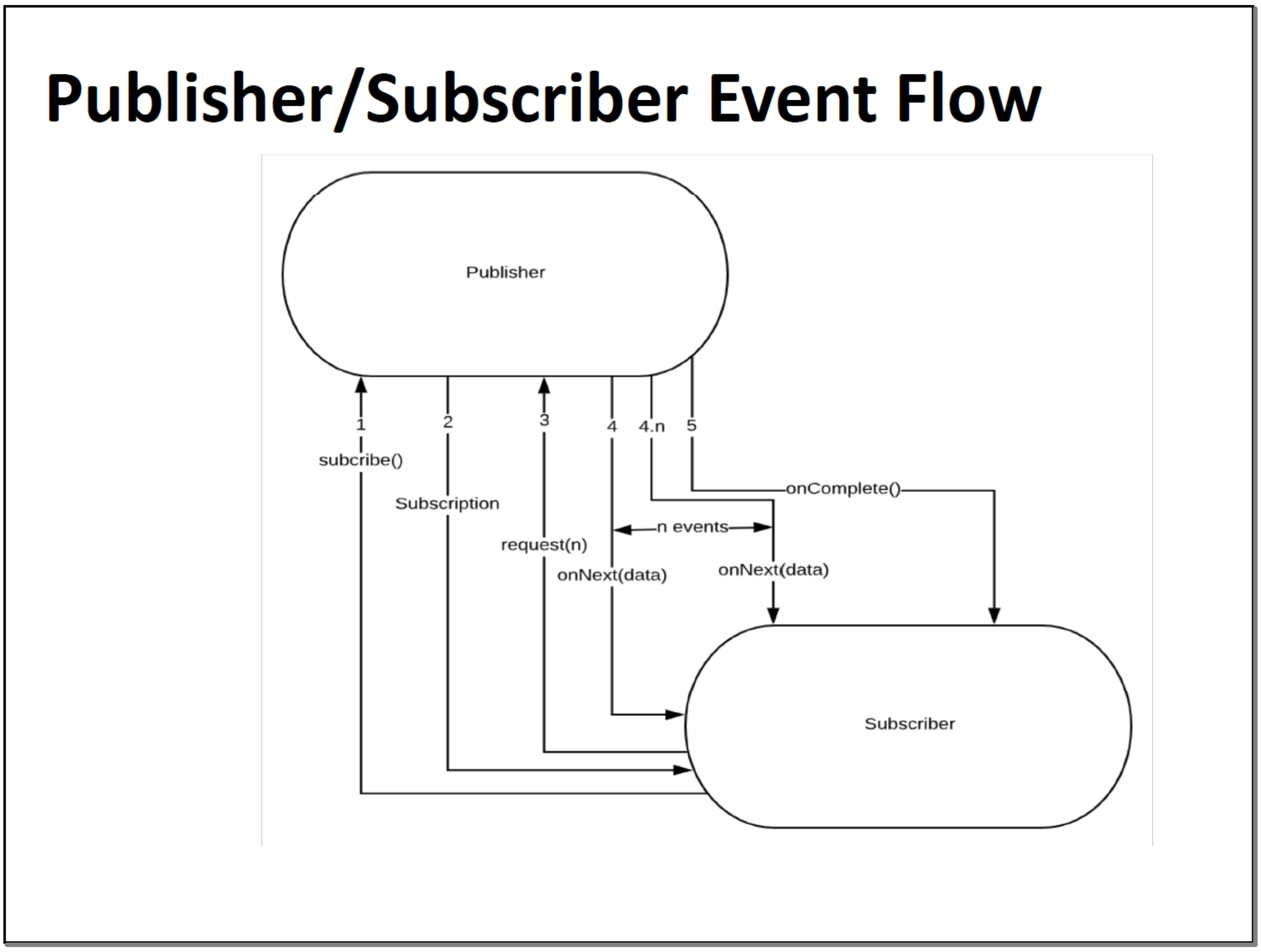
- Error Flow



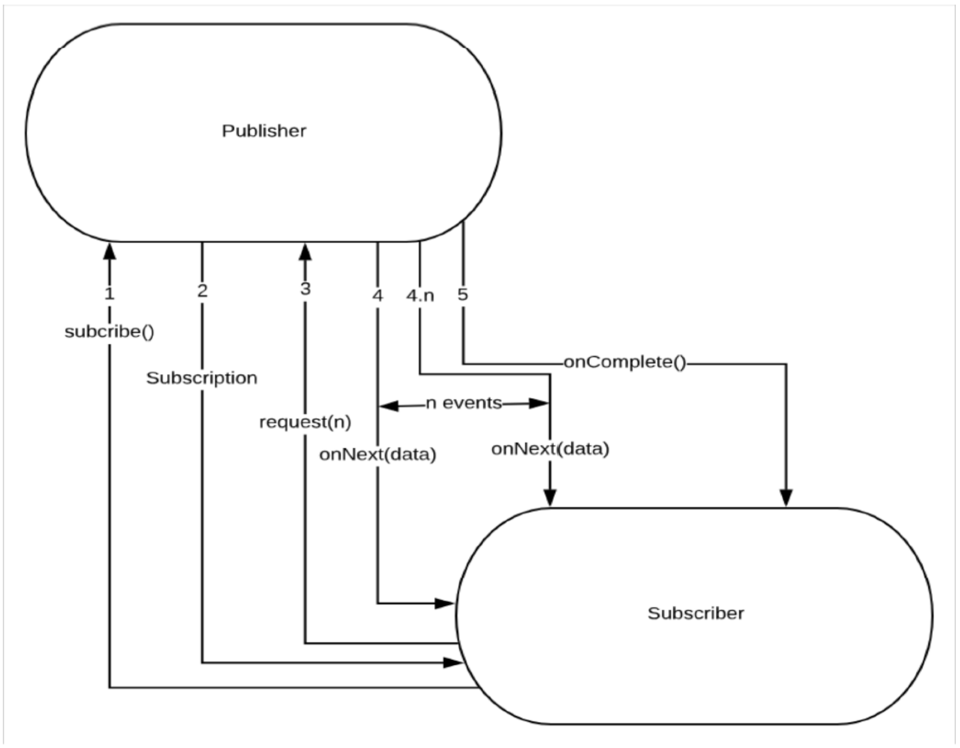
Data flow as an Event Driven stream

- No Data

