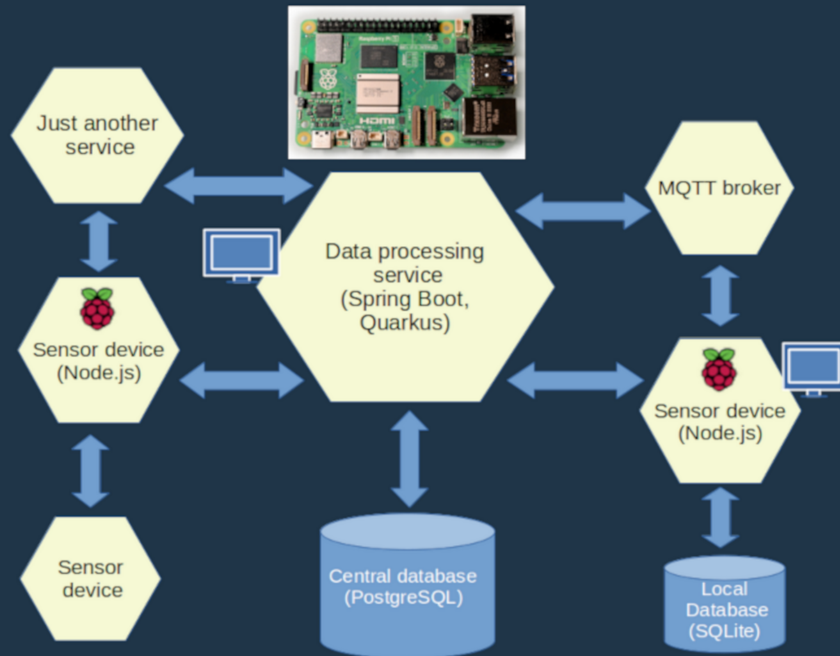
Enterprise Development

with

Raspberry Pi



Sergiy Pylypets

Table of Contents

Introduction......4
Chapter 1 Development of cloud-native services......6
1.1 Hardware and OS set-up, development tools......6
1.2 Project set-up and implementation of the business logic......10
1.3 Securing applications and services with Keycloak......35
1.3.1 Keycloak configuring......37
1.3.2 Application security enhancement......38
1.4 Event streaming with Apache Kafka......45
Chapter 2 Data Storage and Engineering......53
2.1 Implementing the enterprise persistence layer with PostgreSQL database......53
2.1.1 Database set-up......53
2.1.2 Connecting to the database......55
2.1.3 Data migration with Flyway......57
2.1.4 Using Testcontainers API for persistence layer integration tests......59
2.2 Data engineering with Raspberry Pi......62
2.2.1 PySpark installation and set-up......62
2.2.2 Visitor poll data ingestion and processing......63
2.2.3 Analytical data visualization with Grafana......65
2.2.4 Data clean-up and analysis......67
Chapter 3 Raspberry Pi in Cloud......73
3.1 Local Kubernetes cluster set-up......73
3.2 Deploying a SpringBoot application to Kubernetes......76
3.2.1 Building the application image......77
3.2.2 Deploying the application to the cluster......79
3.3 Kubernetes cluster set-up......82
3.4 Deploying applications to a Kubernetes cluster with Tilt......85
3.5 Deploying applications to a Kubernetes cluster with Helm charts......90
3.6 Observability of applications running in Cloud......101
3.6.1 Monitoring the application metrics with Prometheus......101
3.6.2 Setting Up Alerts with Prometheus......104
Chapter 4 Enterprise development with Quarkus......110
4.1 Quarkus environment set-up......110
4.2 The collection application in Quarkus......110
4.3. Building native executables with Quarkus......127
Chapter 5 Development of multi-channel monitoring systems......129
5.1. Development of the data processing layer......131
5.2 Inbound (monitoring) communication with physical devices......140
5.3 Cloud deployment with Quarkus......149
5.4 Outbound (control) communication with physical devices......157
5.4.1 Hardware Set-Up: Relay Board Connection, Circuit Assembling......158

5.4.2 Software Set-Up: OS and Library Installation, the Control Software Development..........158
5.5 Lightweight messaging with MQTT..........163
Conclusion..........171
Index..........172
Index..........172
A..........172
B..........172
C..........172
F..........172
G..........172
H..........172
J..........172
K..........172
L..........172
M..........172
N..........172
O..........172
P..........172
Q..........172
R..........172
S..........172
T..........173

