Reactive and Asynchronous IO Programming in Java

Διπλωματική εργασία

της

Δήμητρας Μαλλιαρού

Θεσσαλονίκη, Ιούνιος 2022

Reactive and Asynchronous IO Programming in Java

Δήμητρα Μαλλιαρού

Πτυχίο Μηχανικών Πληροφορικής ΤΕ
Τεχνολογικό Εκπαιδευτικό Ίδρυμα Άρτας, 2011

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ
ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής

Μαργαρίτης Κωνσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ηη/μμ/εεεε

Ονοματεπώνυμο 1 Ονοματεπώνυμο 2 Ονοματεπώνυμο 2

Περίληψη

Τα τελευταία χρόνια, γλώσσες προγραμματισμού με χαρακτηριστικά αντιδραστικού προγραμματισμού (reactive programming), όπως ροές συμβάντων (events), ροές δεδομένων (streams), έχουν γίνει κοινές στο σχεδιασμό διαδραστικών και κατανεμημένων συστημάτων, όπως για παράδειγμα εφαρμογές Ιστού (web application) ή διεπαφές χρήστη (User Interface). Ο αντιδραστικός προγραμματισμός έχει γίνει μια δημοφιλής επιλογή στην υλοποίηση εφαρμογών, και ακόμη και οι γλώσσες που δεν είναι εγγενώς αντιδραστικές, επεκτείνονται με δομές που ακολουθούν το παράδειγμα του αντιδραστικού προγραμματισμού (Kambona, Boix & De Meuter, 2013). Το βασικό χαρακτηριστικό αυτού του παραδείγματος είναι ο προγραμματισμός ώς αντίδραση (reaction) σε συμβάντα. Τα συστατικά του προγράμματος αντιδρούν ανεξάρτητα το ένα από το άλλο, ανάλογα με τα διαθέσιμα συμβάντα.

Η ανάπτυξη διαδικτυακής εφαρμογής στη δική μας περίπτωση (full stack development) αναφέρεται στη χρήση διαφορετικών χαρακτηριστικών και εργαλείων τα οποία συνεργάζονται μεταξύ τους. Η εφαρμογή χωρίζεται σε δύο τμήματα: στο front end και στο back end. Το front end είναι υπεύθυνο για την αλληλεπίδραση του χρήστη με το πρόγραμμα. Το backend είναι υπεύθυνο για το χειρισμό της λογικής του προγράμματος, της σύνδεσης του με την βάση δεδομένων και της σύνδεσης των Web Services με το front end. Οι τεχνολογίες που χρησιμοποιούνται για την δημιουργία του front end είναι η html css Javascript και TypeScript. Συγκεκριμένα για το έργο έχει χρησιμοποιηθεί η ReactJS που είναι μια βιβλιοθήκη της JavaScript για πιο μοντέρνο προγραμματισμό. Το back end εχει υλοποιηθεί με RXJava και με το έργο Reactor που χρησιμοποιεί λειτουργίες που ανήκουν στο πλαίσιο της Reactivex.

Λέξεις κλειδιά: ReactiveX, User Interface, RXJava, Reactor project, Events, Stream, Web Application, Mobile application, Order application.

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω τις ευχαριστίες μου προς τον επιβλέπων καθηγητή Δρ. Μαργαρίτη Κωνσταντίνο που μου έδωσε πολύτιμες συμβουλές και ήταν πάντα πρόθυμος να με βοηθήσει.
Στη συνέχεια, θα ήθελα να ευχαριστήσω όλους τους καθηγητές του Πανεπιστημίου Μακεδονίας οι οποίοι μου χάρισαν γνώσεις κατά τη διάρκεια των σπουδών μου. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου που μου έδωσε υποστήριξη για την εκπλήρωση της μεταπτυχιακής μου διατριβής.

Περιεχόμενα

---

11. Βιβλιογραφία

Απόσπασμα κώδικα

Ευρετήριο Εικόνων

1 Εισαγωγή

1.1 Σκοποί - Στόχοι

Σκοπός αυτής της διπλωματικής είναι η σχεδίαση και η υλοποίηση ενός συστήματος που χρησιμοποιεί τεχνολογίες αιχμής και χαρακτηριστικά της γλώσσας Java. Απαρτίζεται από πολλές υπηρεσίες και έχει ως στόχο την παρουσίαση μιας εφαρμογής παραγγελιοληψίας με μοντέρνο και ασύγχρονο μοντέλο προγραμματισμού. Η εφαρμογή αποτελείται από δύο ρόλους, αυτό του χρήστη και αυτό του διαχειριστή. Ο χρήστης κάνει την εισαγωγή του στο σύστημα βλέπει τα προϊόντα, επιλέγει τα προϊόντα που επιθυμεί και δημιουργεί το καλάθι. Ο διαχειριστής εισάγει δεδομένα, συγκεκριμένα προϊόντα, μπορεί να δει τους χρήστες και τις αγορές τους. Όλο το σύστημα περιλαμβάνει την εισαγωγή, διαγραφή, ενημερωση και δημιουργία δεδομένων - εγγραφών στη βάση, όπως αλλιώς θα λέγαμε και στην αγγλική ορολογία CRUD (create, read, update, delete). Αυτό γίνεται με reactive programming functional router που είναι ένα είδος μοντέρνου προγραμματισμού σε αντίθεση με το παραδοσιακό μοντέλο του MVC μοντέλου (model view controller), που και τα δύο μπορούν να παράγουν ένα REST API.

1.2 Διάρθρωση την μελέτης

Το υπόλοιπο της διπλωματικής αποτελείται από το δεύτερο κεφάλαιο που αναφέρεται γενικά στο reactive programming, το τρίτο κεφάλαιο που αναφέρεται πιο συγκεκριμένα στο reactivex (reactive extension), το τέταρτο κεφάλαιο RXJava και στα χαρακτηριστικά της βιβλιοθήκης, το πέμπτο κεφάλαιο επικεντρώνεται στο τεχνολογικό υπόβαθρο της Java spring reactor framework, το έκτο κεφάλαιο εγχειρίδιο χρήστη της εφαρμογής και το πως μπορεί να την χρησιμοποιήσει, το έβδομο κεφάλαιο εγχειρίδιο διαχειριστή της εφαρμογής και ποια είναι τα βήματα για να την εγκαταστήσει, το όγδοο κεφάλαιο υλοποίηση απόσπασμα κώδικα της εφαρμογής για κάθε σελίδα, το ένατο κεφάλαιο υλοποίηση κώδικα σε Java spring boot και τέλος ο επίλογος που αναφέρεται στα συμπεράσματα που πηγάζουν από το έργο και η μελλοντική του επέκταση.

2 Reactive programming

Στην επιστήμη των υπολογιστών ο αντιδραστικός προγραμματισμός είναι ένα είδος προγραμματισμού που ασχολείται με την μετάδοση δεδομένων με ασύγχρονο τρόπο στον καταναλωτή (consumer) όταν είναι διαθέσιμος. Είναι ένα σύνολο από συμβάντα που γίνονται με τη σειρά με την πάροδο του χρόνου. Ο αντιδραστικός προγραμματισμός έχει χαρακτηριστεί ως ο τρόπος για να απλοποιήσει την αλληλεπίδραση του χρήστη με τα πραγματικού χρόνου συστήματα. Για παράδειγμα στο μοντέλο MVC ο αντιδραστικός προγραμματισμός μπορεί να διευκολύνει τις αλλαγές σε δεδομένα που αντικατοπτρίζονται αυτόματα στη σχετιζόμενη προβολή (view).

Ο αντιδραστικός προγραμματισμός αποτελείται από τρία κύρια σημεία:

Observable: Είναι ο όρος που σχετίζεται με τη μετάδοση δεδομένων (data stream). Οι observables περιέχουν δεδομένα που μεταδίδονται μεταξύ των νημάτων (threads). Ανάλογα με τις ρυθμίσεις τους δημιουργούν δεδομένα σε περιοδικό χρόνο ή ακριβώς μια συγκεκριμένη στιγμή.

Observers: Οι observers λαμβάνουν μια ρόη δεδομένων από τους observables. Οι observers εγγράφονται (subscribe) σε ένα observable χρησιμοποιώντας την μέθοδο subscribeOn() για να λάβουν τα δεδομένα που μεταδίδουν. Όλα τα δεδομένα από τον observable περνάνε μέσα από μία διαδικασία επεξεργασίας στους observers. Εάν υπάρξει κάποιο λάθος, ο observer το λαμβάνει ένα μήνυμα λάθους.

Schedulers: Ο reactive programming σχεδιάστηκε για ασύγχρονο προγραμματισμό έτσι χρειάζεται ένα σύστημα διαχείρισης νημάτων (thread). Για αυτό το λόγο δημιουργήθηκαν οι χρονοδρομολογητές (schedulers). Οι χρονοδρομολογητές είναι μέρος του αντιδραστικού προγραμματισμού και αποφασίζουν ποιος observable και observer θα τρέξει. Μπορούμε να διαμορφώσουμε ένα νήμα (thread) να εκτελέσει κάθε operator χρησιμοποιώντας τις μεθόδους subscribeOn() και/ή observerOn(). Η μέθοδος subscribeOn() εκτελεί τους operators που είναι πάνω από την αυτή και η μέθοδος observerOn() εκτελεί τους operators που είναι κάτω. Αν υπάρχει μόνο η subscribeOn() τότε όλοι οι operators εκτελούνται σε αυτό το νήμα (thread).
Ένα παράδειγμα κώδικα είναι και ο παρακάτω.

```
        Observable.just("long", "longer", "longest")
            .subscribeOn(Schedulers.io())
            .map(String::length)
            .observeOn(Schedulers.computation())
            .filter { it > 6 }
            .subscribe { length -> println("item length $length") }
        }
```

Απόσπασμα κώδικα 2.1 μέθοδοι subscribeOn, observerOn

- James Shvarts 2017 Reactive programming διαθέσιμος ο κώδικας στον ιστοτόπο https://proandroiddev.com/understanding-rxjava-subscribeon-and-observeon-744b0c6a41ea

Η εκπομπή δεδομένων just και ο operator map θα εκτελεστούν στον io scheduler, filter θα εκτελεστεί στον computation scheduler επειδή είναι κάτω από τον observerOn() operator. Υπάρχουν πολλοί operators που θα αναφερθούν στο κεφάλαιο 4 RxJava .

Μερικά παραδείγματα εφαρμογής του Reactive programming

Ο reactive programming είναι μια αποτελεσματική λύση για τις εφαρμογές με υψηλό φορτίο ή με πολλαπλούς χρήστες. Αυτές μπορεί να είναι:

- συνομιλίες και κοινωνικά δίκτυα
- παιχνίδια
- εφαρμογές βίντεο και ήχου
- μετάδοση δεδομένων πραγματικού χρόνου
- επαυξημένη πραγματικότητα ή μηχανική μάθηση
- server με κλήσεις πολλαπλών χρηστών

Για παράδειγμα οι προγραμματιστές του Netflix χρησιμοποιούν την ανοιχτού κώδικα reactive programming βιβλιοθήκη της Microsoft για τις εφαρμογές τους. Οι ReactiveX μπορούν να αναπτύξουν σύνθετες εφαρμογές που δουλεύουν στο παρασκήνιο πολύ πιο εύκολα. Οι εφαρμογές όπως η Netflix είναι αντιδραστικές καθώς ο κώδικας αντιδρά σε πολλά συμβάντα (τα κλικ του ποντικιού, πάτημα πλήκτρων, ασύγχρονη μετάδοση δεδομένων από τον server). Ακόμα ένα παράδειγμα είναι η εφαρμογές IoT με arduino και αισθητήρες θερμοκρασίας - υγρασίας σε εσωτερικό χώρο στέλνοντας τα δεδομένα σε κάποιον server σε πραγματικό χρόνο.

- **Tom Nolle Reactive Programming**
  https://www.techtarget.com/searchapparchitecture/definition/reactive-programming

3 ReactiveX

Το πιο διαδεδομένο API για τον αντιδραστικό προγραμματισμό είναι το reactiveX (reactive extension) ή για συντομία Rx. Η reactivex είναι ένας συνδυασμός των καλύτερων ιδεών από το μοτίβο του Observer, το μοτίβο της επανάληψης και το μοτίβο του συναρτησιακού προγραμματισμού. Έχει εμπνεύσει πολλά άλλα API, frameworks και γλώσσες προγραμματισμού.

Στο framework ReactiveX δύο είναι οι πρωταγωνιστές: Ο Observable, εκπέμπει συμβάντα σε μια φόρμα δεδομένων αντικειμένων και ο Observer, εγγράφεται στον Observable για να λαμβάνει τα συμβάντα. Πολλαπλοί Observers μπορούν να εγγραφούν σε έναν Observable, ο Observable διαχειρίζεται τα μεταδιδόμενα δεδομένα με operators. Ένα ακόμα κύριο χαρακτηριστικό της ReactiveX είναι ο συναρτησιακός προγραμματισμός, ο οποίος διευκολύνει την εκτέλεση της εφαρμογής και βελτιώνει το χειρισμό των λαθών.

Αν και η ReactiveX έχει καλά καθορισμένες έννοιες και το API πλέον είναι ώριμο, αρχικά οι προγραμματιστές που το βλέπουν για πρώτη φορά δυσκολεύονται με τις προηγμένες έννοιες. Δεν είναι εύκολο να αλλάξει η λογική του κλασικού προγραμματισμού που έχουν οι προγραμματιστές με αυτή του συναρτησιακού προγραμματισμού. Η εκμάθηση της ReactiveX είναι καλύτερη όταν γίνεται με πραγματικά παραδείγματα κώδικα παρά με την θεωρία. Η εξοικείωση με την Java streams βοηθάει πολύ, αλλά χρειάζεται και η κατανόηση με τις βασικές έννοιες της RX.

Το πρόγραμμα που θα ακολουθήσει έχει αναπτυχθεί σε RxJava και Reactor framework spring boot, αλλά οι έννοιες και οι τεχνικές της ReactiveX είναι ίδιες άσχετα αν εφαρμόζεται σε διαφορετικές γλώσσες προγραμματισμού.

Με την χρήση της reactivex βιβλιοθήκης, προστέθηκαν χαρακτηριστικά της reactive σε γλώσσες προγραμματισμού, Swift: RxSwift, Java: RxJava, JavaScript: RxJS ,C#: Rx.NET, C#(Unity): UniRx, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Lua: RxLua, Ruby: Rx.rb, Python: RxPY, Go: RxGo, Groovy: RxGroovy, JRuby: RxJRuby, Kotlin: RxKotlin, PHP: RxPHP, Elixir: reaxive, Dart: RxDart.

- ReactiveX Pınar Koçak Mar 27 2022
  https://medium.com/@pinarkocak/reactive-programming-reactivex-ee01167f19fe
- Adam Jodłowski https://x-team.com/blog/introduction-reactivex-android/

4

RxJava

Η RxJava είναι μια βιβλιοθήκη που εφαρμόζει τις λειτουργίες της Reactive Extension και συνδυάζει τον ασύγχρονο προγραμματισμό με τα events που γίνονται είτε από χρήστες στο UI είτε από το ίδιο το πρόγραμμα για την μετάδοση δεδομένων, για παράδειγμα από έναν σέρβερ.

Όπως είδαμε και παραπάνω στην αναφορά που έγινε για τον αντιδραστικό προγραμματισμό, αυτός χρησιμοποιεί τις κλάσεις observables και subscribers. Η observable χρησιμοποιείται για την εκπομπή δεδομένων και η subscriber χρησιμοποιείται για την κατανάλωση αυτών.

Η RxJava εργάζεται ως εξής: Η subscriber κάνει εγγραφή (subscribe) στην Observable, η observable καλεί την Subscriber.onNext() για κάθε αριθμό από δεδομένα, εάν συμβεί κάποιο λάθος καλείται η Subscriber.onError() και εάν όλα είναι καλά καλείται η Subscriber.onCompleted().

Ας δώσουμε κάποια παραδείγματα. Στο πρώτο παράδειγμα δημιουργούμε ένα απλό Observable και Subscriber, θα τα συνδέσουμε και θα δούμε το αποτέλεσμα. Φτιάχνουμε έναν Observable.

```
Observable myObservable  = Observable.create(new Observable.OnSubscribe() {
      @Override
      public void call(Subscriber<? super String> subscriber) {
          subscriber.onNext(“Blue Factory”);
          subscriber.onComplete();
      }
});
```

Απόσπασμα κώδικα 4.1 αντικείμενο Observable

Το αντικείμενο που δημιουργήσαμε μεταδίδει το λεκτικό Blue Factory και μετά ολοκληρώνει. Το Αλφαριθμητικό (String) λαμβάνεται από την μέθοδο onNext() της Subscriber. Ας φτιάξουμε τώρα ένα αντικείμενο subscriber.

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onComplete() {
        System.out.println(“I am done”);
    }
```

```
    @Override
    public void onError(Throwable e){

    }
    @Override
    public void onNext(String s){
        System.out.println(s);
    }
};
```

Απόσπασμα κώδικα 4.2 αντικείμενο Subscriber

Ο subscriber λαμβάνει την τιμή του String με την μέθοδο onNext() και το εκτυπώνει, επίσης το String “I’m done” εκτυπώνεται όταν καλείται η μέθοδος onCompleted(). Με αυτό το κώδικα συνδέονται αυτά τα δύο αντικείμενα.