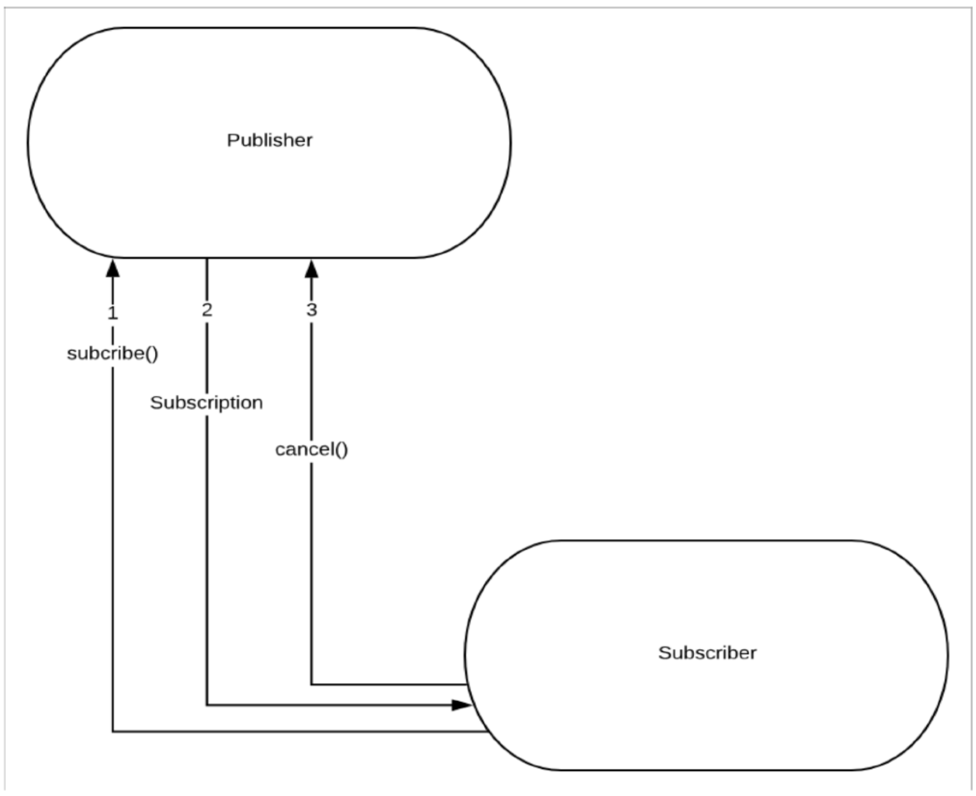




Publisher/Subscriber Event Flow



Publisher/Subscriber Event Flow



Project Reactor : REcommended Library for Spring Boot

Modules :