Introduction

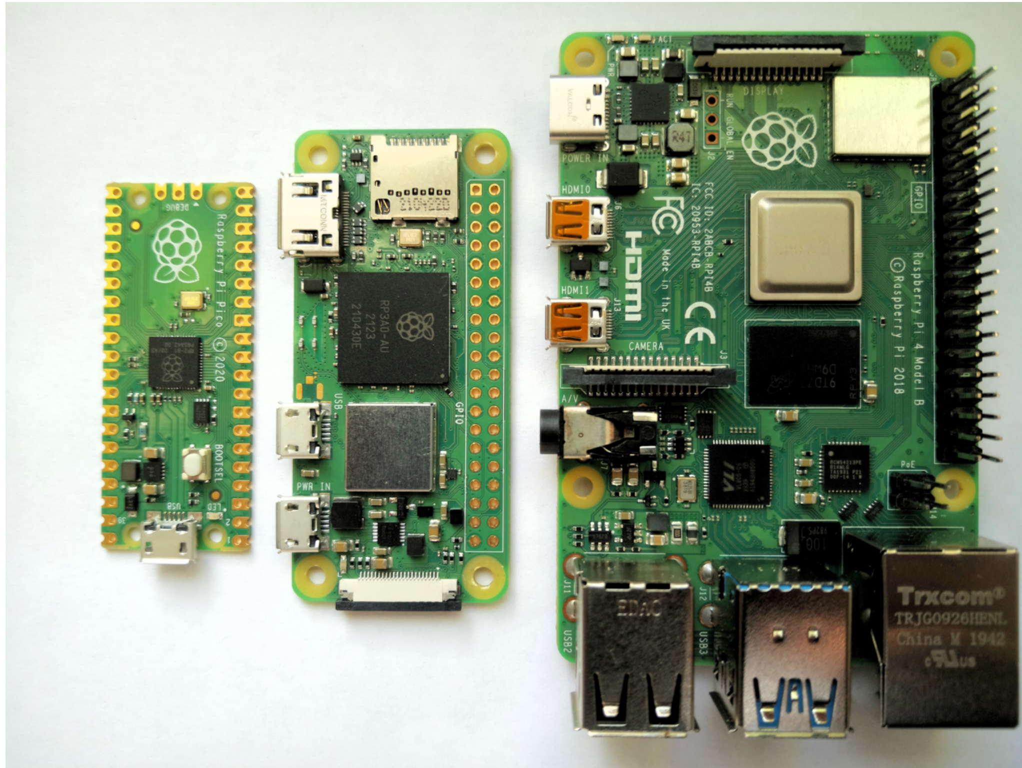
Although the Raspberry Pi platform was initially created for educational and hobby purposes, it got spread quickly on various industrial and scientific domains. Now we can see RPi devices in many places, from underwater devices up to orbital space stations. The Raspberry family was extended by a wide range of devices from Raspberry Pico micro controllers to Raspberry 5 B single-board computers (SBC), which are comparable by their performance with modern desktops and notebooks. Also, a lot of Raspberry PI clones appeared, like Orange Pi, Banana Pi, ROCK Pi, and others.

Besides Raspberry Pi, there are other SBC systems available on the market, like BeagleBone, LattePanda, Odroid, to name a few.

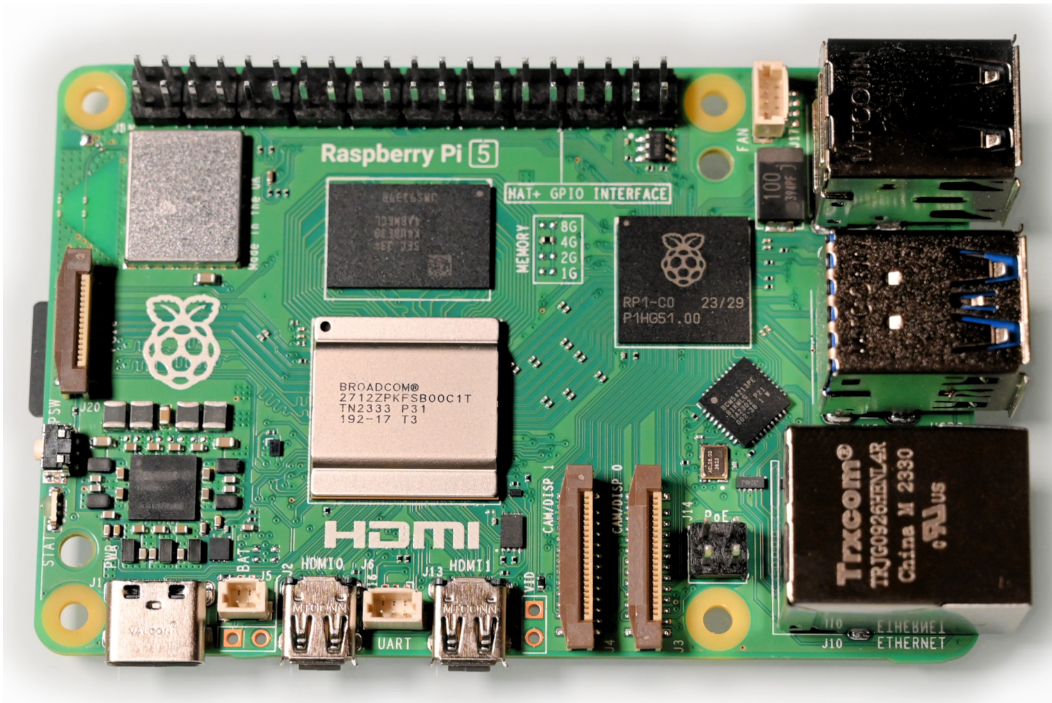
In this book, we are going to check possibilities of using Raspberry PI platform in such a special domain as enterprise development. RPi computers are equipped with ARM processors using RISC instruction set. These processors consume less power and provide a good performance for lightweight tasks. According to resent researches, in some cases, usage of RPi devices can ensure significant power savings. In context of enterprise development, it can be interesting, for example, for implementing micro-service architecture or multi-channel monitoring systems. We will develop a simple example applications, which nevertheless have all features and functionality of real-world enterprise projects.