```
myObservable.subscribe(mySubscriber);
```

Απόσπασμα κώδικα 4.3 σύνδεση Observable Subscriber

Ο παραπάνω κώδικας μπορεί να γίνει πιο μικρός χρησιμοποιώντας τις μεθόδους όπως Observable.just() και Observable.from(). Η μέθοδος Observable.just() μεταδίδει μόνο ένα στοιχείο ενώ η μέθοδος Observable.from() μεταδίδει από ένα μέχρι περισσότερα στοιχεία. Το δεύτερο παράδειγμα που ακολουθεί αναφέρεται στη χρήση της Observable.just().

```
Observable myObservable = Observable.just(“Blue Factory”);
```

Απόσπασμα κώδικα 4.4 δημιουργία Observable με just

Φτιάχνουμε και το αντικείμενο Subscriber:

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onComplete() {
        System.out.println(“I am done”);
    }
    @Override
    public void onError(Throwable e){

    }
    @Override
    public void onNext(String s){
```

```
        System.out.println(s);
    }
};
```

Απόσπασμα κώδικα 4.5 δημιουργία Subscriber

και τα συνδέουμε με τον παρακάτω κώδικα:

```
myObservable.subscribe(mySubscriber);
```

Απόσπασμα κώδικα 4.6 σύνδεση Observable Subscriber

Το αποτέλεσμα είναι: I/System.out: Blue Factory και I/System.out: I'm done.

Στο τρίτο παράδειγμα έχουμε μια λίστα με αλφαριθμητικά (String) και θα χρησιμοποιήσουμε την μέθοδο Observable.from(). Η μέθοδος αυτή λαμβάνει όλα τα στοιχεία της λίστας και τα μεταδίδει ένα προς ένα. Δημιουργούμε την λίστα με τα αλφαριθμητικά :

```
ArrayList myStringArray = new ArrayList<>();
myStringArray.add("Blue");
myStringArray.add("Factory");
myStringArray.add("Blog");
myStringArray.add("Post");
```

Απόσπασμα κώδικα 4.7 δημιουργία λίστας αλφαριθμητικών

Δημιουργούμε την μέθοδο Observable.from()

```
Observable myObservable = Observable.from(myStringArray);
```

Απόσπασμα κώδικα 4.8 δημιουργία Observable με from

Στον παραπάνω κώδικα η μέθοδος παίρνει ως παράμετρο την λίστα με τα αλφαριθμητικά. Παρακάτω δημιουργούμε το αντικείμενο Subscriber και θα το συνδέσουμε με το Observable.

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onComplete() {
        System.out.println("I am done");
    }
    @Override
```

```
    public void onError(Throwable e){

    }
    @Override
    public void onNext(String s){
        System.out.println(s);
    }
};
myObservable.subscribe(mySubscriber);
```

Απόσπασμα κώδικα 4.9 δημιουργία Subscriber και σύνδεση με Observable

Η RxJava όπως είδαμε πριν έχει κάποιους τελεστές που δρουν στον Observable και μεταφέρουν δεδομένα στους observers. Όταν υπάρχουν πολλοί τελεστές μαζί, κάθε τελεστής τελειώνει την εργασία του και περνάει τα μεταδιδόμενα δεδομένα στον επόμενο τελεστή. Θα δούμε ένα απλό και σύντομο κώδικα με τους τελεστές map και filter σε lambda. Ας δούμε πρώτα για τον τελεστές map()
import rx.Observable;

```
public class RxOperators {

   public static void main(String[] args)
   {

       //map operator
       Observable<String> mapObservable = Observable.just("hello world","the
observable emits lower case sentences","subscriber sees it as upper case","map
operator");
       mapObservable.map(String::toUpperCase).subscribe(System.out::println);
   }
}
```

Απόσπασμα κώδικα 4.10 τελεστής map μετατροπή αλφαριθμητικού σε κεφαλαία

Εκτυπώνει κεφαλαία λόγο του τελεστή map()
//HELLO WORLD
//THE OBSERVABLE EMITS LOWER CASE SENTENCES
//SUBSCRIBER SEES IT AS UPPER CASE
//MAP OPERATOR

Ο τελεστής filter()

```
Observable<String> filterObservable = Observable.from(new String[]{"Hello","How
are you?", "doing"});
```

```
    filterObservable.filter(string->string.contains("
")).subscribe(System.out::println);
```

Απόσπασμα κώδικα 4.11 τελεστής filter

Εκτυπώνει αλφαριθμητικό το οποίο έχει κενό ανάμεσα στις λέξεις
//How are you?

Άλλοι τελεστές είναι οι εξής: take, count, skip startWith, reduce, repeat, scan, all, contains, elementAt, distinct, toList, toSortedList, concat, merge, zip, debounce, delay θα δούμε κάποια παραδείγματα για μερικά από αυτά.

```
    Observable<Integer> takeObservable = Observable.range(0,100);
    takeObservable.take(5).subscribe(System.out::println);
```

Απόσπασμα κώδικα 4.12 τελεστής take

εκτυπώνει από 0 μέχρι 4

Ο τελεστής count επιστρέφει έναν αριθμό από τιμές που μπορεί να φτάσει ο subscriber.

```
Observable<String> countObservable = Observable.from(new
String[]{"First","Second", "Third", "Seventh"});
countObservable.filter(string->
string.length()>5).count().subscribe(System.out::println);
```

Απόσπασμα κώδικα 4.13 τελεστής count

εκτυπώνει τον αριθμό 2 που σημαίνει ότι μόνο δύο αλφαριθμητικά έχουν μήκος πάνω από 5.

Ο τελεστής onError επιστρέφει ως τιμή μήνυμα λάθους διότι έχουμε περάσει μέσα αντί για αλφαριθμητικό την τιμή null.

```
Observable<String> countObservable = Observable.from(new
String[]{"First","Second", "Third", null});
countObservable.filter(string->
string.length()>5).count().subscribe(System.out::println, throwable ->
System.out.println("One of the values is not valid"));
```

Εκτυπώνει το μήνυμα λάθος από το throwable
//One of the values is not valid

Ο τελεστής skip προσπερνάει τις τιμές που θα οριστούν ως όρισμα στην μέθοδο skip() και θα εκτυπώσει τα υπόλοιπα.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        Observable<Integer> skipObservable = Observable.from(numbers);
skipObservable.skip(3).subscribe(System.out::println);
```

Απόσπασμα κώδικα 4.14 τελεστής skip

εκτυπώνει 4 and 5

Ο τελεστης startWith προσθέτει το κείμενο που δίνεται στην αρχή της μετάδοσης των δεδομένων

```
Observable<String> startWithObservable = Observable.just(" Rx", "Java", "
Operators", " Tutorial");
    startWithObservable.startWith("Welcome to
the").subscribe(System.out::print);
```

Απόσπασμα κώδικα 4.15 τελεστής startWith

Εκτυπώνει
//Welcome to the RxJava Operators Tutorial

Ο τελεστής delay ο οποίος καθυστερεί να ξεκινήσει την μετάδοση των τιμών από τον observable για ένα συγκεκριμένο διάστημα.

```
      Observable.just(1,2,3,4,5,6).delay(5, TimeUnit.SECONDS)
      .subscribe(System.out::println, System.out::println, () ->
System.out.print("OnComplete"));Thread.sleep(4000);
```

Απόσπασμα κώδικα 4.16 τελεστής delay

Ο παραπάνω κώδικας δείχνει ότι τίποτα δεν συμβαίνει επειδή η main μέθοδο θα επιστρέψει μετά από 4 δευτερόλεπτα αποτρέποντας την μετάδοση που θα γινόταν μετά από 5 δευτερόλεπτα.

Ο flatmap τελεστής καλεί μια συνάρτηση που επιστρέφει έναν observable, δηλαδή μετατρέπει ένα εκπεμπόμενο στοιχείο από ένα Observable μέσω μιας συνάρτησης σε έναν άλλο Observable. Η τελική μετάδοση αυτών των στοιχείων δε γίνεται σε μια συγκεκριμένη σειρά. Για παράδειγμα ένας Observable με αλφαριθμητικα με την χρήση της flatmap παίρνει κάθε ένα αλφαριθμητικό και το βάζει ως όρισμα στην μέθοδο performLongOperation και επιστρέφει το μήκος του αλφαριθμητικού σε Observable.

```
public static void main(String[] args) throws InterruptedException {
    Observable.just("long", "longer", "longest")
        .flatMap(v ->
    performLongOperation(v)
            .doOnNext(s -> System.out.println("processing item on thread " +
Thread.currentThread().getName()))
            .subscribeOn(Schedulers.newThread()))
        .subscribe(length -> System.out.println("received item length " + length + " on
thread " + Thread.currentThread().getName()));

    Thread.sleep(10000);
}
/**
 * Returns length of each param wrapped into an Observable.
 */
protected static Observable<Integer> performLongOperation(String v) {
    Random random = new Random();
    try {
        Thread.sleep(random.nextInt(3) * 1000);
        return Observable.just(v.length());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return null;
}
```

Απόσπασμα κώδικα 4.17 τελεστής flatmap

Το αποτέλεσμα από αυτό το κώδικα θα είναι :

```
processing item on thread RxNewThreadScheduler-3
processing item on thread RxNewThreadScheduler-1
processing item on thread RxNewThreadScheduler-2
received item length 7 on thread RxNewThreadScheduler-3
received item length 4 on thread RxNewThreadScheduler-1
received item length 6 on thread RxNewThreadScheduler-2
```

Παρόμοιος τελεστής είναι και ο concatmap με την διαφορά ότι τα δεδομένα π χ αλφαριθμητικά που επιστρέφονται ως Observables μεταδίδονται σε μία διάταξη όπως ήταν αρχικά στον Observable που τα είχε δημιουργήσει. Ως παράδειγμα παίρνουμε τον παραπάνω κώδικα με τη διαφορά ότι αντί για flatmap έχουμε concatmap καλώντας την performLongOperation που επιστρέφει Observable.

```
public static void main(String[] args) throws InterruptedException {
    Observable.just("long", "longer", "longest")
        .concatMap(v ->
    performLongOperation(v)
            .doOnNext(s -> System.out.println("processing item on thread " +
Thread.currentThread().getName()))
            .subscribeOn(Schedulers.newThread()))
        .subscribe(length -> System.out.println("received item length " + length + " on
thread " + Thread.currentThread().getName()));

    Thread.sleep(10000);
}
```

Απόσπασμα κώδικα 4.18 τελεστής concatMap

Το αποτέλεσμα θα είναι :

processing item on thread RxNewThreadScheduler-1
received item length 4 on thread RxNewThreadScheduler-1
processing item on thread RxNewThreadScheduler-2
received item length 6 on thread RxNewThreadScheduler-2
processing item on thread RxNewThreadScheduler-3
received item length 7 on thread RxNewThreadScheduler-3

Παραδείγματα με αποσπάσματα κώδικα υπάρχουν στους παρακάτω ιστότοπους:

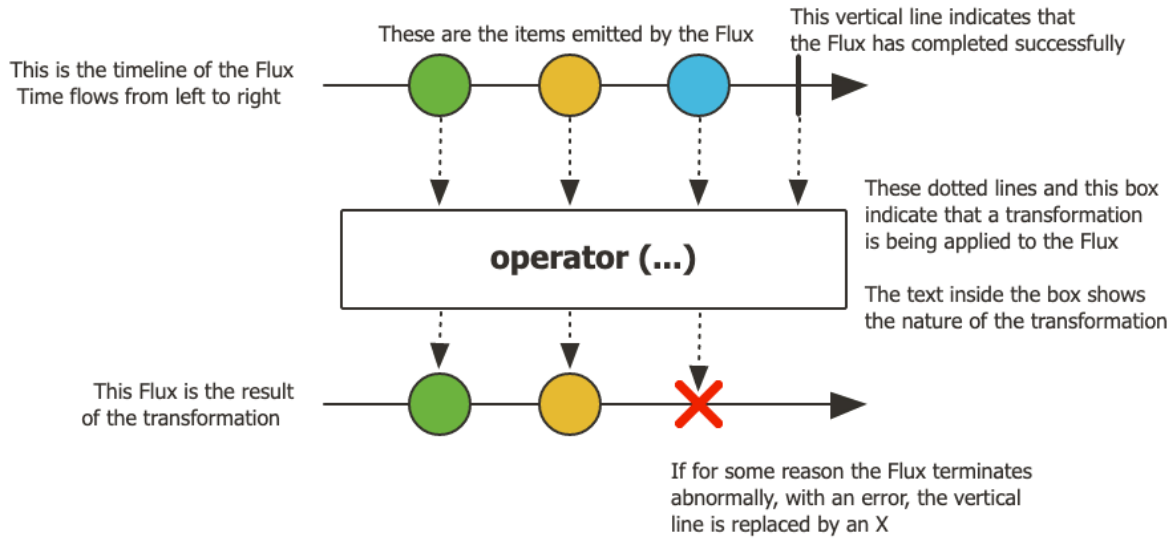
- https://www.section.io/engineering-education/rxjava-operators/
- https://reactivex.io/documentation/operators.html
- https://factoryhr.medium.com/understanding-java-rxjava-for-beginners-5eacb8de12ca

5

Τεχνολογικό Υπόβαθρο

5.1 Project reactor

Η reactor project είναι μια βιβλιοθήκη της java που εισάγει έναν συνδυασμό από ιδιότητες της reactiveX που υλοποιούνται στην κλάση publisher. Επίσης διαθέτει ένα μεγάλο αριθμό από τελεστές όπως Flux και Mono. Το αντικείμενο Flux αντιπροσωπεύει μια ακολουθία από στοιχεία 0…N. Ενώ το αντικείμενο Mono αντιπροσωπεύει μια μόνο τιμή ή τίποτα σαν αποτέλεσμα.
Η παρακάτω εικόνα δείχνει πώς μεταμορφώνει τα δεδομένα :

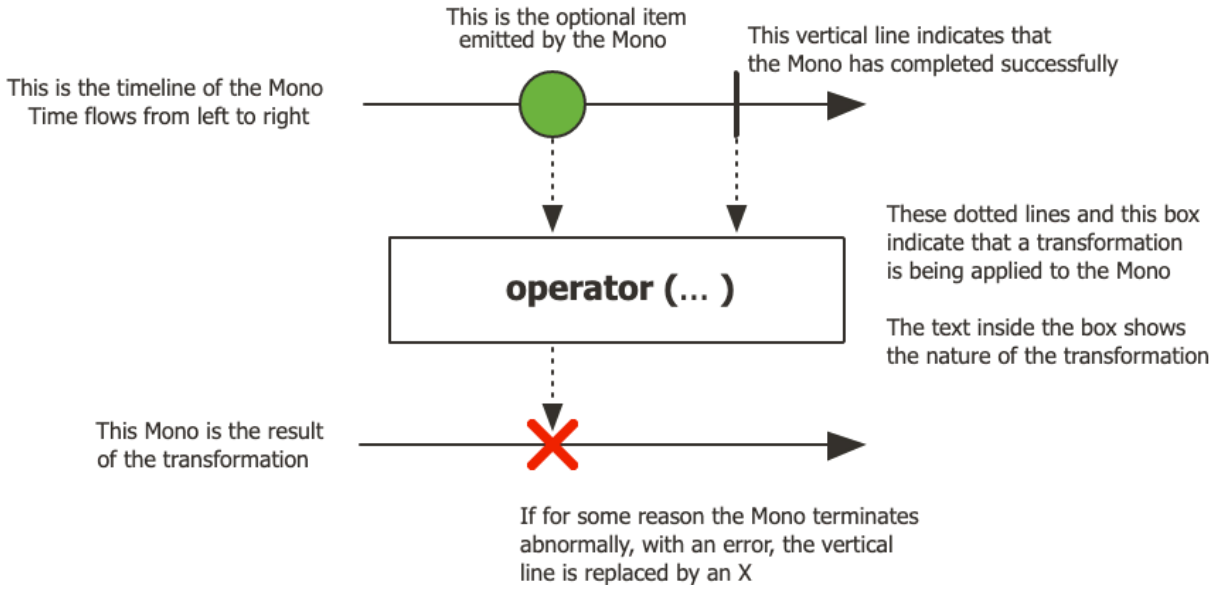


Εικόνα 5.1 περιγραφή της λειτουργίας του τελεστή Flux

- Flux διάγραμμα διαθέσιμο στον ιστότοπο : https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

Ένα Flux<T> αντικείμενο όπως αναφέρθηκε είναι μια ασύγχρονη ακολουθία στοιχείων 0...N και σταματάει όταν τελειώνει η μετάδοση. Στέλνει είτε ένα σήμα ολοκλήρωσης είτε ένα σήμα λάθους, OnNext(), onError(), onComplete().