- reactor-core : contain implementations of Reactive Stream Specifications (interface)
- reactor-test : api to create unit tests for REactive Streams
- reactor-netty : non-blocking http server

Reactor-core provides implementation of interface in form of :

Reactive types/Streams

Flux : 0-N
Mono : 0-1

Publisher Stream (implementation of publisher)

Traditional (Imperative)

DAO implementation

```
List<User> users = userDao.getUsers();
List<String> names = new ArrayList<String>();
for(int i = 0; i<user.size(); i++)
      names.add(users.get(i).getName());
```

Functional (Java Stream: declarative)

```
List<String> names = userDao.getUsers().stream()
                                  .map(user -> user.getName())
                                  .collect(Collectors.toList());
```

if db is busy (lot of data)

take some time:

our thread will be blocked

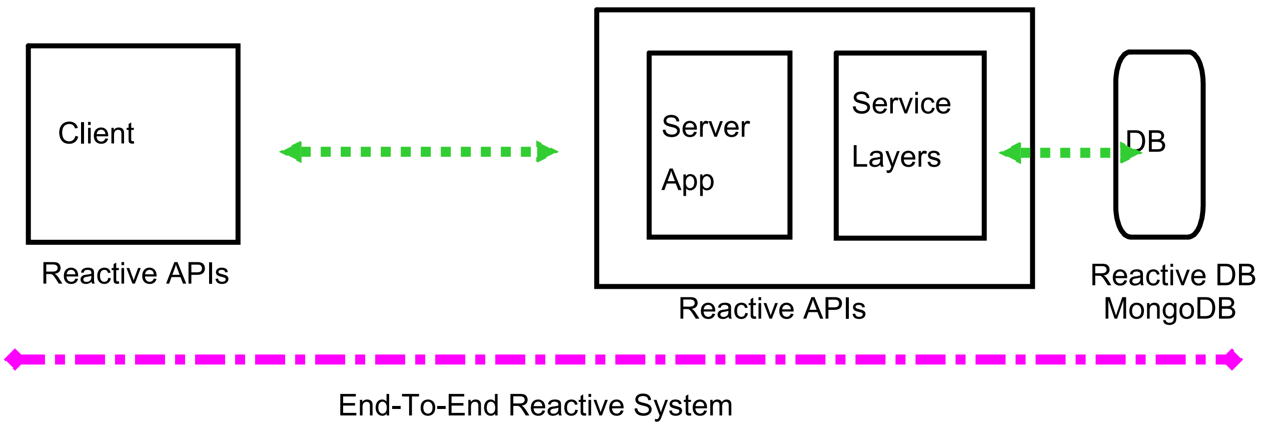
Reactive Stream

Reactive DAO implementation

```
Flux<String> names = reactiveUserDao.getUsers()
            .map(user->user.getName());
```

Non-blocking approach

To take complete advantage of Reactive Programming, a end-to-end Reactive system must be setup



Webflux web framework for Spring Boot (uses the project reactor)

reacter-core api : unit test cases

Creating a Flux Stream/ Reactive Stream/ Reactive Publisher

Flux.just(<data>) Mono.just(<single data>)

# java 8 Stream are not - reusable
# Reactive Streams (Flux) is reusable

# By default when we subscribe to Publisher (Flux/Mono),

other activities :

returning subscription, sending request(unbounded) : behind the scene

publisher will start streaming data using onNext(data) event, for each data (auto)

==> Directly access the data as a stream

Factory Methods : for creating Flux and Mono

Just like Java 8 Stream a series of activities can be associated with Reactive Stream

Filter

Map/flatMap

Combine Reactive Streams...

Restart/Retry generating reactive stream after error : multiple retry...