For real full-scale projects, we advise to use Raspberry Pi 5 boards, whose performance and capabilities are substantially higher. Also, it is advisable to equip the board with a sufficient cooling system.

Raspberry Pi family members: Pico, Zero, 4B



The current flagship - Raspberry Pi 5



Chapter 1 Development of cloud-native services

1.1 Hardware and OS set-up, development tools

Raspberry Pi single-board computers don’t have internal data storage equipment like hard drives. So, we need to use micro SD cards or SSD devices, e.g. PCIe NVMe drives. Such drives can be connected to Raspberry Pi with the usage of special HATs, or externally, via USB 3.0 ports. There is a great variety of external SSD devices of many types and capacities. For purposes of this book, we used a Patriot NVMe drive with an external case. To ensure the maximum data transfer rate, we used a 3.2 USB cable. To avoid the CPU overheating, we used a Raspberry Pi 5 active cooler combining a fan with a big aluminum heat sink.

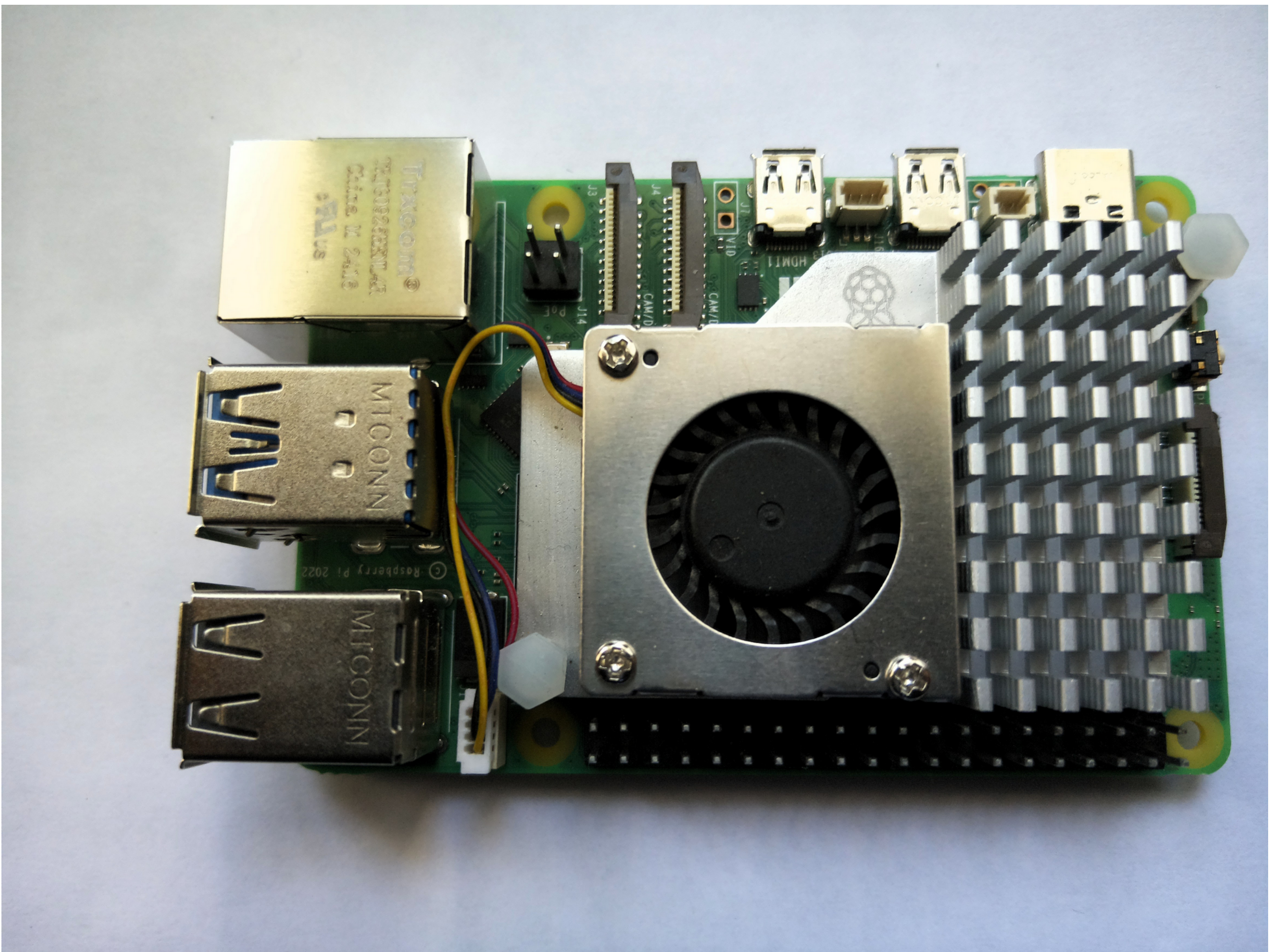


Figure 1.1 Raspberry Pi 5 board with mounted combined active cooler

To use Raspberry Pi SBC, we need to install some OS onboard. There are several way to do it, we are going to use Raspberry Pi Imager. It is free, has a friendly, intuitive UI, and can be run in any computer

with Windows, MacOS, or Linux OS installed (including another Raspberry Pi board), see figure 1.2. Provided having the SSD drive connected and the imager application running on our base computer, we can install OS of our choice on the Raspberry board. Raspberry Pi 4B supports several Linux-based operation systems. We will use Raspberry Pi OS Full (64-bit), which is based on a Debian Trixie distribution and includes a bunch of useful applications. We can select it in the Operating System drop-down under Raspberry Pi OS (Other) menu item.
In the Storage panel, we select the connected SSD drive. Then we push the Next button to set up some configuration parameters (optionally) and start installation.