Ένα Mono<T> είναι ένα ασύγχρονο αποτέλεσμα από 0..1
Η ακόλουθη εικόνα δείχνει πως το Mono μεταμορφώνει ένα στοιχείο :



Εικόνα 5.2 περιγραφή της λειτουργίας του τελεστής Mono

- Mono διαγραμμα διαθέσιμο στον ιστότοπο: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html

Το Mono<T> μεταδίδει το ένα στοιχείο μέσω της μεθόδου onNext και τερματίζει με ένα σήμα ολοκλήρωσης (completion) με επιτυχία ή με ένα σήμα λάθος (onError). Το Mono διαθέτει μόνο ένα υποσύνολο τελεστών από το Flux και μερικοί τελεστές από το Mono κάνουν μετάβαση σε Flux. Για παράδειγμα: Flux<Integer> fluxOfIntegers = evenNumbers.concatWith(oddNumbers); ενώ από ένα Mono επιστρέφει ένα Mono για παράδειγμα: session.changeSessionId().then(thenTask1()).

Για να φτιάξουμε ένα Flux και ένα Mono είναι να χρησιμοποιήσουμε μια από τις μεθόδους from και just. Για παράδειγμα αν θέλουμε να φτιάξουμε μια ακολουθία από αλφαριθμητικά (String) είτε απαριθμώντας τα είτε βάζοντας τα σε ένα Collection όπως δείχνει ο παρακάτω κώδικας :

```
Flux<String> seq1 = Flux.just(“foo”,”bar”,”foobar”);
List<String> iterable = Arrays.asList(“foo”,”bar”,”foobar”);
Flux<String> seq2 - Flux.fromIterable(iterable);
```

Απόσπασμα κώδικα 5.1 δημιουργία Flux με just και from

Για να φτιάξουμε ένα Mono χρησιμοποιούμε τις αντίστοιχες μεθόδους όπως δείχνει παρακάτω ο κώδικας :

```
Mono<String> noData = Mono.empty();
Mono<String> data = Mono.just(“foo”);
```

Απόσπασμα κώδικα 5.2 δημιουργία Mono με empty, just

Η πρώτη παράμετρος της μεθόδου Flux.range δείχνει η θέση από όπου ξεκινάει η μετάδοση των στοιχείων και η δεύτερη παράμετρος δείχνει τον αριθμό των στοιχείων που μεταδίδει.

Τέλος φτιάχνουμε ένα Flux για την μετάδοση των δεδομένων χρησιμοποιώντας την subscribe, η εκτύπωση γίνεται με την system.out.println μέσω της onNext μεθόδου. Εάν προκύψει ένα λάθος εκτυπώνεται error μέσω της onError μέθοδο και Done όταν ολοκληρωθεί μέσω της μεθόδου onComplete. Το κομμάτι κώδικα είναι γραμμένο με lambda expressions.

```
Flux<Integer> numbersFromFiveToSeven = Flux.range(5,3);
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(i -> System.out.println(i),
   error -> System.err.println("Error " + error),
   () -> System.out.println("Done"));
```

Απόσπασμα κώδικα 5.3 τελεστής range

5.2 Repositories

Η σχεσιακή βάση δεδομένων R2DBC ανήκει στο περιβάλλον της Spring Data και υλοποιείται με τα repositories. Το R2DBC σημαίνει Reactive Relational Database Connectivity, χρησιμοποιεί reactive drivers (προγράμματα) για να ενσωματώσει SQL βάσης δεδομένων όπως mariadb, postgresql, mysql και άλλες. Δημιουργεί reactiveX εφαρμογές χρησιμοποιώντας τεχνολογίες σχεσιακών βάσεων δεδομένων χωρίς να έχει τα χαρακτηριστικά του ORM framework (πλαίσιο λογισμικού) όπως lazy-loading, cash, write-behind. Η R2DBC είναι πιο απλή και βασίζεται στις ασύγχρονες μεθόδους συνδεσης με τις βάσεις δεδομένων.

Οι βάσεις δεδομένων που μπορεί να συνδεθεί είναι :

1. H2
2. MariaDB
3. Microsoft SQL Server
4. MySQL
5. Postgres
6. Oracle

Ακολουθεί ένα παράδειγμα κώδικα repository με ερωτήματα SQL:

```
interface PersonRepository extends ReactiveCrudRepository<Person, String> {
        Flux<Person> findByFirstname(String firstname);
        @Modifying
        @Query("UPDATE person SET firstname = :firstname where lastname =
:lastname")
```

```
        Mono<Integer> setFixedFirstnameFor(String firstname, String lastname);

        @Query("SELECT * FROM person WHERE lastname = :#{[0]}")
        Flux<Person> findByQueryWithExpression(String lastname);
}
```

Απόσπασμα κώδικα 5.4 δημιουργια r2dbc repository

Η Εγκατάσταση της Spring reactive web, MariaDB driver, Spring Data R2DBC και Lombok γίνεται στο περιβάλλον maven. Στο αρχείο application.properties γράφουμε τις παρακάτω πληροφοριές :
spring.r2dbc.url=r2dbc:mariadb://127.0.0.1:3306/todo το κατάλληλο R2DBC URL περιλαμβάνει πληροφορίες σχετικά με το σχήμα της R2DBC, τον MariaDB driver, το host (εξυπηρετητής), port (πόρτα) και το όνομα της βάσης δεδομένων.
spring.r2dbc.username=app_user το username της βάσης δεδομένων
spring.r2dbc.password=Password123! το password της βάσης δεδομένων

5.3 Functional routers

Οι Spring MVC εφαρμογές παραδοσιακά χρησιμοποιούν annotations όπως @GetMapping, @PostMapping για να χαρτογραφήσουν διαδρομές αιτημάτων (paths requests) στους ελεγκτές (controllers). Οι μέθοδοι δρομολογητών (functional routing) που από εδώ και στο εξής θα χρησιμοποιούμε τον αγγλικό όρο είναι ένας εναλλακτικός τρόπος για αυτή τη χαρτογράφηση. Η Functional Router είναι μέθοδος που οδηγεί στη μέθοδο διαχείρισης (handler method). Παρακάτω δίνεται ένα παράδειγμα κώδικα Functional router:

```
        @Configuration
        public class MyRoutes {

            @Bean
            RouterFunction<ServerResponse> home() {
                return route(GET("/"), request -> ok().body(fromValue("Home page")));
            }
            @Bean
            RouterFunction<ServerResponse> about() {
                return route(GET("/about"), request -> ok().body(fromValue("About
page")));
            }
        }
```

Απόσπασμα κώδικα 5.5 δημιουργία functional router

Ο κώδικας της functional router είναι απλός και κομψός. Εδώ επιστρέφει μια τιμή - κείμενο στην Homepage και στην About Page.

Αντίστοιχος κώδικας υπάρχει και σε junit test για να ελέγξουμε τη λειτουργικότητα του προγράμματος.

```
@Autowired
    private WebTestClient client;

    @Test
    public void test_home_page() {

        client.get().uri("/").exchange().expectStatus().isOk()
                .expectBody(String.class).isEqualTo("Home page");
    }

    @Test
    public void test_about_page() {

        client.get().uri("/about").exchange().expectStatus().isOk()
                .expectBody(String.class).isEqualTo("About page");
    }
```

Απόσπασμα κώδικα 5.6 webtestclient

5.4 Controllers

Ένας άλλος τρόπος για τη δημιουργία spring webflux διαδικτυακής εφαρμογής είναι οι controllers με τα annotations εκτός από το Functional endpoints που έχουμε δει. Χρησιμοποιούμε τα annotations @Controllers, @RestControllers για τη δημιουργία εφαρμογής με τον ίδιο τρόπο όπως και στην spring boot MVC. Τα annotations που χρησιμοποιούνται επίσης για τα HTTP request είναι το @PostRequest, @GetRequest, @PutRequest, @DeleteRequest. Για να δημιουργήσουμε functional endpoints και controllers προσθέτουμε το annotation @EnableWebFlux στην configuration class (κλάση διαμόρφωσης). Παρακάτω δίνεται ένα παράδειγμα κώδικα δημιουργίας controller με http service σε σύνδεση με μία βάση βιβλίων από την οποία τραβάει μια λίστα με βιβλία, ένα συγκεκριμένο βιβλίο με

βάση το id του, εγγραφή βιβλίου στη βάση, ανανέωση βιβλίου στη βάση και διαγραφή βιβλίου από τη βάση.

```
@RestController
public class BookController {
	@Autowired
	private BookService bookService;

	@GetMapping(value="/books", produces = {
MediaType.APPLICATION_JSON_VALUE })
	public Mono<ResponseEntity<List<Book>>> getAllBooks() {
		return bookService.getAllBooks()
				.map(list -> new ResponseEntity<List<Book>>(list,
HttpStatus.OK));
	}
	@GetMapping("/books/{id}")
	public Mono<ResponseEntity<Book>> getBookById(@PathVariable("id")
Integer id) {
		return bookService.getBookById(id)
				.map(book -> new ResponseEntity<Book>(book,
HttpStatus.OK))
				.defaultIfEmpty(new
ResponseEntity<>(HttpStatus.NOT_FOUND));
	}
	@PostMapping(value = "/add", consumes = {
MediaType.APPLICATION_JSON_VALUE })
	public Mono<ResponseEntity<Void>> addBook(@RequestBody Book book,
UriComponentsBuilder builder) {
		return bookService.addBook(book)
				.map(newBook -> {
				  HttpHeaders headers = new HttpHeaders();

headers.setLocation(builder.path("/books/{id}").buildAndExpand(newBook.getId()).t
oUri());
					return new ResponseEntity<Void>(headers,
HttpStatus.CREATED);
				 });
	}
	@PutMapping(value = "/update", consumes = {
MediaType.APPLICATION_JSON_VALUE })
	public Mono<ResponseEntity<Book>> updateBook(@RequestBody Book
book) {
		return bookService.updateBook(book)
```

```
                                .map(modBook -> new
ResponseEntity<Book>(modBook, HttpStatus.OK));
        }
        @DeleteMapping("/books/{id}")
        public Mono<ResponseEntity<Void>> deleteBookById(@PathVariable("id")
Integer id) {
                return bookService.deleteBookById(id)
                                .map(val -> {
                                        if (val == true) {
                                          return new
ResponseEntity<Void>(HttpStatus.NO_CONTENT);
                                        }
                                        return new
ResponseEntity<Void>(HttpStatus.NOT_FOUND);
                                });
        }
}
```

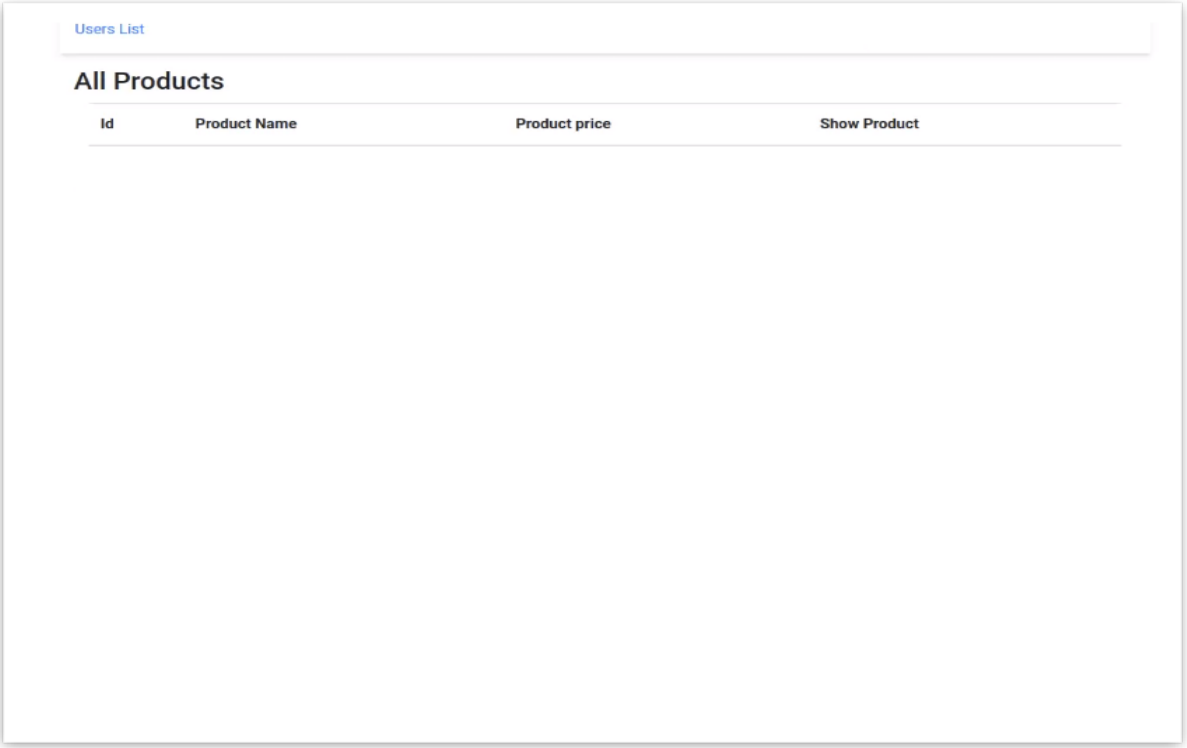
Απόσπασμα κώδικα 5.7 δημιουργία controller

Διαθέσιμοι οι κώδικες και οι ορισμοί από τους ιστοτόπους:

- By Arvind Rai, September 23, 2020
  https://www.concretepage.com/spring-5/spring-webflux-controller

6 Εγχειρίδιο χρήστη

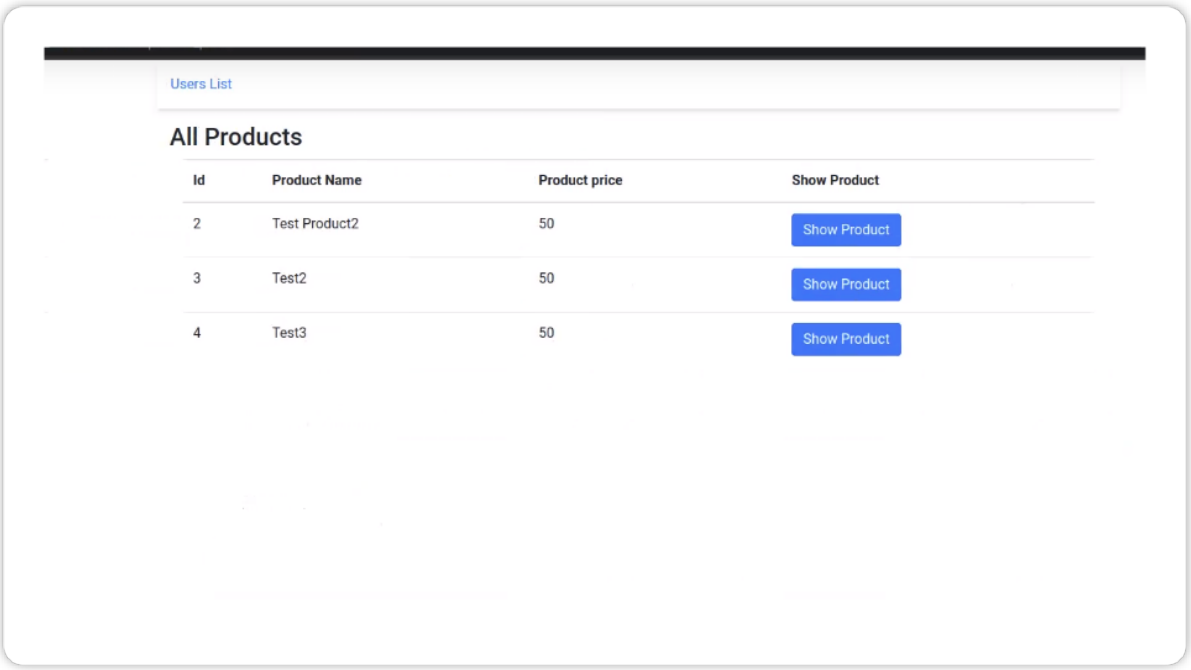
6.1 Products - Λίστα προϊόντων



Εικόνα 6.1 εμφάνισης σελίδας χωρίς προϊόντα

Στην παραπάνω εικόνα εμφανίζεται η αρχική σελίδα προορισμού του ιστότοπου, για όποιον την επισκέπτεται. Προς το παρόν ο ιστότοπος υποστηρίζει μόνο την ελληνική γλώσσα, επομένως όλα θα εμφανίζονται σε αυτήν τη γλώσσα. Όμως, όπως περιγράφεται στο κεφάλαιο «Συμπεράσματα και Μελλοντικές Επεκτάσεις», αυτό πρόκειται να αλλάξει καθώς προτείνεται να προστεθεί νέα δυνατότητα στο μέλλον και να υποστηρίξει περισσότερες από μία γλώσσες. Μετά την προβολή του ιστότοπου, ένας χρήστης μπορεί να μεταβεί στη σελίδα σύνδεσης αλλά και να δει τα προϊόντα στη σελίδα All Products.

Σε αυτή τη σελίδα ο χρήστης μπορεί να βρει οποιοδήποτε είδος επιθυμεί, εφόσον έχει προηγουμένως προστεθεί από λογαριασμό διαχειριστή, όπως φαίνεται παρακάτω. Ο χρήστης μπορεί να εμφανίσει προϊόντα από τη λίστα αλλά και να προσθέσει στο καλάθι του προϊόντα και να τα δει μετέπειτα στη σελίδα Your Cart.