BackOff Retry if still OnError() is propagated : IllegalStateException

BackPressure

TO have an absolute control over backpressure, we need to provide an implementation of abstract class: BaseSubscriber

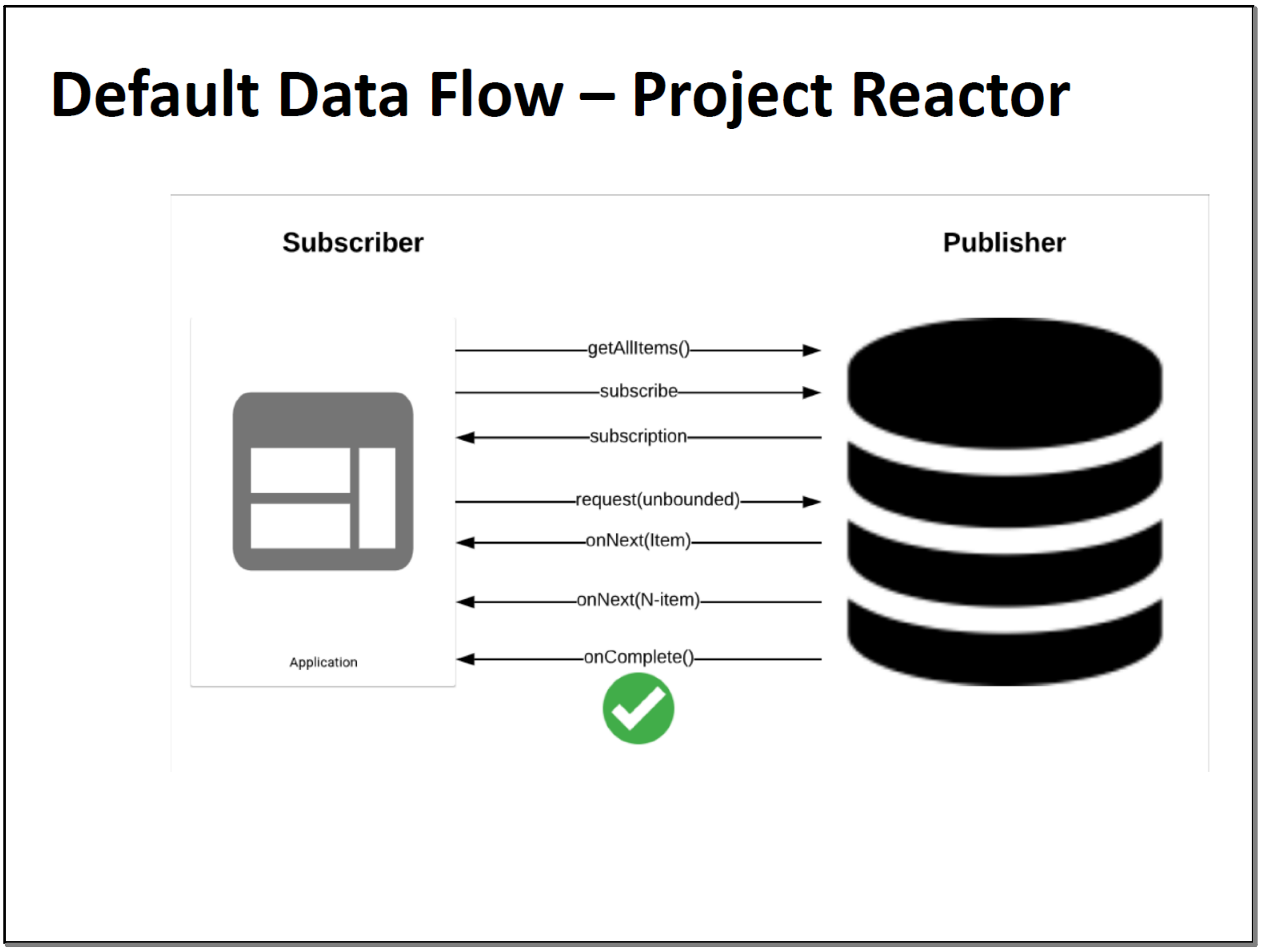
Traditional Client-server : Push based model (Server push data to client)