Be aware that all the data stored on the SSD will be erased during the installation process!

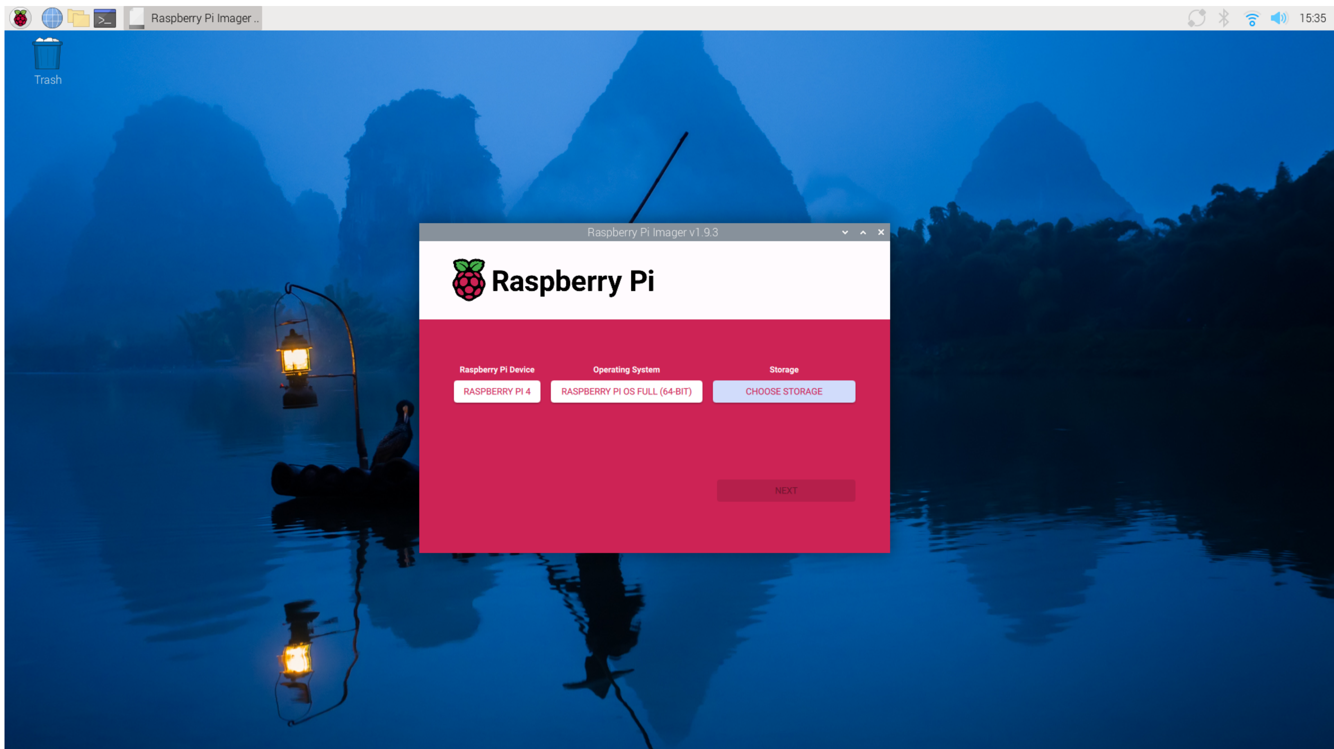


Figure 1.2   Installing OS with Raspberry Pi Imager

Provided having OS installed on our SSD drive, we can connect it to a 3.0 USB connector (blue-colored) of our Raspberry board and power up board. Then we should see some booting information on the display, something like: “Progress: Trying boot mode USB-MSD”

If the booting process was successful, we can see the initial Raspberry OS desktop screen, as shown in figure 1.3.

Figure 1.3 Initial Raspberry OS (Debian Trixie) desktop screen.

Now we can check what stuff we have for enterprise development out-of-box. Each Raspberry Pi distribution has a Python version:

```
$ python --version
Python 3.13.5
```

Also, the full installation includes a Java distribution:

```
$ java --version
openjdk 21.0.8 2025-07-15
OpenJDK Runtime Environment (build 21.0.8+9-Debian-1)
OpenJDK 64-Bit Server VM (build 21.0.8+9-Debian-1, mixed mode, sharing)
```

Spring Boot is one of the most popular enterprise development frameworks and we are going to use it in our projects. To work with Spring Boot technology, we choose Spring Tool Suite IDE. It is built on the base of Eclipse and tailored for developing enterprise applications using Spring Framework and Spring Boot. It is free, open-source, and have a good community support. At the moment of writing, the latest STS version is the 4.32.0 one. We can download the Linux ARM_64 distribution, suitable for

Raspberry Pi environment, from Spring download page, unpack it into directory of our choice and run the SpringToolSuite4 executable file. The application screen will look like it is shown in fig 1.4.