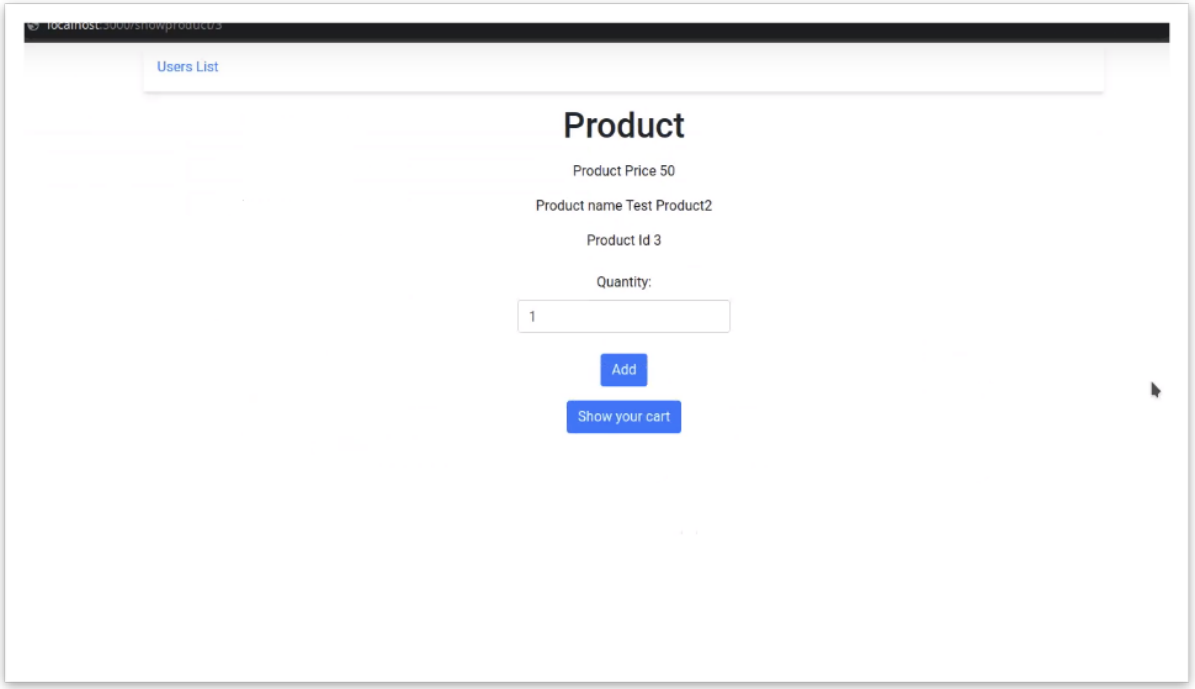
Εικόνα 6.2 εμφάνιση σελίδας με όλα τα προϊόντα

Στο πρώτο σενάριο λοιπόν, αφού ο χρήστης συνδεθεί επιτυχώς ως κανονικός απλός χρήστης, μπορεί να προβάλει τα προϊόντα και να τα προσθέσει στο καλάθι**.** Συνολικά, ο χρήστης μπορεί να εκτελέσει τις εξής ενέργειες:

- Προβολή διαθέσιμων προϊόντων.
- Προβολή πληροφοριών προϊόντων.
- Προσθήκη προϊόντων στο καλάθι

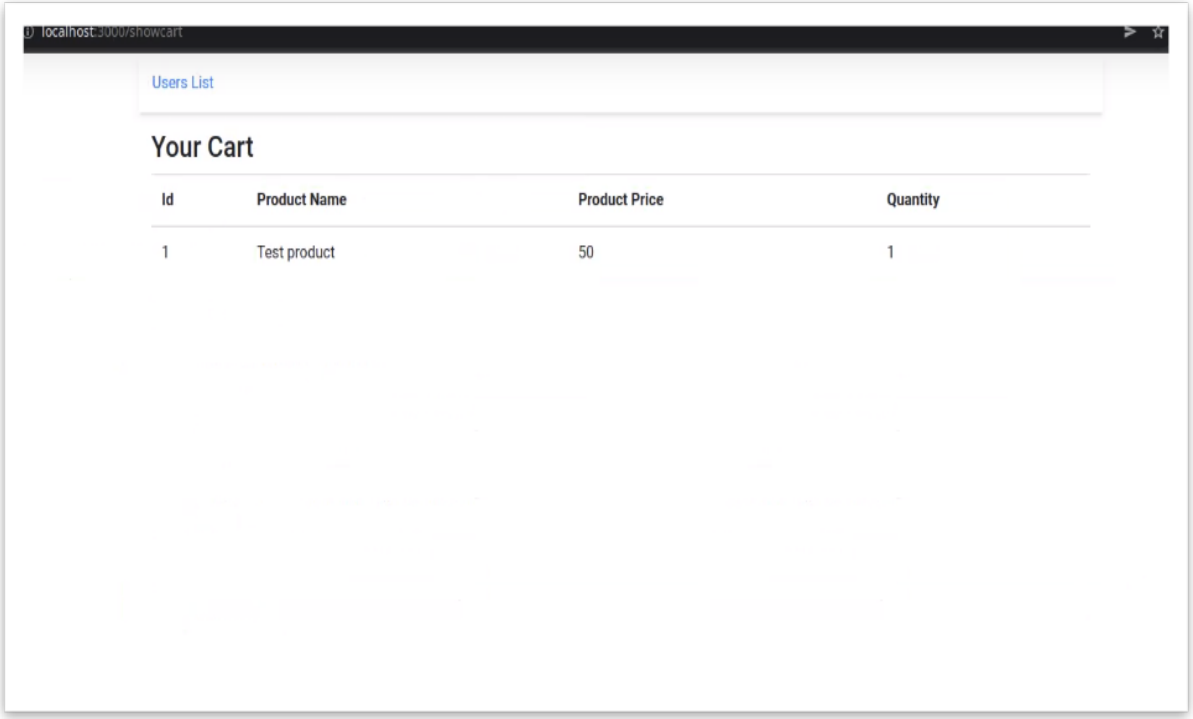
Στις παρακάτω εικόνες εμφανίζεται η σελίδα προβολής των στοιχείων ενός προϊόντος, στην οποία παρέχονται μέσω των αντίστοιχων κουμπιών και οι δυνατότητες προσθήκης στο καλάθι και προβολής του καλαθιού όπως επίσης και η σελίδα προβολής του καλαθιού Cart.

6.2 Λεπτομέρειες προϊόντος



Εικόνα 6.3 λεπτομέρειες του προϊόντος και προσθήκης

6.3 Παραγγελία του χρήστη



Εικόνα 6.4 εμφάνιση παραγγελίας του χρήστη

7 Εγχειρίδιο Διαχειριστή

7.1 Απαιτήσεις συστήματος

Όπως αναφέρθηκε ένας δεύτερος ρόλος χρήστη είναι αυτός του διαχειριστή. Ο διαχειριστής κάνει την εγκατάσταση του IntelliJ idea στο λειτουργικό του υπολογιστή του. Απαιτήσεις συστήματος :

Requirement	**Minimum**	**Recommended**
RAM	2 GB of free RAM	8 GB of total system RAM
CPU	Any modern CPU	Multi-core CPU. IntelliJ IDEA supports multithreading for different operations and processes making it faster the more CPU cores it can use.
Disk space	2.5 GB and another 1 GB for caches	SSD drive with at least 5 GB of free space
Monitor resolution	1024×768	1920×1080
Operating system	Officially released 64-bit versions of the following: • Microsoft Windows 8 or later • macOS 10.14 or later • Any Linux distribution that supports Gnome, KDE , or Unity DE.	Latest 64-bit version of Windows, macOS, or Linux (for example, Debian, Ubuntu, or RHEL)

	Pre-release versions are not supported.	

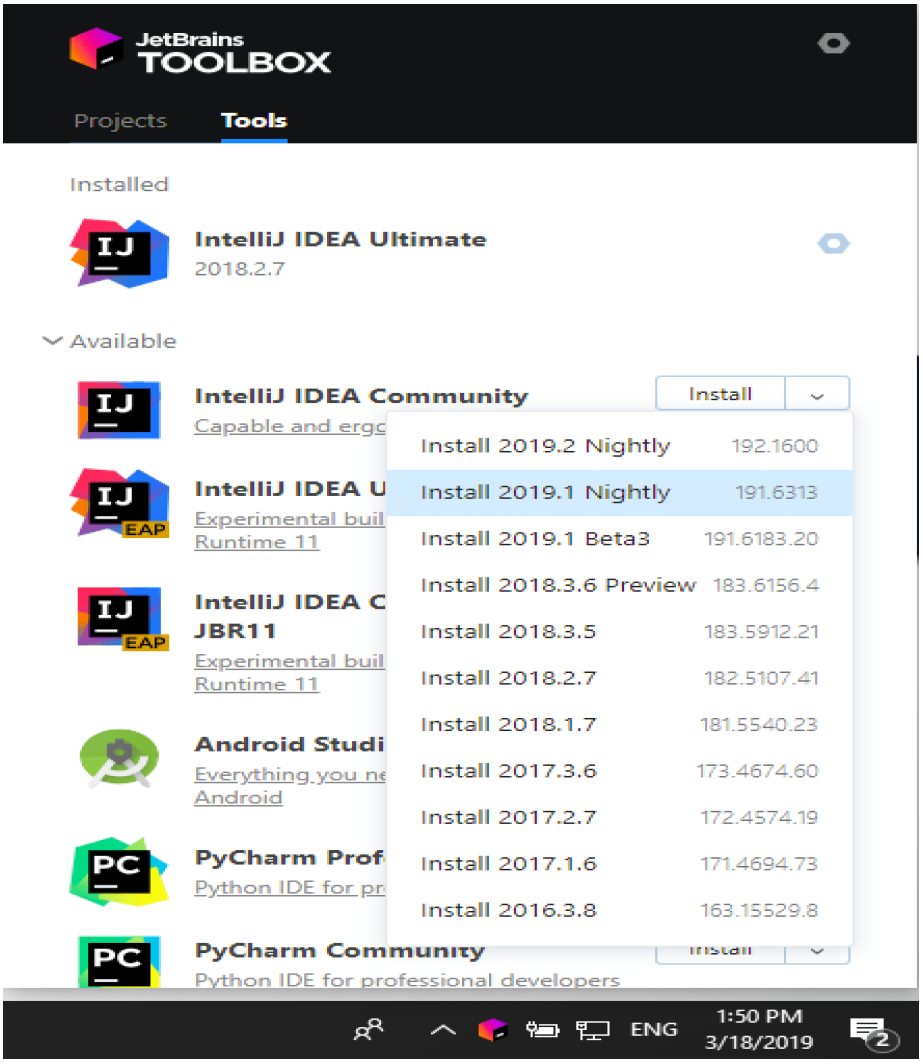
Πίνακας 7.1 προδιαγραφές υπολογιστή για την εγκατάσταση του intelliJ

7.2 Εγκατάσταση Intellij IDEA με το JetBrain

Δεν είναι απαραίτητο να εγκατασταθεί η Java για να τρέξει το IntelliJ IDEA γιατί συνδυάζεται με το ενσωματωμένο πρόγραμμα ανάπτυξης λογισμικού (IDE) που βασίζεται στο (JRE 11). Ωστόσο για να αναπτύξει Java εφαρμογές πρέπει να είναι εγκατεστημένο το Java development kit (JDK).
Προτείνεται να εγκατασταθεί το JetBrain toolbox App για να εγκαταστήσει ο χρήστης διάφορα προγράμματα ανάπτυξης λογισμικού και σε διάφορες εκδόσεις του ίδιου του προγράμματος.
Η εγκατάσταση για το λειτουργικό Windows είναι η εξής :

1. Λήψη του αρχείου εγκατάστασης .exe από το site του Toolbox App Web Page
2. Τρέχουμε το αρχείο εγκατάστασης και ακολουθούμε τα βήματα.

Μετά αφού τρέχουμε το Tollbox App, κάνουμε κλικ στα εικονίδια στην περιοχή των ειδοποιήσεων και διαλέγουμε ποιο πρόγραμμα και ποια έκδοση θέλουμε να εγκαταστήσουμε.



Εικόνα 7.1 επιλογή έκδοσης IntelliJ IDEA για εγκατάσταση

Είσοδος με τον λογαριασμό του στο JetBrain και αμέσως ενεργοποιούνται όλες οι διαθέσιμες άδειες για κάθε περιβάλλον ανάπτυξης λογισμικού (IDE).

7.3 Εγκατάσταση Intellij IDEA standalone

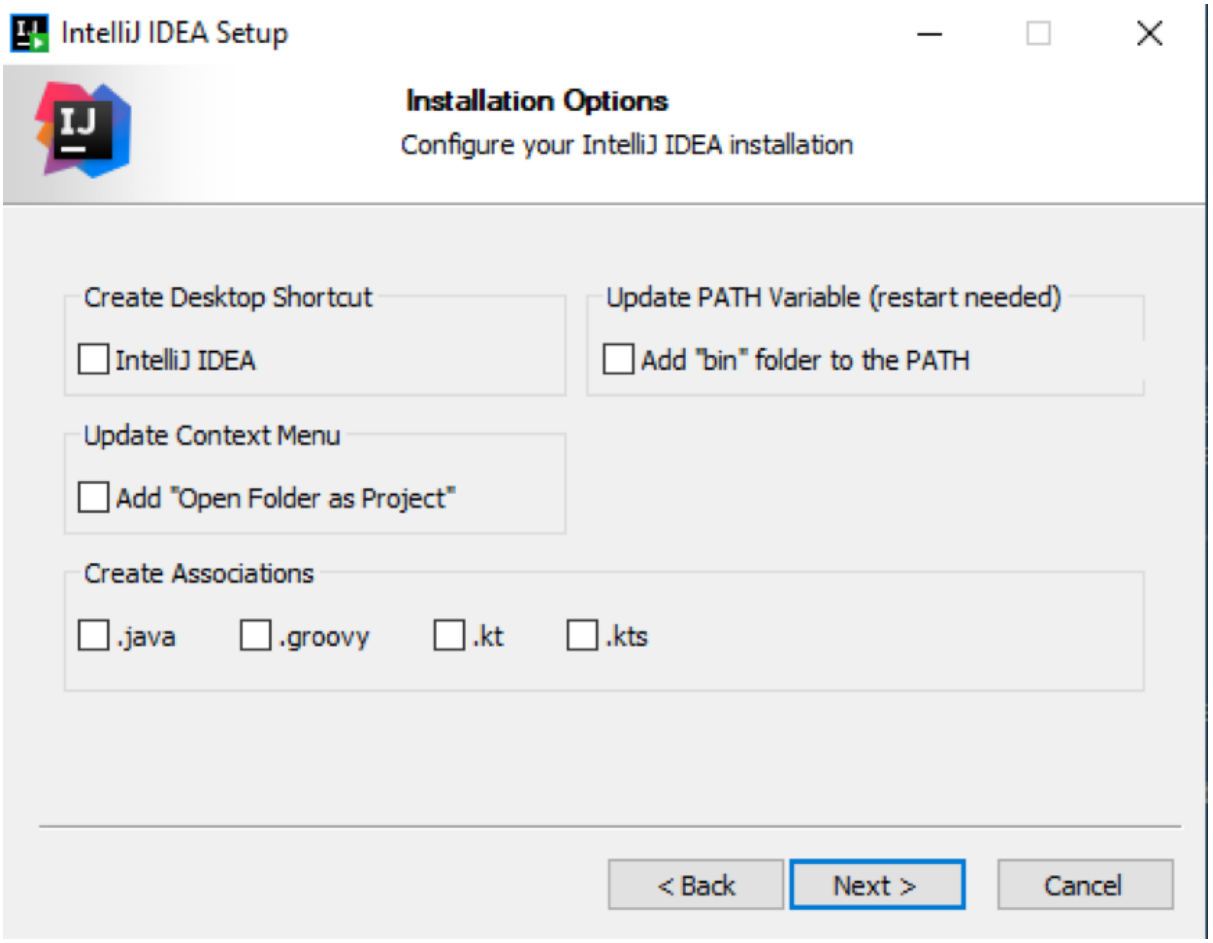
Η Εγκατάσταση του Intellij IDEA μπορεί να γίνει με εύκολο τρόπο, χειροκίνητα.

1. Λήψη του IntelliJ IDEA .exe
2. Τρέξιμο του προγράμματος εγκατάστασης και ακολουθία των βημάτων.