Push/Pull model : Client-Server both are having equal control over data flow

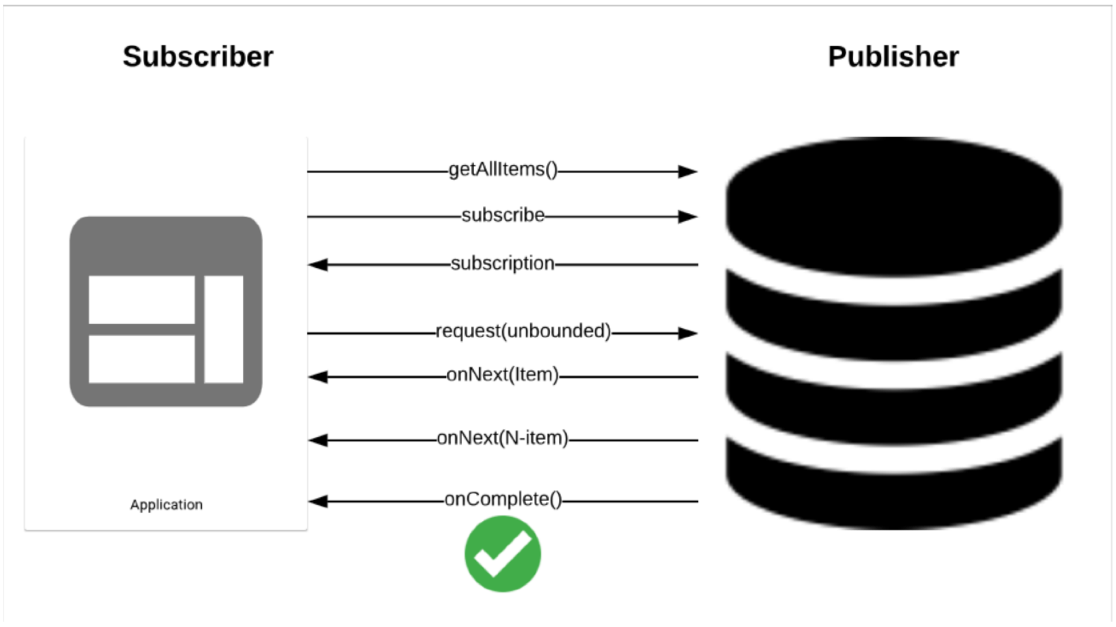
Flux (publisher)variant :

cold (default)

hot

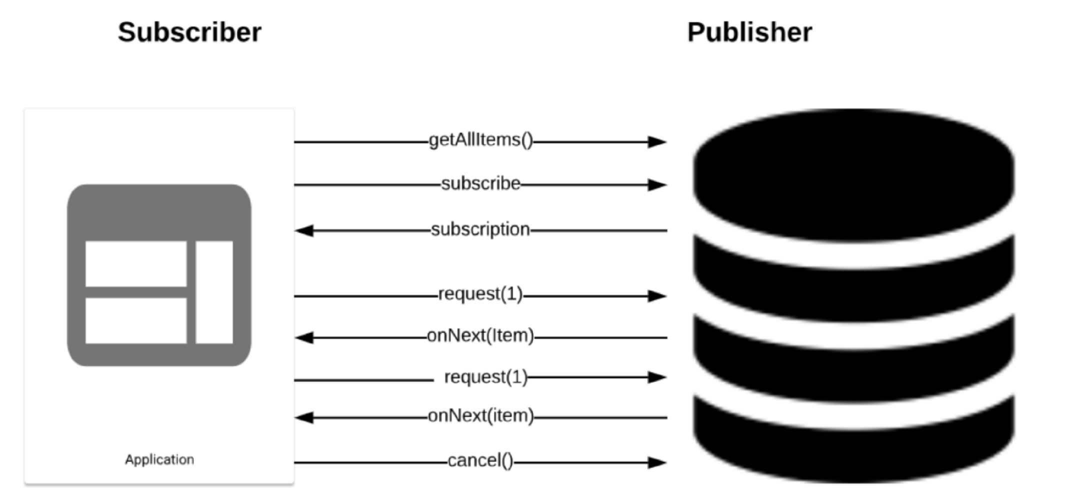


Default Data Flow – Project Reactor



What is Backpressure?

- Subscriber controls the data flow from the Publisher.



Spring Boot

Develop Reactive REST APIs

# Traditional Spring MVC does not have reactive support

# Webflux : Web MVC Reactive framework

Webflux : Two approaches of reactive development

# Similar approch of traditional MVC style (Annotation based controller)

# Functional Web

Traditional style built on top of new tech

Traditional request is converted into reactive request by netty server

Unit Testing RestAPI

Traditional MVC : Test RestTemplate

WebFlux : WebClient

1. WebClient : Reactive Client
2. WebTestClient : Unit Test for Reactive Rest Endpoints