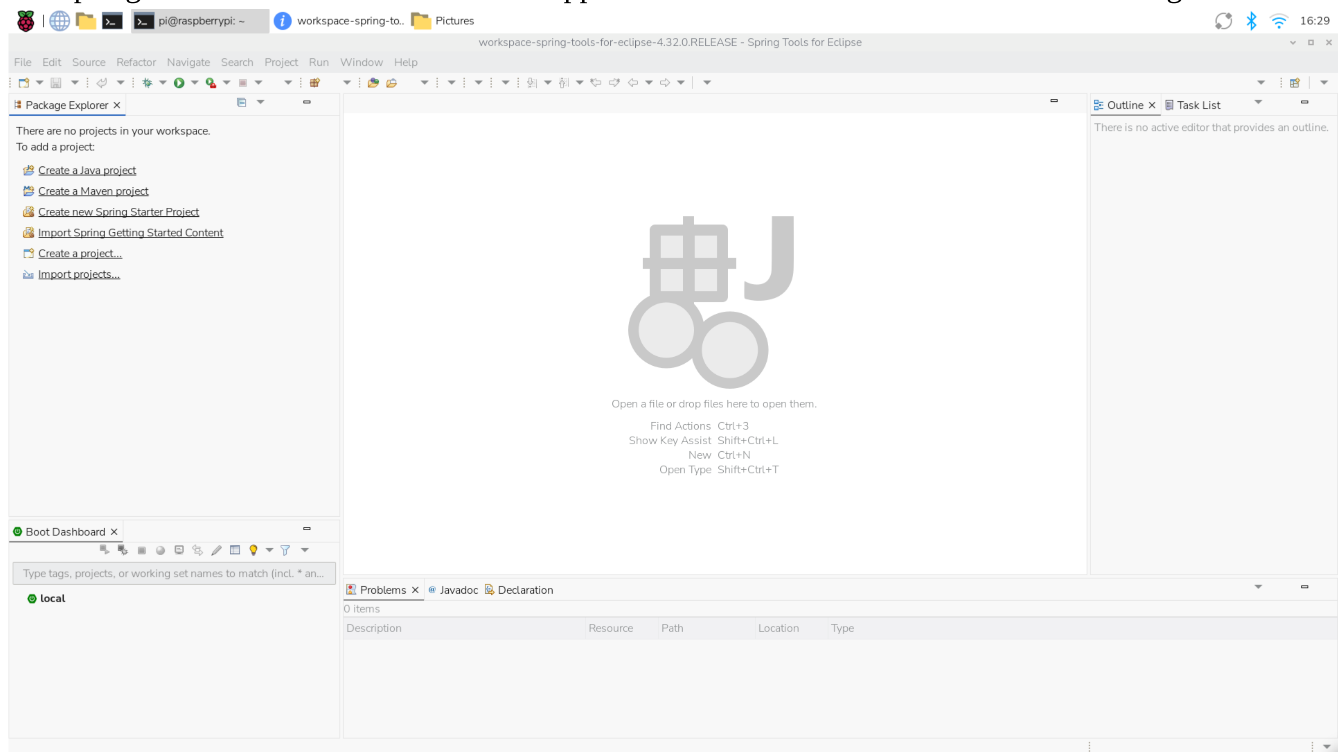
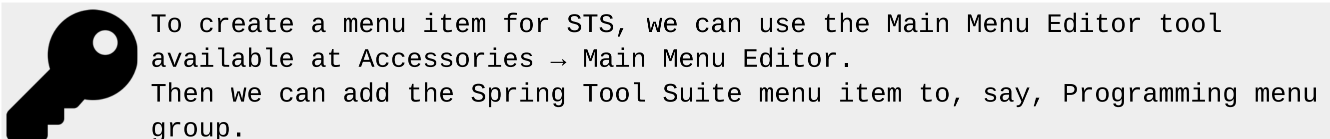
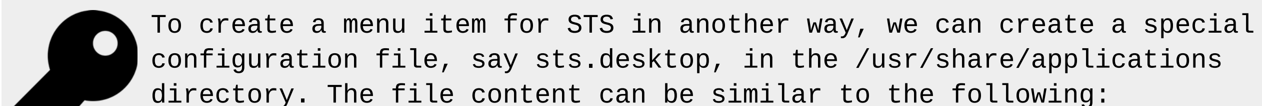


Figure 1.4 Initial window of Spring Tool Suit IDE



To create a menu item for STS, we can use the Main Menu Editor tool available at Accessories → Main Menu Editor.
Then we can add the Spring Tool Suite menu item to, say, Programming menu group.



To create a menu item for STS in another way, we can create a special configuration file, say sts.desktop, in the /usr/share/applications directory. The file content can be similar to the following:

```
[Desktop Entry]
Name=Spring Tool Suite
Comment=Code Editing. Redefined.
GenericName=Text Editor
Exec=/home/pi/Tools/sts-4.32.0.RELEASE/SpringToolSuite4
Icon=/home/pi/Tools/sts-4.32.0.RELEASE/icon.xpm
Type=Application
StartupNotify=false
StartupWMClass=Code
Categories=TextEditor;Development;IDE;
MimeType=application/x-code-workspace;
```

```
Actions=new-empty-window;
Keywords=sts;

where the Exec and Icon VALUES should correspond to your STS installation path.
```

Another great tool for enterprise development in Java (and not only) is JetBrains IDEA. They provide Linux ARM64 distribution suitable for Raspberry Pi. According to the documentation, while the minimal RAM size for working with IDEA is 2 GB, preferable size is at least 8 GB.