Στα βήματα των επιλογών εγκατάστασης μπορεί να γίνει διαμόρφωση των ακόλουθων :

- Δημιουργία συντόμευσης στην επιφάνεια εργασίας

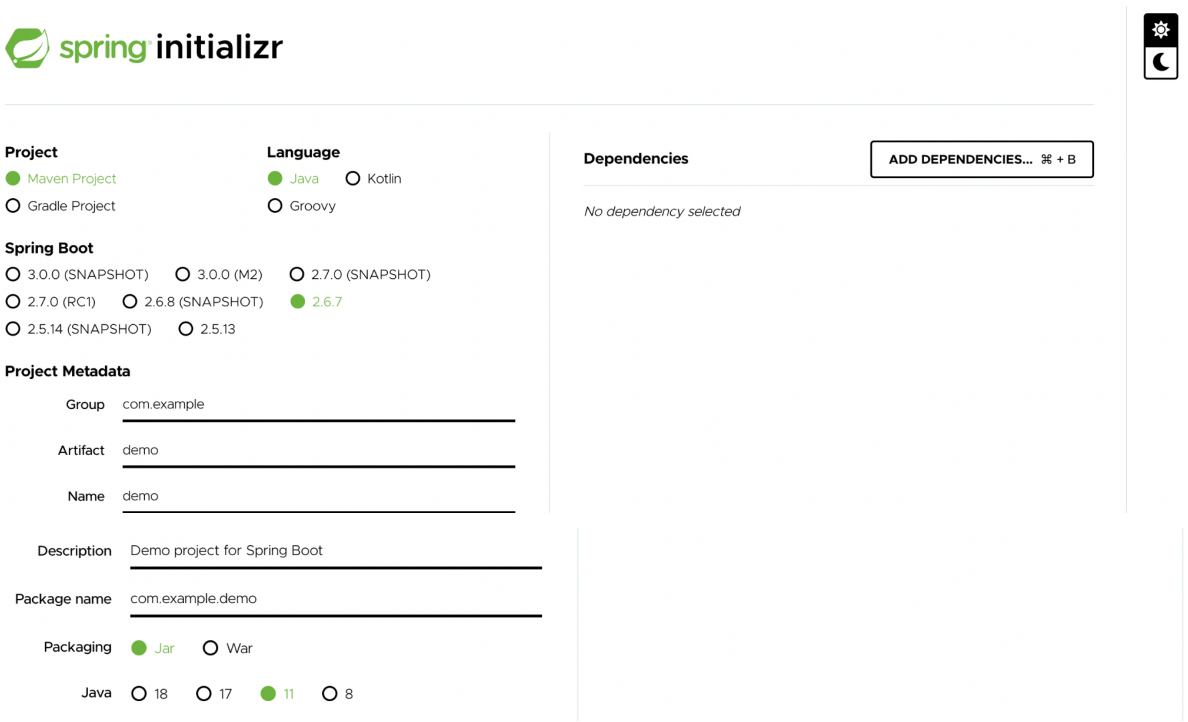
- Προσθήκη φακέλου με το Intellij IDEA στην μεταβλητή περιβάλλοντος για να μπορεί ο χρήστης να τρέχει το πρόγραμμα από κάθε κατάλογο στον οποίο εργάζεται.
- Πατώντας το δεξί κλικ εμφανίζεται το πτυσσόμενο μενού και κάνουμε κλικ την επιλογή άνοιγμα ως έργο (open as folder)



Εικόνα 7.2 εγκατάσταση IntelliJ IDEA για windows λειτουργικό

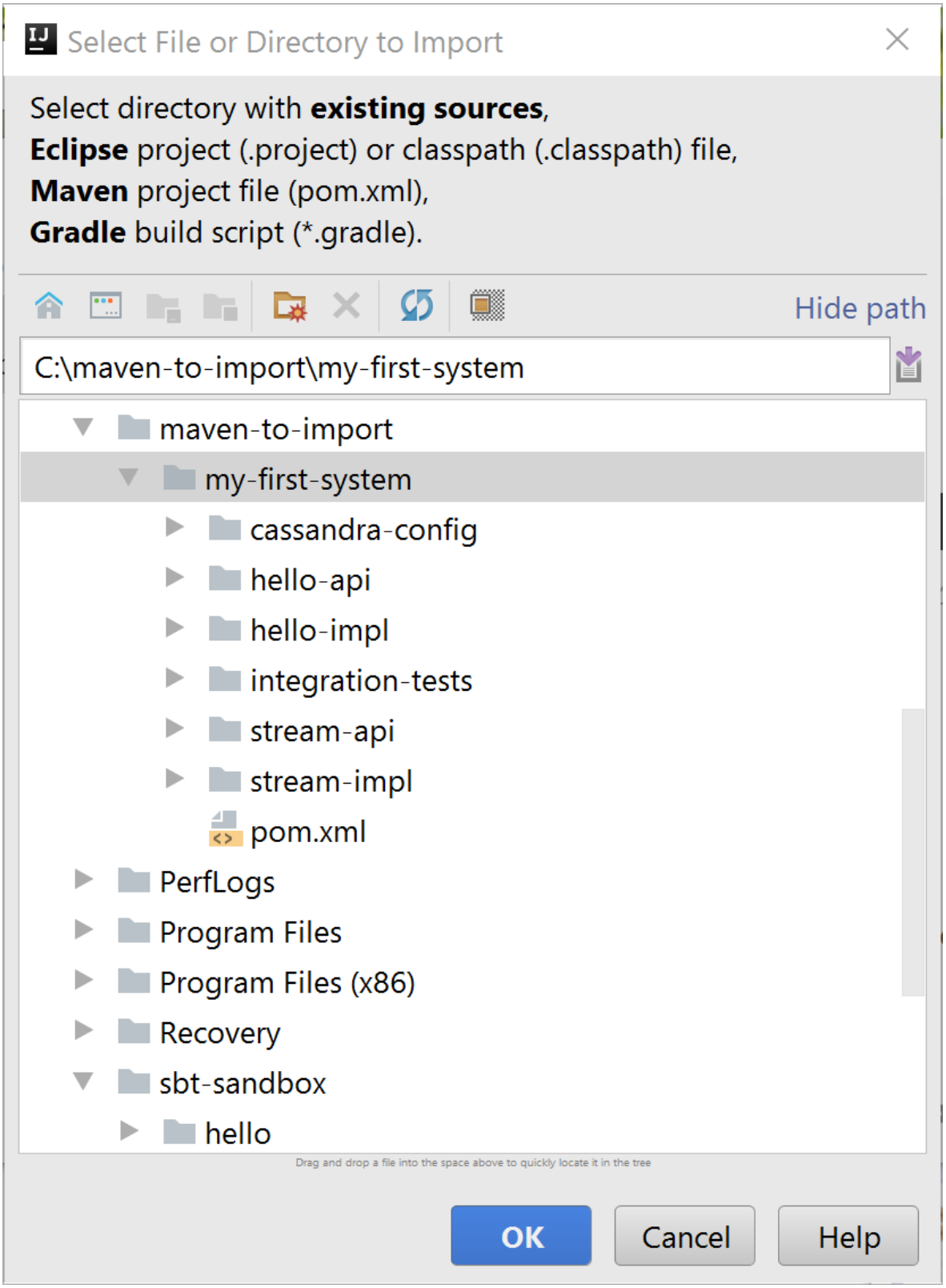
7.4 Εισαγωγή Έργου

Για τη δημιουργία έργου από το site https://start.spring.io/ ο χρήστης - διαχειριστής διαλέγει τον τύπο του έργου δηλαδή αν είναι Maven ή Gradle. Η Γλώσσα προγραμματισμού στη συγκεκριμένη περίπτωση είναι η Java, ο χρήστης επιλέγει την έκδοση της Spring boot όπως επίσης τα dependencies στην δεξιά πλευρά της εικόνας και τα στοιχεία του έργου.

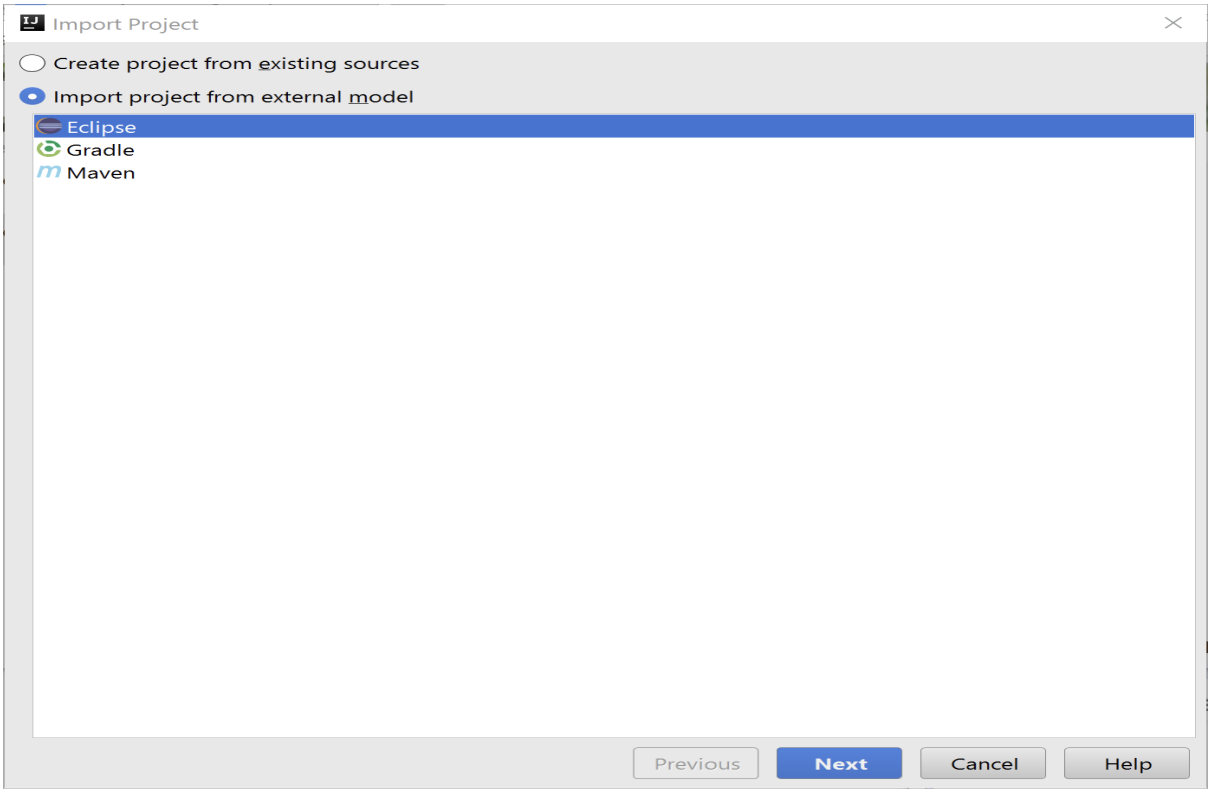


Εικόνα 7.3 δημιουργίας έργου RxJava spring boot

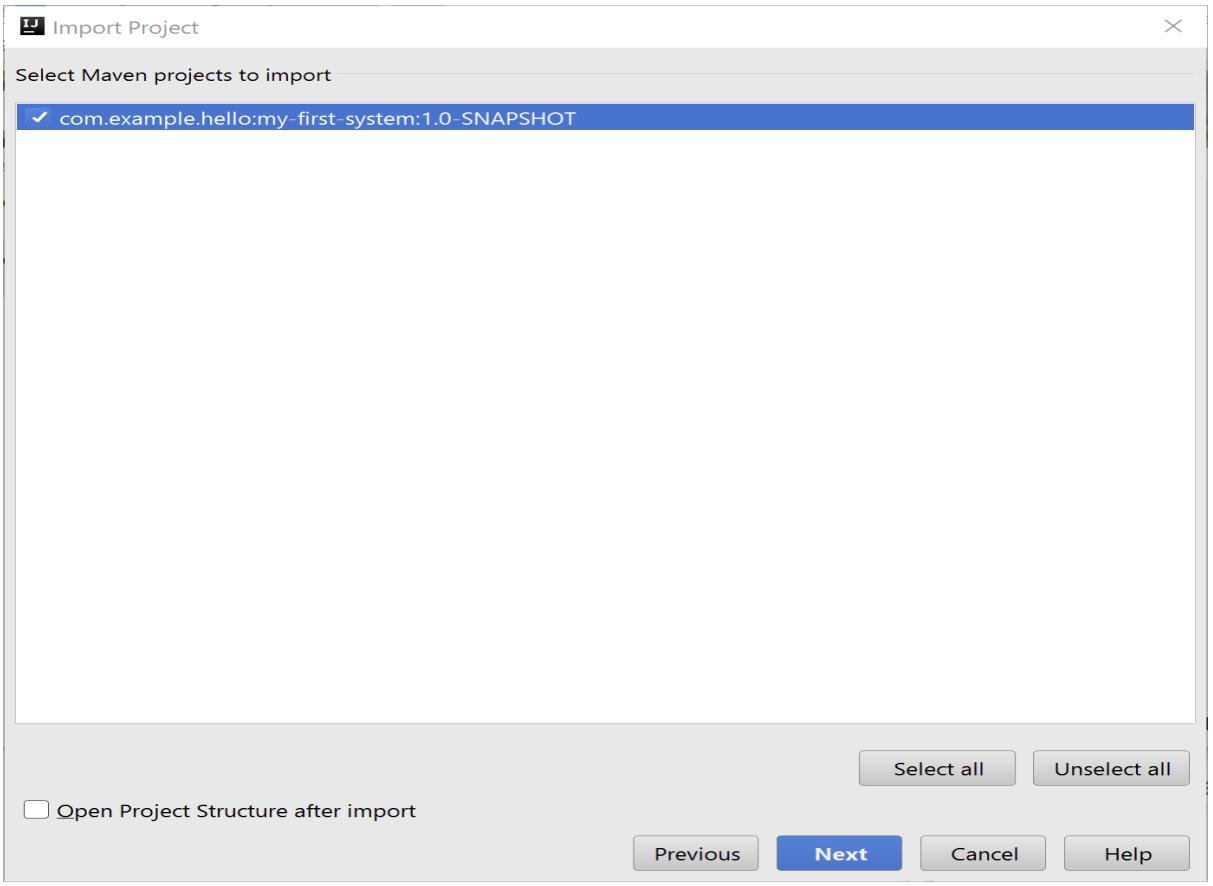
Εγκατάσταση της Mysql και εισαγωγή της βάσης δεδομένων από το αρχείο .sql. Εισαγωγή του έργου από το περιβάλλον του Intellij IDEA. Άνοιγμα του Intellij IDEA. Από την οθόνη καλωσορίσματος (welcome) πατάμε εισαγωγή έργου (import project), το επιλογή αρχείου και φακέλου (Select File or Directory to Import) παράθυρο ανοίγει. Πατάμε OK . Εμφανίζεται το παράθυρο import project. Από την επιλογή Import project from external model επιλέγουμε το Maven και πατάμε επόμενο (NEXT) . Επιλέγουμε το έργο που θέλουμε και αφήνουμε τα άλλα πεδία στις προκαθορισμένες τιμές. Πατάμε επόμενο (NEXT) .



Εικόνα 7.4 επιλογή έργου



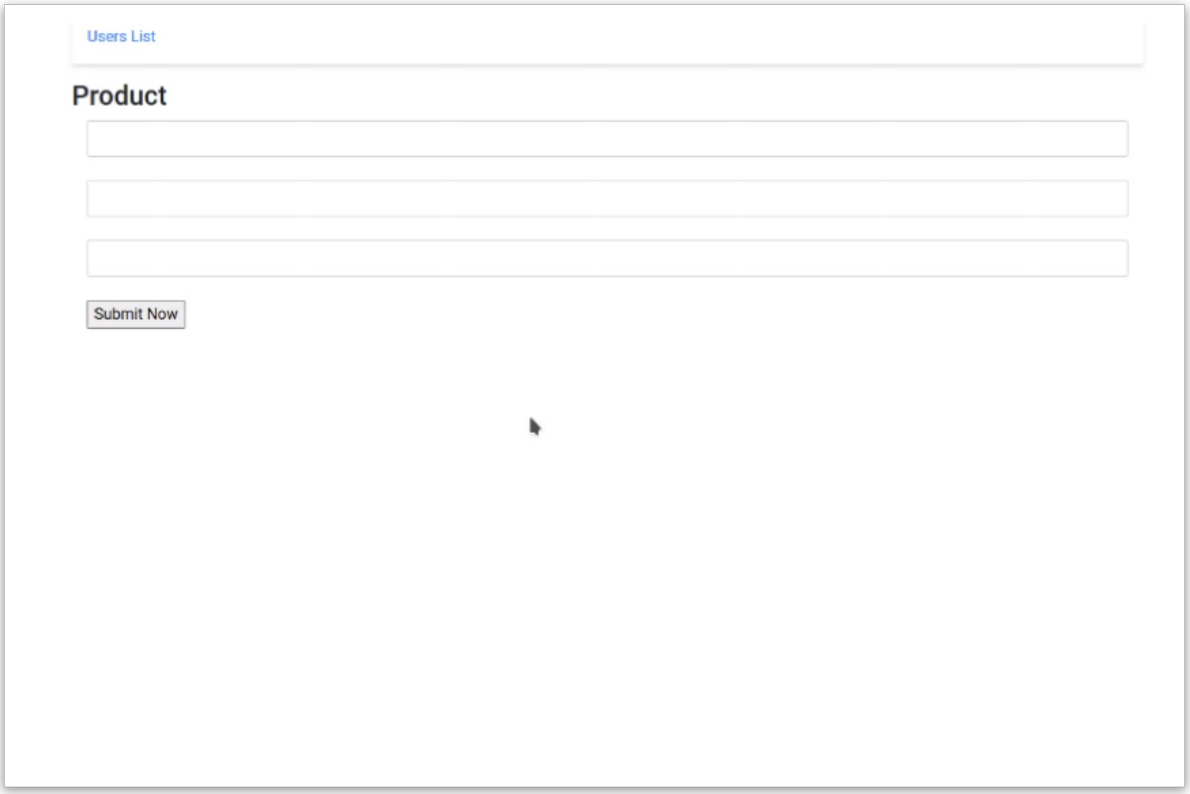
Εικόνα 7.5 επιλογή έργου από το maven



Εικόνα 7.6 τέλος επιλογής έργου

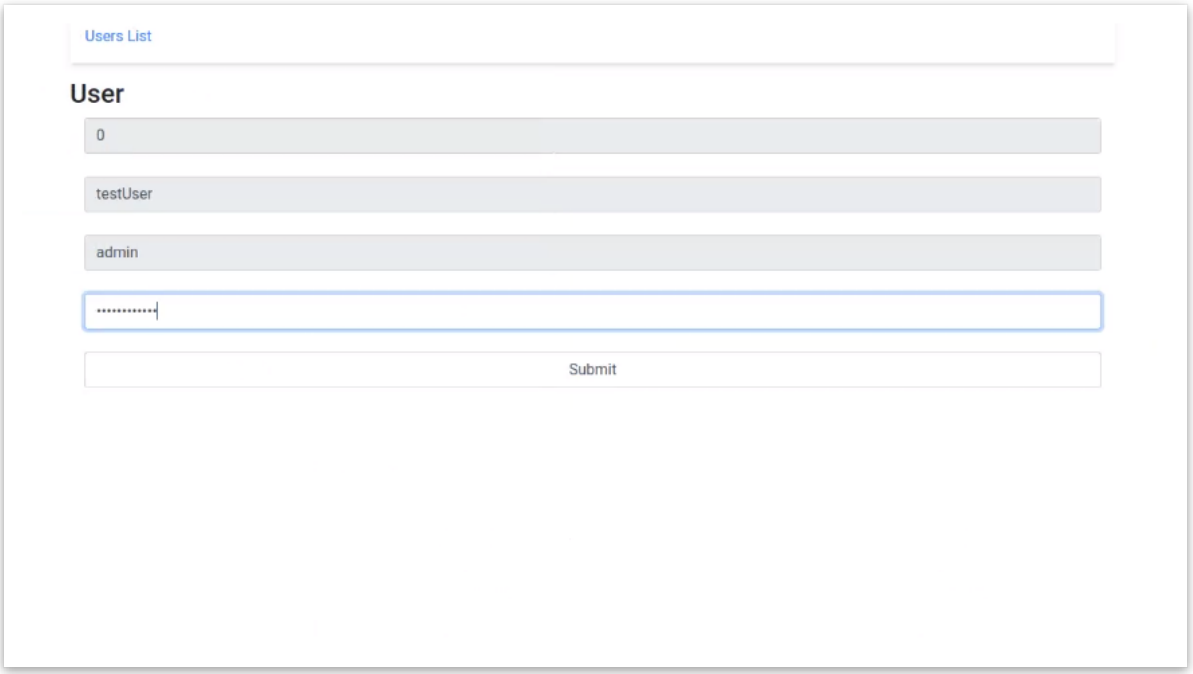
7.5 Εισαγωγή προϊόντων

Εφόσον έχει γίνει με επιτυχία η εισαγωγή του έργου και έχει γίνει build και run από τον διαχειριστή, στην εφαρμογή υπάρχουν οι σελίδες (pages) που ανήκουν στον διαχειριστή όπως η προσθήκη προϊόντων όπως αναπαρίσταται παρακάτω.
Εισαγωγή προϊόντων



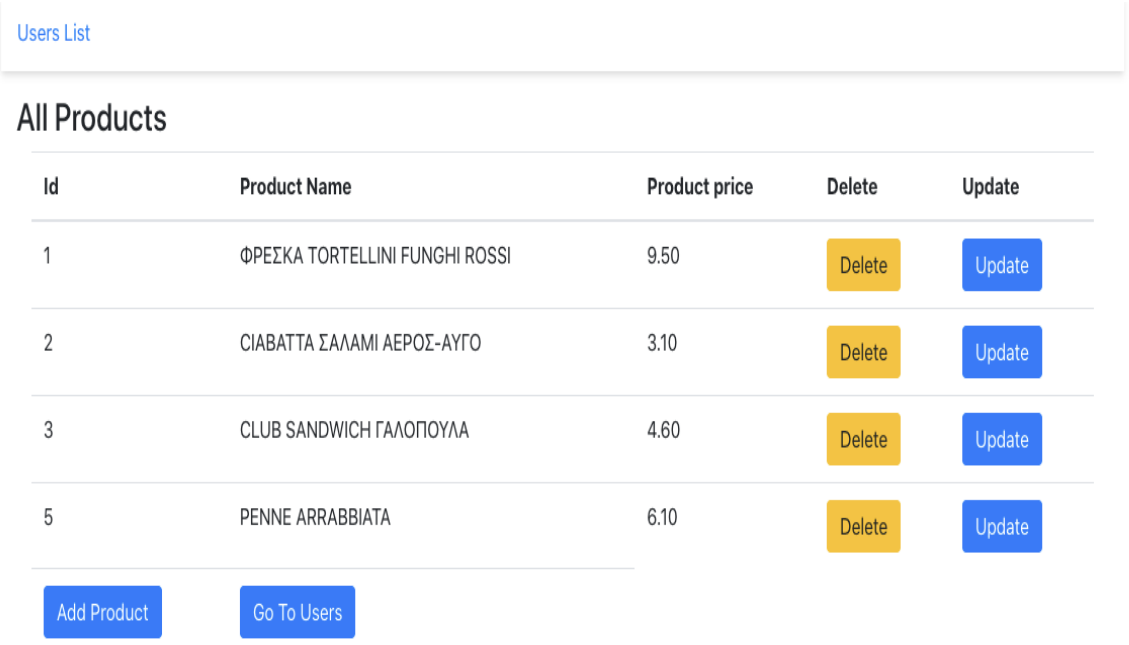
Εικόνα 7.7 εισαγωγή προϊόντος

7.6 Εισαγωγή χρήστη



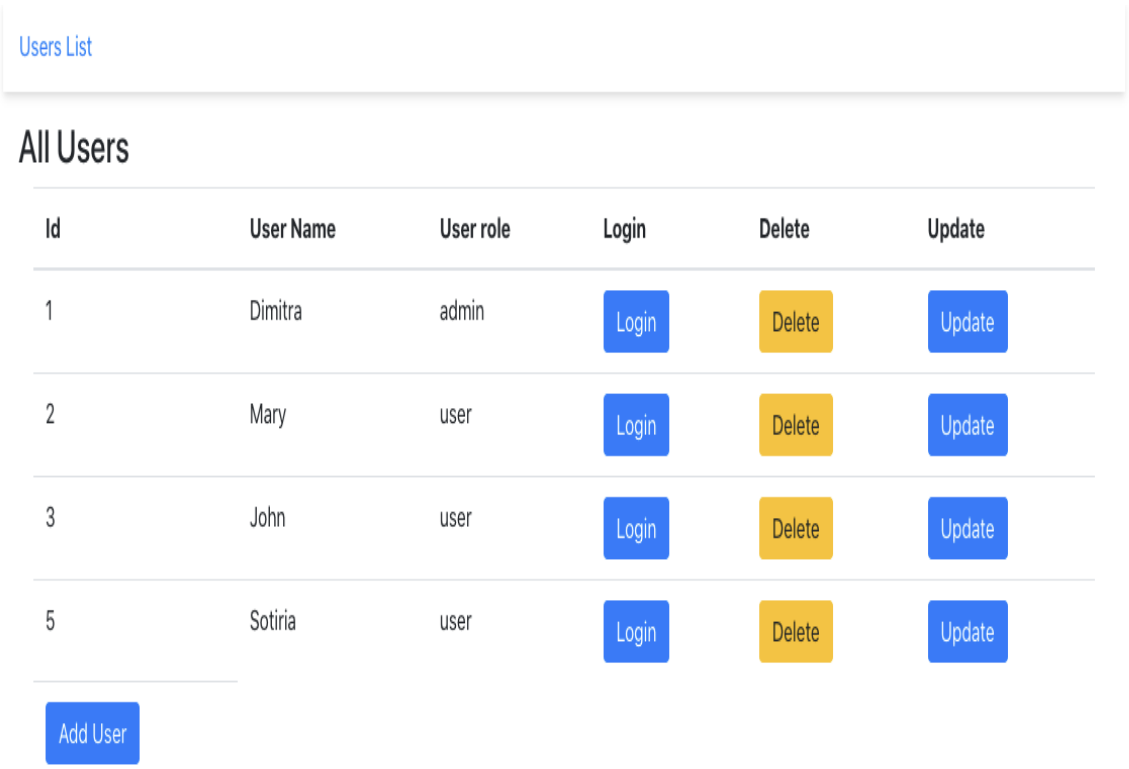
Εικόνα 7.8 εισαγωγή χρήστη

7.7 Ενημέρωση και διαγραφή προϊόντων



Εικόνα 7.9 εμφάνιση όλων των προϊόντων - διαγραφή - ενημέρωση

7.8 Ενημέρωση και διαγραφή χρηστών



Εικόνα 7.10 διαγραφή - ενημέρωση χρήστη

8 Υλοποίηση - Απόσπασμα Κώδικα RxJava Spring Boot

Σε αυτή την ενότητα θα εξηγήσουμε τη λειτουργικότητα του κώδικα από την πλευρά του προγραμματιστή. Η γλώσσα που έχει χρησιμοποιηθεί είναι η Java spring boot και πιο συγκεκριμένα RXJava με WebFlux και r2dbc. Τα σχεδιαστικά πρότυπα του Reactive προγραμματισμού έχουν αναπτυχθεί ραγδαία λόγω της επεκτασιμότητας, της ελαστικότητας και της ανταπόκρισης που παρέχουν. Συγκεκριμένα για να αποθηκεύσουμε τα δεδομένα χρησιμοποιούμε την r2dbc μαζί με mysql driver για να φτιάξουμε ένα πρόγραμμα που δουλεύει ασύγχρονα. Οι παλιές πρακτικές της σχεσιακής βάσης δεδομένων επεξεργάζονταν μεγάλο όγκο δεδομένων με ερωτήματα που εκτελούνται με σύγχρονο τρόπο. Με την εγκατασταση της r2dbc οι βάσεις δεδομένων μπορούν να αναπτύξουν και να χτίσουν ένα ασύγχρονο (non-blocking) δίκτυο κώδικα, παρέχοντας όλα τα προσδοκώμενα οφέλη του reactive προγραμματισμού. Η r2dbc από το Δεκέμβριο του 2020 υποστηρίζει τις ακόλουθες βάσεις δεδομένων :

- H2
- Microsoft SQL Server
- PostgreSQL
- MySQL Driver (r2dbc-mysql)
- Google Cloud Spanner
- MariaDB
- SAP Hana
- Oracle
- DB2

Στο Maven που είναι το αρχείο εγκατάστασης τοποθετούμε τα dependencies (εξαρτήσεις) για να εγκαταστήσουμε την βάση δεδομένων r2dbc για τον reactiveX και για τον driver της mySQL όπως φαίνεται παρακάτω:

```
<dependency>
        <groupId>  org.springframework.boot   </groupId>
          <artifactId>  spring-boot-starter-data-r2dbc   <artifactId>
  </dependency>

<dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
        <groupId>dev.miku</groupId>
        <artifactId>r2dbc-mysql</artifactId>
        <version>0.8.2.RELEASE</version>
 </dependency>
```

Απόσπασμα κώδικα 8.1 εγκατάσταση στο maven το R2DBC, Mysql

Δημιουργούμε το reactive crudrepository για να κάνουμε ερωτήματα στη βάση δεδομένων. Όπως το να ανακτούμε δεδομένα από εγγραφές της βάσης, τη διαγραφή εγγραφών, την ανανέωση των εγγραφών π χ όταν αλλάζει ένα πεδίο στη βάση και τη δημιουργία καινούργιων εγγραφών και όλα αυτά με βάση κάποια συγκεκριμένη τιμή ενός πεδίου συνήθως του ID. Μπορούμε να φτιάξουμε και δικά μας συγκεκριμένα ερωτήματα, για παράδειγμα να δίνει ο χρήστης το όνομα ενός προϊόντος και να εμφανίζονται οι λεπτομέρειες του. Παρακάτω βλέπουμε τα repositories για τον χρήστη, τα προϊόντα και το καλάθι.

```
@Repository
public interface UserRepository extends ReactiveCrudRepository<User, Long> {
  @Query("select * from users where id =:id and password =:password")
  Mono<User> getUserByIdPassword(Long id, String password);
}
```

Απόσπασμα κώδικα 8.2 δημιουργία repository χρήστη, custom ερώτημα

Το παραπάνω ερώτημα έχει ως παραμέτρους το id και το password του χρήστη για την ανάκτηση των στοιχείων ενός συγκεκριμένου χρήστη. Το ερώτημα εκτελείται τη στιγμή της εισαγωγής του στο σύστημα.
Άλλα ερωτήματα υπάρχουν ήδη στη κλάση ReactiveCrudRepository όπως userRepository.save(user), userRepository.findById(id), userRepository.delete(user).
Η κλάση ReactiveCrudRepository έχει υλοποιηθεί για το framework reactor, μοιάζει πολύ με την παραδοσιακή JPARepository κλάση με τη διαφορά ότι όλες οι μέθοδοι που κληρονομεί προέρχονται από το framework reactor όπως web flux, Mono, flux. Υπάρχουν και άλλες κλάσεις για τη δημιουργία βάσης δεδομένων, για παράδειγμα η RxJava2CrudRepository που κληρονομεί τις μεθόδους Observable και Single.

```
@Repository
public interface ProductRepository extends ReactiveCrudRepository<Product,
Long> {
  @Query("select * from products where itemName =:itemName")
  Mono<Product> findByItemName(String itemName);
```

```
}

@Repository
public interface CartRepository extends ReactiveCrudRepository<Cart, Long> {
  @Query("select * from cart where user_id =:userId")
  Flux<Cart> findCartById(Long userId);
}
```

Απόσπασμα κώδικα 8.3 δημιουργία repository προϊόντων και παραγγελιών

Τέλος έχει δημιουργηθεί και το repository (αποθετήριο) για το καλάθι (cart) των προϊόντων που διαλέγει ο χρήστης. Η μέθοδος χρησιμοποιείται για την ανάκτηση όλων των προϊόντων του χρήστη για αυτό υπάρχει και η μέθοδος Flux που φέρνει πολλά δεδομένα μαζί ασύγχρονα. Ο παραπάνω κώδικας αναφέρεται στο repository για την κλάση των προϊόντων. Το ερώτημα αναφέρεται στην ανάκτηση προϊόντος με βάση το όνομα.