Provided having a basic tool set installed and running, we can start development.

According to Cloud Native Computing Foundation, “CNCF Cloud Native Definition v1.0,” http://mng.bz/de1w, cloud-native applications are loosely coupled systems that are resilient, manageable, and observable. In the next section, we are going to develop such an application in Raspberry Pi environment.

1.2 Project set-up and implementation of the business logic

For our first project, we are going to use domain model from The Magic of Spring Data article, updating the stuff with up-to-date technologies and approaches. For example, we are going to use the reactive Spring programming model, records, and Spring Data R2DBC library for implementing the persistence layer. To quickly develop a working draft of our application, we can use in-memory H2 database, which is automatically activated by the Spring Boot engine provided having the corresponding dependencies added to the pom.xml descriptor. In chapter 2, we will discuss how to use PostgreSQL database for the data storage in production environment and Testcontainerts library for testing the persistence layer. We open the `STS File → New → Spring Starter Project` wizard and set a new project up step-by-step, including dependencies mentioned in the article and those we want to use in the updated version of our application, like it is shown in figures 1.5 and 1.6.

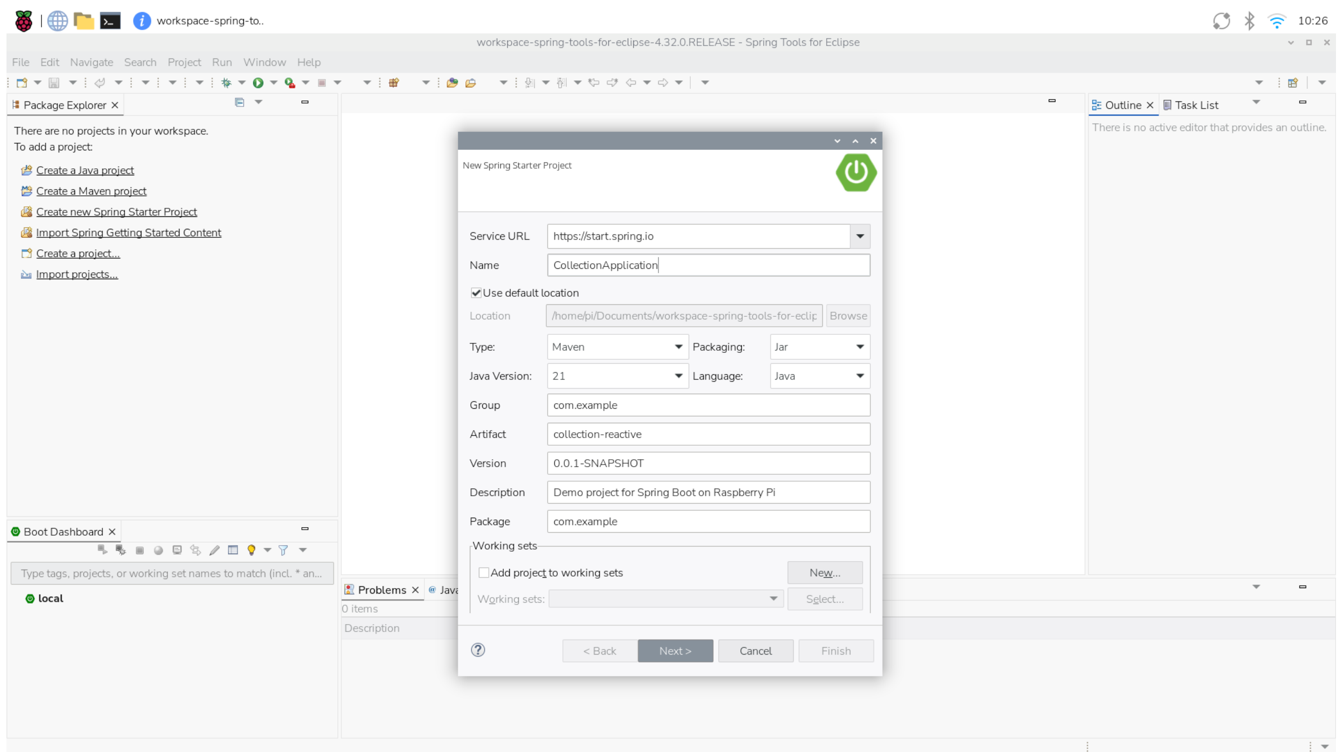


Figure 1.5 Creating a new project in the Spring Boot starter wizard.