Ο ενδιάμεσος κώδικας οποίος είναι η κλάση service διαχειρίζεται τα ερωτήματα των repositories προς τη βάση και τους controllers (ελεγκτές). Τα δεδομένα φτάνουν στον τελικό χρήστη μέσα από το UI (διεπαφή χρήστη). Η κλάση service δηλώνεται ως annotation service. Να πούμε εδώ ότι τα annotations (σχολιασμοί) αποτελούν μια μορφή μεταδεδομένων που παρέχουν πληροφορίες σχετικά με το πρόγραμμα. Χρησιμοποιούνται για να παρέχουν συμπληρωματικές πληροφορίες σχετικά με το πρόγραμμα και δεν έχουν άμεση επίδραση στη λειτουργικότητα του κώδικα και δεν επιφέρουν καμία αλλαγή στη μεταγλώττιση (compile) του προγράμματος. Σχετικά με το service annotation, χρησιμοποιείται μόνο σε κλάσεις για να δείξει ότι οι κλάσεις παρέχουν κάποιες λειτουργικότητες στο πρόγραμμα. Ο κώδικας παρακάτω περιέχει μεθόδους λειτουργικότητας του προγράμματος παραγγελιοληψίας. Η Δημιουργία χρήστη που δέχεται μια παράμετρο, ένα json (Request Body) αλφαριθμητικό (String) με δεδομένα όπως τα προσωπικά του στοιχεια, επιστρέφει ολοκληρωμένη την εγγραφή του νέου χρήστη Mono<ResponseEntity<User>> ασύγχρονα επειδή έχει την ιδιότητα Mono. Η κλήση αυτή εκτελείται με την μέθοδο post και μαρκάρεται στην Java με @PostMapping. Το ίδιο ισχύει για την ενημέρωση του συγκεκριμένου χρήστη δηλαδή παίρνει την ίδια παράμετρο με τις αλλαγές που έχουν γίνει για έναν συγκεκριμένο χρήστη. Η μέθοδος ονομάζεται @PutMapping και επιστρέφει τον χρήστη αλλαγμένο ResponseEntity<User> ασύγχρονα. Η μέθοδος delete @DeleteMapping παίρνει ως όρισμα το ID του χρήστη που είναι μεταβλητή η οποία μπαίνει πάνω στο σύνδεσμο (url) (@PathVariable) και διαγράφεται χωρίς να επιστρέφει κάτι. Άλλες δύο μέθοδοι που αναφέρονται στους χρήστες είναι η ανάκτηση ενός χρήστη με την μέθοδο @GetMapping και το ID του με την ιδιότητα Mono ως απάντηση (ResponseEntity) Mono<ResponseEntity<User>> και την Flux<User> listUsers() που επιστρέφει όλους τους χρήστες χρησιμοποιώντας την ιδιότητα Flux που φέρνει ασύγχρονα όλους τους χρήστες - εγγραφές από την βάση. Η ίδια διαδικασία ισχύει και για τις άλλες Οντότητες (Entities) Cart και Product.

```
@Service
@Slf4j
public class RestApiServices {

    private final Path basePath = Paths.get("./src/main/resources/upload/");

    @Autowired
    private final ProductRepository productRepository;
    @Autowired
    private final UserRepository userRepository;
    @Autowired
    private final CartRepository cartRepository;

    public RestApiServices(ProductRepository productRepository, UserRepository
userRepository, CartRepository cartRepository){
        this.productRepository = productRepository;
        this.userRepository = userRepository;
        this.cartRepository = cartRepository;
    }

    public Mono<Product> newProduct(Product product){
        return productRepository.save(product);
    }

    public Flux<Product> getAll(){
        return productRepository.findAll();
    }
    public Mono<Product> getProduct(long id) { return
productRepository.findById(id); }

    public Mono<User> getUser(long id) { return userRepository.findById(id); }

    public Mono<User> newUser(User user){
        return userRepository.save(user);
    }

    public Mono<Product> updateProduct(long id,Product productMono) {
        return this.productRepository.findById(id)
                .flatMap(product -> {
                    product.setItemName(productMono.getItemName());
                    product.setProductPrice(productMono.getProductPrice());
                    return productRepository.save(product);
```

```
        });
}

public Mono<User> updateUser(long id,User user) {
    return this.userRepository.findById(id)
            .flatMap(user1 -> {
                user1.setName(user.getName());
                user1.setRole(user.getRole());
                return userRepository.save(user1);
            });
}

public Mono<Void> deleteProduct(final long id){
    return this.productRepository.findById(id)
            .flatMap(product ->
                    this.productRepository.delete(product));
}

public Flux<User> getAllUser(){
    return userRepository.findAll();
}

public Mono<User> getUserByIdPassword(long id, String password){
    return this.userRepository.getUserByIdPassword(id,password);
}

public Mono<Void> deleteUser(final long id){
    return this.userRepository.findById(id)
            .flatMap(user ->
                    this.userRepository.delete(user));
}
public Mono<Cart> newCart(Cart cart){
    return cartRepository.save(cart);
}

public Flux<Cart> getCartById(Long id){
    return cartRepository.findCartById(id);
}

public Mono<Cart> updateCart(long id,Cart cart) {
    return this.cartRepository.findById(id)
            .flatMap(cart1 -> {
                cart1.setProductId(cart.getProductId());
```

```
                cart1.setProductName(cart.getProductName());
                cart1.setProductPrice(cart.getProductPrice());
                cart1.setUserId(cart.getUserId());
                return cartRepository.save(cart1);
            });
    }

    public Mono<Void> deleteCart(final long id){
        return this.cartRepository.findById(id)
                .flatMap(cart ->
                        this.cartRepository.delete(cart));
    }
}
```

Απόσπασμα κώδικα 8.4 δημιουργία κλάσης Service

Στον παραπάνω κώδικα φαίνεται με ποιον τρόπο χρησιμοποιεί τις μεθόδους των repositories για όλη την λειτουργία του CRUD συστήματος για κάθε μοντέλο (model, POJO).

Το επόμενο και επίσης σημαντικό κομμάτι του έργου είναι οι Controllers με το annotation @RestController που σηματοδοτεί ότι η κλάση αυτή θα χρησιμοποιηθεί για τη δημιουργία RestApi από τους client όπως για παράδειγμα από το front end (ReactJs) για το δικό μας πρόγραμμα ή από τη iOS (swift) και android (Java). Ο κώδικας λαμβάνει γεγονότα (events) από τους χρήστες μέσω του UI και χρησιμοποιεί τις λειτουργίες του service όπως είδαμε παραπάνω για να φτάσει τελικά στην βάση. Γίνεται χρήση των μεθόδων του reactor για την ασύγχρονη εκπομπή (streaming) των δεδομένων όπως το Mono και το Flux.

```
@CrossOrigin(origins ={"http://localhost:3000"})
@RestController
@RequestMapping("/api")
public class UploadController {

    private final Path basePath = Paths.get("./src/main/resources/upload/");
    @Autowired
    private final RestApiServices restApiServices;

    //constractor
    public UploadController(RestApiServices restApiServices){
        this.restApiServices = restApiServices;
    }

    //get all products
```

```
    @GetMapping("/products")
    public Flux<Product> listProducts(){
        return restApiServices.getAll();
    }

    //get all products by id of product
    @GetMapping("/products/{id}")
    public Mono<ResponseEntity<Product>> product(@PathVariable Long id) {
        return restApiServices.getProduct(id).
        map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());

    }

    //create new product
    @PostMapping(value = "/create")
    public Mono<Product> addNewProduct(@RequestBody Product product) {
        if (product != null){
            product.toString();
            System.out.println(product.toString());
        }
        return restApiServices.newProduct(product);
    }
    //update product by id
    @PutMapping("/upload/{id}")
    public Mono<ResponseEntity<Product>> updateProduct(@PathVariable Long
id,@RequestBody Product productMono){
        return
this.restApiServices.updateProduct(id,productMono).map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    //delete product by id
    @DeleteMapping("/delete/{id}")
    public Mono<ResponseEntity<Void>> deleteProduct(@PathVariable Long id){
        return this.restApiServices.deleteProduct(id)
                .then(Mono.just(new ResponseEntity<Void>(HttpStatus.OK)))
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    //get user by id
    @GetMapping("/users/{id}")
    public Mono<ResponseEntity<User>> user(@PathVariable Long id) {
```

```
        return restApiServices.getUser(id).
                map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());

    }

    //create user
    @PostMapping(value = "/create/user")
    public Mono<User> addNewUser(@RequestBody User user) {
        if (user != null){
            user.toString();
            System.out.println(user.toString());
        }
        return restApiServices.newUser(user);
    }

    //update user by id
    @PutMapping("/upload/user/{id}")
    public Mono<ResponseEntity<User>> updateUser(@PathVariable Long
id,@RequestBody User user){
        return this.restApiServices.updateUser(id,user).map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    //get all users
    @GetMapping("/users")
    public Flux<User> listUsers(){
        return restApiServices.getAllUser();
    }

    //get user by id and password
    @GetMapping("/users/{id}/{password}")
    public Mono<ResponseEntity<User>> userByPassword(@PathVariable Long id,
@PathVariable String password){
        return restApiServices.getUserByIdPassword(id,password)
                .map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    //delete user by id
    @DeleteMapping("/delete/user/{id}")
    public Mono<ResponseEntity<Void>> deleteUser(@PathVariable Long id){
        return this.restApiServices.deleteUser(id)
```

```
            .then(Mono.just(new ResponseEntity<Void>(HttpStatus.OK)))
            .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    //get cart bu user id
    @GetMapping("/cart/{id}")
    public Flux<Cart> cart(@PathVariable Long id) {
        return restApiServices.getCartById(id);

    }

    //create cart for each user
    @PostMapping(value = "/create/cart")
    public Mono<Cart> addNewCart(@RequestBody Cart cart) {
        if (cart != null){
            cart.toString();
            System.out.println(cart.toString());
        }
        return restApiServices.newCart(cart);
    }

    //update cart by user id
    @PutMapping("/upload/cart/{id}")
    public Mono<ResponseEntity<Cart>> updateCart(@PathVariable Long
id,@RequestBody Cart cart){
        return this.restApiServices.updateCart(id,cart).map(ResponseEntity::ok)
                .defaultIfEmpty(ResponseEntity.notFound().build());
    }

    //delete cart by card id
    @DeleteMapping("/delete/cart/{id}")
    public Mono<ResponseEntity<Void>> deleteCart(@PathVariable Long id){
        return this.restApiServices.deleteCart(id)
                .then(Mono.just(new ResponseEntity<Void>(HttpStatus.OK)))
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
}
```

Απόσπασμα κώδικα 8.5 δημιουργία Restcontroller

Ένας άλλος τρόπος για να δημιουργήσουμε RestApi είναι ο συναρτησιακός προγραμματισμός σε τελικά σημεία (functional endpoints). Το Spring συναρτησιακό πλαίσιο λογισμικού (Spring Functional Web Framework) στο διαδίκτυο δημιουργήθηκε στην Spring WebFlux αλλά είναι διαθέσιμο και στις εφαρμογές Spring

MVC (Model View Controller). Είναι ένα εναλλακτικό μοντέλο προγραμματισμού το οποίο χρησιμοποιεί συναρτήσεις για την δρομολόγηση και τον χειρισμό αιτημάτων (Routing and handling requests).
Η συνάρτηση χειρισμού είναι μια μέθοδος που παίρνει ένα αίτημα από το διακομιστή (Server Request) ως όρισμα. Μπορεί να γραφεί και ως lambda έκφραση (expression). Παρακάτω δίνεται ο κώδικας από την κλάση
WelcomeHandler διαχειρίζεται τις λειτουργίες (operations) CRUD γραμμένα ως συναρτήσεις διαχείρισης (handler functions).

```
@Component
@Slf4j
public class WelcomeHandler {

    @Autowired
    private final RestApiServices restApiServices;

    public WelcomeHandler(RestApiServices restApiServices) {
        this.restApiServices = restApiServices;
    }
    public Mono<ServerResponse> listProducts(ServerRequest serverRequest) {
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(restApiServices.getAll(), Product.class);
    }
    public Mono<ServerResponse> getProduct(ServerRequest serverRequest) {
        Mono<Product> studentMono = restApiServices.getProduct(
                Long.parseLong(serverRequest.pathVariable("id")));
        return studentMono.flatMap(student -> ServerResponse.ok()
                        .body(fromValue(student)))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
    public Mono<ServerResponse> addNewProduct(ServerRequest serverRequest)
{
        Mono<Product> productMono = serverRequest.bodyToMono(Product.class);
        return productMono.flatMap(product ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(restApiServices.newProduct(product), Product.class));
    }

    public Mono<ServerResponse> updateProduct(ServerRequest serverRequest) {
        final long id = Long.parseLong(serverRequest.pathVariable("id"));
        Mono<Product> productMono = serverRequest.bodyToMono(Product.class);
```

```
        return productMono.flatMap(product ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(restApiServices.updateProduct(id,product), Product.class));
    }
    public Mono<ServerResponse> deleteProduct(ServerRequest serverRequest) {
        final long id= Long.parseLong(serverRequest.pathVariable("id"));
        return ServerResponse.noContent().build(restApiServices.deleteProduct(id))
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> getUser(ServerRequest serverRequest) {
        Mono<User> userMono = restApiServices.getUser(
                Long.parseLong(serverRequest.pathVariable("id")));
        return userMono.flatMap(user -> ServerResponse.ok()
                .body(fromValue(user)))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
    public Mono<ServerResponse> getUserByIdPassword(ServerRequest
serverRequest) {
        Mono<User> userMono = restApiServices.getUserByIdPassword(

Long.parseLong(serverRequest.pathVariable("id")),serverRequest.pathVariable("p
assword"));
        return userMono.flatMap(user -> ServerResponse.ok()
                .body(fromValue(user)))
                .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> addNewUser(ServerRequest serverRequest) {
        Mono<User> userMono = serverRequest.bodyToMono(User.class);
        return userMono.flatMap(user ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(restApiServices.newUser(user), User.class));
    }
    public Mono<ServerResponse> updateUser(ServerRequest serverRequest) {
        final long id = Long.parseLong(serverRequest.pathVariable("id"));
        Mono<User> userMono = serverRequest.bodyToMono(User.class);

        return userMono.flatMap(user ->
                ServerResponse.status(HttpStatus.OK)
```

```
                .contentType(MediaType.APPLICATION_JSON)
                .body(restApiServices.updateUser(id,user), User.class));
    }

    public Mono<ServerResponse> listUsers(ServerRequest serverRequest) {
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(restApiServices.getAllUser(), User.class);
    }
    public Mono<ServerResponse> deleteUser(ServerRequest serverRequest) {
        final long id= Long.parseLong(serverRequest.pathVariable("id"));
        return ServerResponse.noContent().build(restApiServices.deleteUser(id))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
    public Mono<ServerResponse> getCart(ServerRequest serverRequest) {
        final String id = serverRequest.pathVariable("id");
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(restApiServices.getCartById(Long.parseLong(id)),Cart.class);
    }

    public Mono<ServerResponse> addNewCart(ServerRequest serverRequest) {
        Mono<Cart> cartMono = serverRequest.bodyToMono(Cart.class);
        return cartMono.flatMap(cart ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(restApiServices.newCart(cart), Cart.class));
    }
    public Mono<ServerResponse> updateCart(ServerRequest serverRequest) {
        final long id = Long.parseLong(serverRequest.pathVariable("id"));
        Mono<Cart> cartMono = serverRequest.bodyToMono(Cart.class);
        return cartMono.flatMap(cart ->
                ServerResponse.status(HttpStatus.OK)
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(restApiServices.updateCart(id,cart), Cart.class));
    }


    public Mono<ServerResponse> deleteCart(ServerRequest serverRequest) {
        final long id= Long.parseLong(serverRequest.pathVariable("id"));
        return ServerResponse.noContent().build(restApiServices.deleteCart(id))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
```

```
}
```

Απόσπασμα κώδικα 8.6 δημιουργία handler

Το επόμενο στάδιο είναι να δημιουργήσουμε τις συναρτήσεις δρομολογητών (router functions). Ο πιο απλός τρόπος είναι να χρησιμοποιήσουμε τη μέθοδο RouterFunctions.route(). Υπάρχουν πολλοί τρόποι να δημιουργήσουμε πολλαπλές συναρτήσεις δρομολογητών μαζί.
Οι routes εγγράφονται ως Spring οντότητες (beans) @Bean και έτσι μπορούν να δημιουργηθούν σε κάθε διαμορφωμένη κλάση @Configuration.

```
@Configuration
public class WelcomeRouter {
   @Bean
   public RouterFunction<ServerResponse> route(WelcomeHandler
welcomeHandler) {
       return RouterFunctions
            .route(POST("/api/create")

.and(accept(APPLICATION_JSON)),welcomeHandler::addNewProduct)
            .andRoute(GET("/api/products/{id:[0-9]+}")
                    .and(accept(APPLICATION_JSON)),
welcomeHandler::getProduct)
            .andRoute(GET("/api/products/")
                .and(accept(APPLICATION_JSON)),
welcomeHandler::listProducts)
            .andRoute(GET("/add/redirect"), req ->
                ServerResponse.temporaryRedirect(URI.create("/product"))
                    .build())

.andRoute(PUT("/api/upload/{id:[0-9]+}").and(accept(APPLICATION_JSON)),welco
meHandler::updateProduct)

.andRoute(DELETE("/api/delete/{id:[0-9]+}").and(accept(APPLICATION_JSON)),w
elcomeHandler::deleteProduct)
            .andRoute(GET("/api/users/{id:[0-9]+}")
                .and(accept(APPLICATION_JSON)), welcomeHandler::getUser)
            .andRoute(GET("/api/users/{id:[0-9]+}/{password}")
                .and(accept(APPLICATION_JSON)),
welcomeHandler::getUserByIdPassword)

.andRoute(PUT("/api/upload/user/{id:[0-9]+}").and(accept(APPLICATION_JSON)),
welcomeHandler::updateUser)
```

```
            .andRoute(POST("/api/create/user")

.and(accept(APPLICATION_JSON)),welcomeHandler::addNewUser)
            .andRoute(GET("/api/users/")
                  .and(accept(APPLICATION_JSON)), welcomeHandler::listUsers)

.andRoute(DELETE("/api/delete/user/{id:[0-9]+}").and(accept(APPLICATION_JSO
N)),welcomeHandler::deleteUser)
            .andRoute(GET("/api/cart/{id:[0-9]+}")
                  .and(accept(APPLICATION_JSON)), welcomeHandler::getCart)

.andRoute(PUT("/api/upload/cart/{id:[0-9]+}").and(accept(APPLICATION_JSON)),
welcomeHandler::updateCart)
            .andRoute(POST("/api/create/cart")

.and(accept(APPLICATION_JSON)),welcomeHandler::addNewCart)

.andRoute(DELETE("/api/delete/cart/{id:[0-9]+}").and(accept(APPLICATION_JSON
)),welcomeHandler::deleteCart);
    }
}
```

Απόσπασμα κώδικα 8.7 δημιουργία functional router

Τέλος μπαίνουμε στο τεστ κώδικα τον οποίο γράψαμε για να δοκιμάσουμε τα WebFlux endpoints. Αυτός ο κώδικας χρησιμοποιεί εσωτερικά την functional router ή έναν controller ή και ακόμα τον πραγματικό διακομιστή (real server). Δημιουργούμε ένα αντικείμενο τύπου WebTestClient και με τις μεθόδους που διαθέτει δοκιμάζουμε τον κώδικα που θα χρησιμοποιηθεί από τους τελικούς χρήστες. Για να εκτελεστεί ένα αίτημα - ερώτημα θα πρέπει να χρησιμοποιηθεί η μέθοδος exchange().
Η μέθοδος exchange() επιστρέφει ένα αντικείμενο (WebTestClient.ResponceSpec) το οποίο έχει χρήσιμες μεθόδους για να επαληθεύσει την απάντηση από τις μεθόδους expectStatus, expectBody και expectHeader. Παρακάτω παρατίθεται ο κώδικας που εκτελεί το test.

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
public class UserRepositoryTest {

    @MockBean
    UserRepository userRepository;
```

```
@MockBean
RestApiServices restApiServices;

@Autowired
private WebTestClient webClient;

@Test
public void addSingleUser(){
    User user = new User();
    user.setId(6L);
    user.setName("Mathew");
    user.setRole("user");

    System.out.println(user.toString());

    Mockito.when(restApiServices.newUser(user)).thenReturn(Mono.just(user));

    webClient.post()
            .uri("/api/create/user")
            .contentType(MediaType.APPLICATION_JSON)
            .body(BodyInserters.fromObject(user))
            .exchange()
            .expectStatus().isOk();

    Mockito.verify(restApiServices, times(1)).newUser(user);
}

@Test
void testGetUsersById()
{
    User user = new User();
    user.setId(1L);
    user.setName("Dimitra");
    user.setRole("admin");

    System.out.println(user.toString());

    Mockito
            .when(restApiServices.getUser(1L))
            .thenReturn(Mono.just(user));

    webClient.get().uri("/api/users/{id}", 1)
            .exchange()
```

```
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.id").isEqualTo(1)
            .jsonPath("$.name").isEqualTo("Dimitra")
            .jsonPath("$.role").isEqualTo("admin");

        Mockito.verify(restApiServices, times(1)).getUser(1L);
    }

    @Test
    public void testDeleteUser(){
        User user = new User();
        user.setId(1L);
        user.setName("Dimitra");
        user.setRole("admin");

        Mono<Void> voidReturn  = Mono.empty();
        Mockito.when(restApiServices.deleteUser(1L))
                .thenReturn(voidReturn);
        webClient.delete().uri("/api/delete/user/{id}", 1)
                .exchange()
                .expectStatus().isNoContent();
    }
}
```

Απόσπασμα κώδικα 8.8 δημιουργία UserRepositoryTest με WebTestClient

Μετά το back end γίνεται λόγος για το front end. Η επικοινωνία των δύο γίνεται με την χρήση των Rest Api για τον User, Product, Cart. Όπως είδαμε παραπάνω το rest api μπορεί να δημιουργηθεί με δύο τρόπους : με controllers και με functional router. Για την επικοινωνία του front end με το back end χρησιμοποιήθηκε ο Controller ως endpoint.

Το front end υλοποιήθηκε με τη γλώσσα προγραμματισμού JavaScript και το framework ReactJS.

Η σύνδεση μεταξύ του front end και backend γίνεται στον παρακάτω κώδικα.

```
// Create services to connect with Rest api
class ProductDataService {

    // get all products
    getAll(){
        return axios.get(`${INSTRUCTOR_API_URL}/products`);
    }
```

```
    // create product json application
    create(data) {
        return axios.post(`${INSTRUCTOR_API_URL}/create`, data, {headers});
    }

    // update product by Id and json application
    update(id,data) {
        return axios.put(`${INSTRUCTOR_API_URL}/upload/${id}`, data, {headers});
    }

    // delete product by Id
    delete(id) {
        return axios.delete(`${INSTRUCTOR_API_URL}/delete/${id}`,{headers});
    }

    // retrieve product by id
    retrive(id) {
        return axios.get(`${INSTRUCTOR_API_URL}/products/${id}`, {headers})
    }

    redirect() {
        return axios.get(`${INSTRUCTOR_API_URL}/add/redirect`, {headers})
    }

    // get all users
    getAllUsers(){
        return axios.get(`${INSTRUCTOR_API_URL}/users`);
    }

    //retrive user by Id
    retrive_user(id) {
        return axios.get(`${INSTRUCTOR_API_URL}/users/${id}`, {headers})
    }

    // retrive user by id and password
    retrive_user2(id,password) {
        return axios.get(`${INSTRUCTOR_API_URL}/users/${id}/${password}`,
{headers})
    }

    // create user with json application
    create_user(data) {
```

```
        return axios.post(`${INSTRUCTOR_API_URL}/create/user`, data, {headers});
    }

    // update user by Id and json application
    update_user(id,data) {
        return axios.put(`${INSTRUCTOR_API_URL}/upload/user/${id}`, data,
{headers});
    }

    // delete user by Id
    delete_user(id) {
        return axios.delete(`${INSTRUCTOR_API_URL}/delete/user/${id}`,{headers});
    }

    // retrieve car by user Id
    retrive_cart(id) {
        return axios.get(`${INSTRUCTOR_API_URL}/cart/${id}`, {headers})
    }

    // create cart by user
    create_cart(data) {
        return axios.post(`${INSTRUCTOR_API_URL}/create/cart`, data, {headers});
    }

    // update cart by user Id
    update_cart(id,data) {
        return axios.put(`${INSTRUCTOR_API_URL}/upload/cart/${id}`, data,
{headers});
    }

    // delete user by Id
    delete_cart(id) {
        return axios.delete(`${INSTRUCTOR_API_URL}/delete/cart/${id}`,{headers});
    }
}
export default new ProductDataService();
```

Απόσπασμα κώδικα 8.9 Reactjs σύνδεση front end με backend

9 Υλοποίηση - Απόσπασμα Κώδικα Java Spring Boot

Σε άλλη περίπτωση φτιάχνουμε τη βάση δεδομένων με την JPA (Java Persistence Api) η οποία δε δουλεύει με ασύγχρονο τρόπο. Το Api αυτό είναι ευρέως διαδεδομένο και χρησιμοποιείται πολύ συχνά για τη δημιουργία web services. Παρακάτω φαίνεται η εγκατάσταση της JPA και mysql στο maven.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
</dependency>
```

Απόσπασμα κώδικα 9.1 εγκατάσταση JPA Mysql στο maven

Αντίστοιχα φτιάχνουμε ένα CRUD (create, read, update, delete) repository JpaRepository με τα default ερωτήματα που έχει η κλάση ή με custom ερωτήματα που φτιάχνουμε. Για το δικό μας το πρότζεκτ θα είναι:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select * from users where id =:id and password =:password")
  User getUserByIdPassword(Long id, String password);
}
```

Απόσπασμα κώδικα 9.2 JPA Repository του χρήστη

Η διαφορά με το reactive crud repository είναι ότι στο JPA δε χρησιμοποιείται ο Mono operator. Μπορούμε να καλέσουμε και την default μέθοδο findAll() που

επιστρέφει μία λίστα από τους χρήστες και χρειάζεται να είναι το πρόγραμμα σε αναμονή μέχρι να ολοκληρωθεί η μετάδοση όλων των εγγραφών. Το ίδιο είναι και για τα repositories των προϊόντων και του καλαθιού. Παρατίθεται ο παρακάτω κώδικας :

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
  @Query("select * from products where itemName =:itemName")
  Product findByItemName(String itemName);
}

@Repository
public interface CartRepository extends JpaRepository<Cart, Long> {
  @Query("select * from cart where user_id =:userId")
  ArrayList<Cart> findCartById(Long userId);
}
```

Απόσπασμα κώδικα 9.3 JPA Repository προϊόντων και Παραγγελιών

Όπως και στην RxJava έτσι και εδώ υπάρχει μια ενδιάμεση κλάση με το annotation @Service που στο πρότζεκτ ονομάζεται RestApiServices και συνδέει τα repositories με τους controllers με τελικό στόχο τη δημιουργία των web services. Παρακάτω παρατίθεται κώδικας στην κλασική Java spring boot:

```
public Product newProduct(Product product){
  return productRepository.save(product);
}

public ArrayList<Product> getAll(){
  return productRepository.findAll();
}

public Product getProduct(long id) { return productRepository.findById(id); }

public Product updateProduct(long id,Product product) {
  return this.productRepository.findById(id)
        .map(product -> {
           product.setItemName(product.getItemName());
           product.setProductPrice(product.getProductPrice());
           return productRepository.save(product);
        });
}

public void deleteProduct(final long id){
```

```
   return this.productRepository.findById(id)
           .map(product ->
                   this.productRepository.delete(product));
}

public User getUser(long id) { return userRepository.findById(id); }

public User newUser(User user){
   return userRepository.save(user);
}

public User updateUser(long id,User user) {
   return this.userRepository.findById(id)
           .map(user1 -> {
              user1.setName(user.getName());
              user1.setRole(user.getRole());
              return userRepository.save(user1);
           });
}

public ArrayList<User> getAllUser(){
   return userRepository.findAll();
}

public User getUserByIdPassword(long id, String password){
   return this.userRepository.getUserByIdPassword(id,password);
}

public void deleteUser(final long id){
   return this.userRepository.findById(id)
           .flatMap(user ->
                   this.userRepository.delete(user));
}

public Cart newCart(Cart cart){
   return cartRepository.save(cart);
}

public ArrayList<Cart> getCartById(Long id){
   return cartRepository.findCartById(id);
}

public Car updateCart(long id,Cart cart) {
```

```
  return this.cartRepository.findById(id)
        .map(cart1 -> {
           cart1.setProductId(cart.getProductId());
           cart1.setProductName(cart.getProductName());
           cart1.setProductPrice(cart.getProductPrice());
           cart1.setUserId(cart.getUserId());
           return cartRepository.save(cart1);
        });
}

public void deleteCart(final long id){
  return this.cartRepository.findById(id)
        .map(cart ->
              this.cartRepository.delete(cart));
}
```

Απόσπασμα κώδικα 9.4 κλάση service με απλή Java

Τέλος η δημιουργία των controllers διαφέρει στην Java spring boot όπως θα δούμε στον κώδικα διότι δε χρησιμοποιεί τους operators Mono και Flux. Χρησιμοποιούνται τα ίδια annotations για την κατασκευή web services όπως @RestController, @PostMapping, @GetMapping, @PutMapping και @DeleteMapping.

```
@Autowired
private final RestApiServices restApiServices;

//constractor
public UploadController(RestApiServices restApiServices){
  this.restApiServices = restApiServices;
}

//get all products
@GetMapping("/products")
public List<Product> listProducts(){
  return restApiServices.getAll();
}

//get all products by id of product
@GetMapping("/products/{id}")
public ResponseEntity<Product> product(@PathVariable Long id) {
  Product product =  restApiServices.getProduct(id);
  return ResponseEntity.ok(product);
```

```
}

//create new product
@PostMapping(value = "/create")
public Product addNewProduct(@RequestBody Product product) {
   if (product != null){
       product.toString();
       System.out.println(product.toString());
   }
   return restApiServices.newProduct(product);
}

//update product by id
@PutMapping("/upload/{id}")
public ResponseEntity<Product> updateProduct(@PathVariable Long
id,@RequestBody Product product){
   return this.restApiServices.updateProduct(id,productMono);
}

//create new product
@PostMapping(value = "/create")
public Product addNewProduct(@RequestBody Product product) {
   if (product != null){
       product.toString();
       System.out.println(product.toString());
   }
   return restApiServices.newProduct(product);
}

//update product by id
@PutMapping("/upload/{id}")
public ResponseEntity<Product> updateProduct(@PathVariable Long
id,@RequestBody Product product){
   Optional<Product> updateProduct
this.restApiServices.updateProduct(id,product);
   updateProduct.map(e -> ResponseEntity.ok(e))
                   .orElse(ResponseEntity
                           .status(HttpStatus.NOT_FOUND)
                           .body(MessageUtil.parse(MSG_404_Product, id + "")));
}

//delete product by id
@DeleteMapping("/delete/{id}")
```

```
public Long deleteProduct(@PathVariable Long id){
   Long isRemoved = restApiServices.deleteProduct(id)
    if (!isRemoved) {
           return new ResponseEntity<>(HttpStatus.NOT_FOUND);
       }

       return new ResponseEntity<>(id, HttpStatus.OK);
}

@GetMapping("/users/{id}")
public ResponseEntity<User> user(@PathVariable Long id) {
   Optional<User> user =  restApiServices.getUser(id);
   return ResponseEntity.ok(user);
}

//create user
@PostMapping(value = "/create/user")
public User addNewUser(@RequestBody User user) {
   if (user != null){
       user.toString();
       System.out.println(user.toString());
   }
   return restApiServices.newUser(user);
}

//update user by id
@PutMapping("/upload/user/{id}")
public ResponseEntity<User> updateUser(@PathVariable Long id,@RequestBody
User user){
   Optional<User> updateUser = restApiServices.updateUser(id,user);
   updateUser.map(e -> ResponseEntity.ok(e))
                   .orElse(ResponseEntity
                           .status(HttpStatus.NOT_FOUND)
                           .body(MessageUtil.parse(MSG_404_USER, id + "")));
}

//get all users
@GetMapping("/users")
public List<User> listUsers(){
   return restApiServices.getAllUser();
}

//get user by id and password
```

```
@GetMapping("/users/{id}/{password}")
public ResponseEntity<User> userByPassword(@PathVariable Long id,
@PathVariable String password){
   Optional<User> user =  restApiServices.getUserByIdPassword(id,password);
   return ResponseEnity.ok(user);
}

//delete user by id
@DeleteMapping("/delete/user/{id}")
public Long deleteUser(@PathVariable Long id){
   Long isRemoved = restApiServices.deleteUser(id);
    if (!isRemoved) {
          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
       }

       return new ResponseEntity<>(id, HttpStatus.OK);
  }

@GetMapping("/cart/{id}")
   public List<Cart> cart(@PathVariable Long id) {
      return restApiServices.getCartById(id);

   }

   //create cart for each user
   @PostMapping(value = "/create/cart")
   public Cart addNewCart(@RequestBody Cart cart) {
      if (cart != null){
         cart.toString();
         System.out.println(cart.toString());
      }
      return restApiServices.newCart(cart);
   }

      //update cart by user id
   @PutMapping("/upload/cart/{id}")
   public ResponseEntity<Cart> updateCart(@PathVariable Long
id,@RequestBody Cart cart){
      Optional<Cart> updateCart =  restApiServices.updateCart(id,cart);
      return updateCart.map(e -> ResponseEntity.ok(e))
                  .orElse(ResponseEntity
                        .status(HttpStatus.NOT_FOUND)
                        .body(MessageUtil.parse(MSG_404_Cart, id + "")));
```

```
  }
  //delete cart by card id
  @DeleteMapping("/delete/cart/{id}")
  public ResponseEntity<Long> deleteCart(@PathVariable Long id){
      Long isRemoved =  restApiServices.deleteCart(id);
      if (!isRemoved) {
          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
      }

      return new ResponseEntity<>(id, HttpStatus.OK);
  }
}
```

Απόσπασμα κώδικα 9.5 κλάση controller με απλή Java

10 Επίλογος

10.1 Σύνοψη και συμπεράσματα

Οι ενότητες του αντιδραστικού προγραμματισμού ανταποκρίνονται πολύ καλά. Είναι ικανές στο να δίνουν στους χρήστες αποτελεσματική και διαδραστική ανατροφοδότηση. Ως αποτέλεσμα η εφαρμογή βελτιώνεται.

Ο κώδικας του αντιδραστικού προγραμματισμού είναι καθαρός και πιο συνοπτικός και είναι ευανάγνωστος. Επίσης ο αντιδραστικός προγραμματισμός διαχειρίζεται πιο καλά τα σφάλματα και είναι καλύτερος στη διαχείριση μεγάλων δεδομένων έτσι ώστε να μην μπλοκάρει η μνήμη. Αν και η κατανόηση και η εκμάθηση του αντιδραστικού προγραμματισμού μπορεί να απαιτεί αφοσιωμένη εργασία και διάβασμα, είναι πολύ χρήσιμος στις μέρες μας. Έχει πάρα πολλά πλεονεκτήματα με αποτέλεσμα να διευκολύνει την εργασία των προγραμματιστών, να ενισχύει την εκτέλεση των εφαρμογών και επιπλέον να βελτιώνει την εμπειρία των χρηστών.

Βιβλιογραφία

---

[1] Introduction to RXJava https://www.baeldung.com/rx-java

[2] Reactive programming with spring framework 5
https://www.udemy.com/

[3] R2DBC Reactive Database Example in Spring
https://codetinkering.com/r2dbc-reactive-database-example-in-spring/

[4] ReactiveX Introduction https://reactivex.io/intro.html

[5] The IoT - architecture on the principles of Reactive Programming
https://ieeexplore.ieee.org/document/8109897

[6] Reactive programming
https://www.techtarget.com/searchapparchitecture/definition/reactive-programming

[7] Reactive extension https://en.wikipedia.org/wiki/ReactiveX

[8] Observer pattern https://en.wikipedia.org/wiki/Observer_pattern

[9] RxJava
https://factoryhr.medium.com/understanding-java-rxjava-for-beginners-5eacb8de12ca

[10] Reactor Project
https://projectreactor.io/docs/core/release/reference/#core-features

[11] R2DBC
https://mariadb.com/resources/blog/unblock-your-applications-with-r2dbc-spring-data-and-mariadb/
https://spring.io/projects/spring-data-r2dbc#overview

[12] Reactive programming
https://proandroiddev.com/understanding-rxjava-subscribeon-and-observeon-744b0c6a41ea