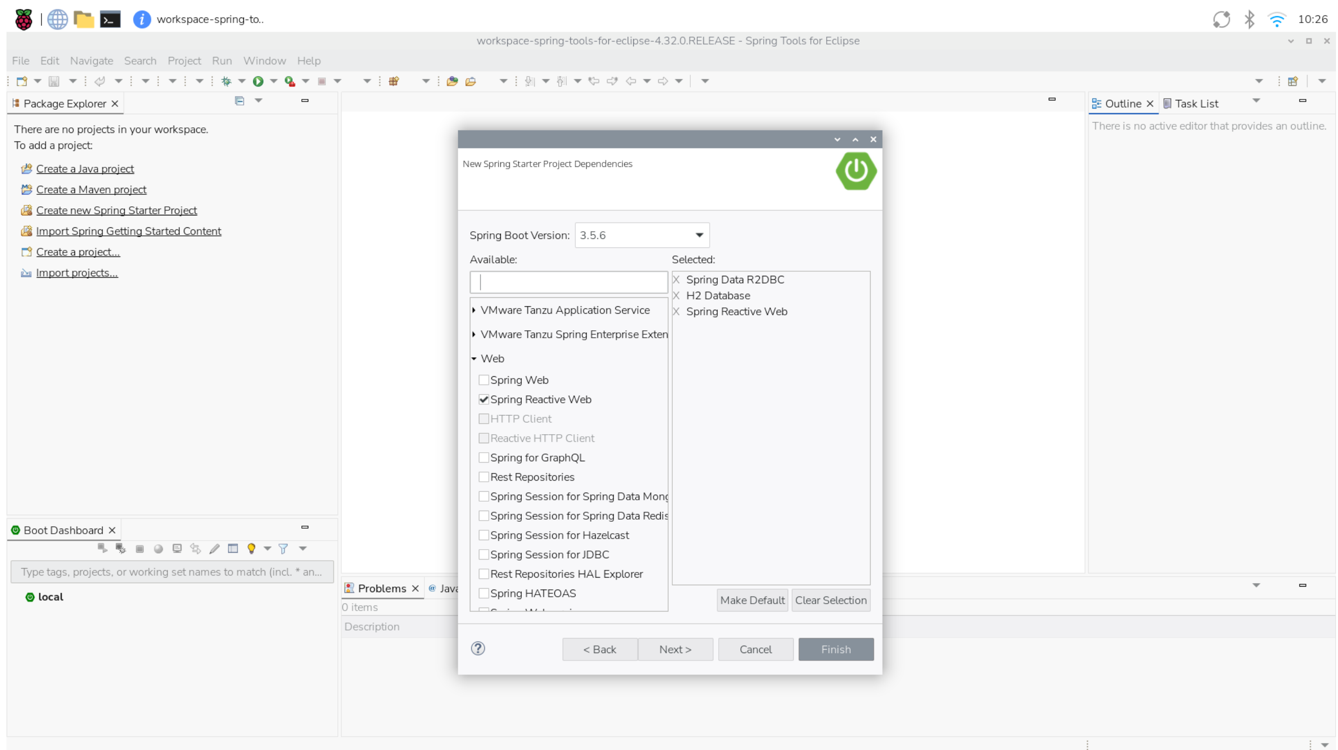


Figure 1.6 Choosing necessary dependencies in the Spring Boot starter wizard.

The pom.xml descriptor is shown in listing 1.1.

Listing 1.1 The Collection Application project pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>3.5.7</version>
            <relativePath/> <!-- lookup parent from repository -->
      </parent>
      <groupId>com.example</groupId>
      <artifactId>collection-reactive</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <name>CollectionApplication</name>
      <description>Demo project for Spring Boot on Raspberry Pi</description>
      <url/>
      <licenses>
            <license/>
      </licenses>
      <developers>
            <developer/>
      </developers>
      <scm>
            <connection/>
            <developerConnection/>
            <tag/>
            <url/>
      </scm>
      <properties>
            <java.version>21</java.version>
      </properties>
      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-data-r2dbc</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-webflux</artifactId>
            </dependency>
            <dependency>
                <groupId>io.github.joselion</groupId>
                <artifactId>spring-r2dbc-relationships</artifactId>
                <version>1.1.0</version>
            </dependency>
            <dependency>
                  <groupId>com.h2database</groupId>
```

```
                  <artifactId>h2</artifactId>
                  <scope>runtime</scope>
            </dependency>
            <dependency>
                  <groupId>io.r2dbc</groupId>
                  <artifactId>r2dbc-h2</artifactId>
                  <scope>runtime</scope>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-test</artifactId>
                  <scope>test</scope>
            </dependency>
            <dependency>
                  <groupId>io.projectreactor</groupId>
                  <artifactId>reactor-test</artifactId>
                  <scope>test</scope>
            </dependency>
      </dependencies>

      <build>
            <plugins>
                  <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                  </plugin>
            </plugins>
      </build>

</project>
```

While using Java classes for DTO instead of records (and not only for that), we can add dependencies for Lombok project, which is a utility package generating a lot of boil-plate code for enterprise applications. The pom.xml descriptor with Lombok dependencies is shown in listing 1.2.

Listing 1.2 Enabling Lombok in The Collection Application project

```
      <dependencies>
        ...
            <dependency>
                  <groupId>org.projectlombok</groupId>
                  <artifactId>lombok</artifactId>
                  <optional>true</optional>
            </dependency>
        ...
       <build>
            <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

To add support for Lombok in your IDE, you need to run the Lombok distribution jar file as a Java application and select your IDE there, like it is shown in figure 1.7

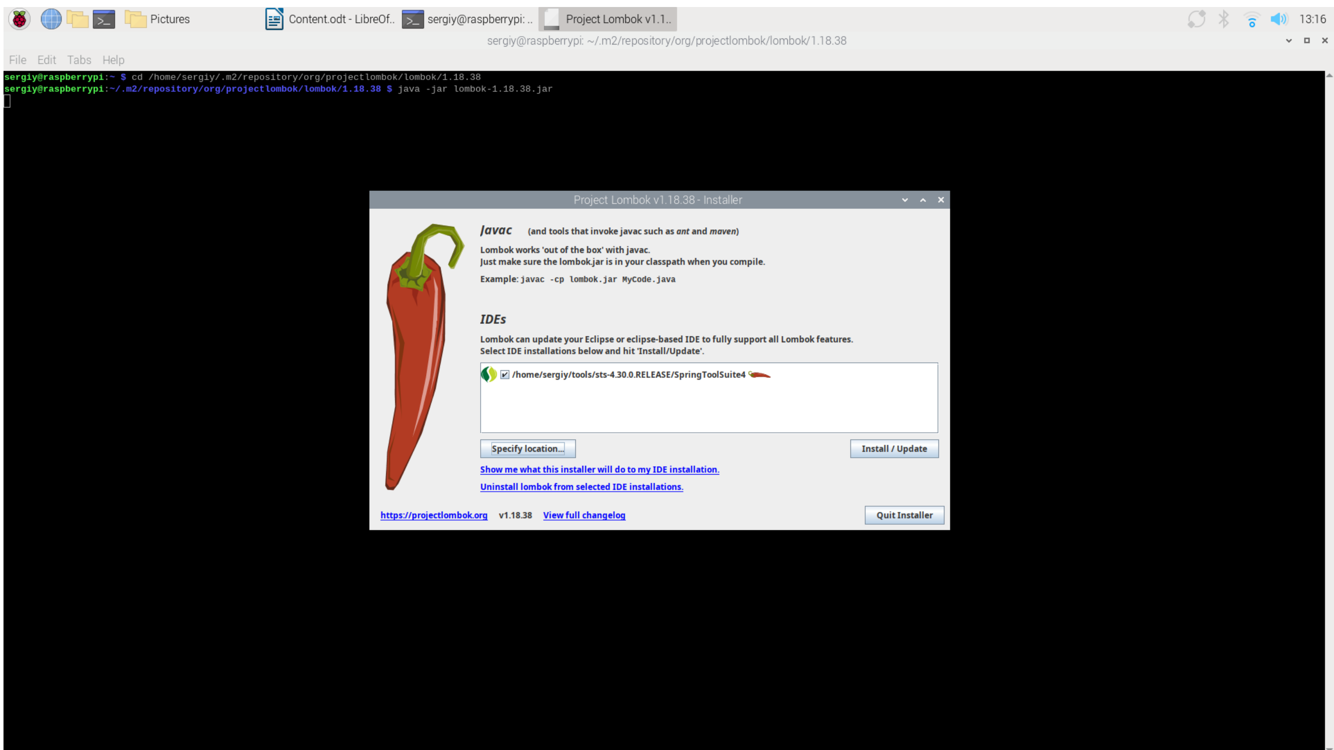


Figure 1.7 Enabling Lombok support in STS.

Provided using the H2 database, we can populate our database with test data using the following SQL scripts with predefined names, schema.sql for the database structure and data.sql for the data to be inserted into the tables:

Listing 1.3 Database schema initialization script - schema.sql

```
DROP SCHEMA IF EXISTS collection cascade;
CREATE SCHEMA collection;

DROP SEQUENCE IF EXISTS collection.CollectionItems_seq;
CREATE SEQUENCE collection.CollectionItems_seq
INCREMENT BY 1
START WITH 1;

DROP SEQUENCE IF EXISTS collection.Images_seq;
CREATE SEQUENCE collection.Images_seq
INCREMENT BY 1
START WITH 1;

CREATE TABLE collection.ITEMS (
        id bigint not null default nextval('CollectionItems_seq'),
        name varchar(255),
            summary varchar(1000),
            description text,
            country varchar(2),
```

```
            creation_year smallint,
            price numeric(10,2),
            small_image bigint,
            image bigint,
            topics varchar(255) array,
        PRIMARY KEY (id)
);

CREATE TABLE collection.IMAGES (
        id bigint not null default nextval('Images_seq'),
            file_name varchar(255),
        PRIMARY KEY (id)
);
```

Listing 1.4 Database data script - data.sql

```
INSERT INTO collection.IMAGES(id, file_name) VALUES(1, 'juke-small.jpg');
INSERT INTO collection.IMAGES(id, file_name) VALUES(2, 'juke.jpg');
INSERT INTO collection.IMAGES(id, file_name) VALUES(3, 'penny-black-small.png');
INSERT INTO collection.IMAGES(id, file_name) VALUES(4, 'penny-black.png');
INSERT INTO collection.IMAGES(id, file_name) VALUES(5, 'juggling-juke-small.jpg');
INSERT INTO collection.IMAGES(id, file_name) VALUES(6, 'juggling-juke.jpg');

INSERT INTO collection.ITEMS (id, name, summary, description, creation_year,
country, price, small_image, image, topics)
 VALUES(1, 'The Penny Black', 'The very first stamp',
        'The very first post stamp but surprisingly not the most expensive one',
1840, 'uk', 1000, 3, 4, ARRAY['history']);
INSERT INTO collection.ITEMS (id, name, summary, description, creation_year,
country, price, small_image, image, topics)
 VALUES(2, 'Juke', 'The Juke stature', 'Porcelain stature of Juke', 1996, 'us',
1000000, 1, 2, ARRAY['arts', 'programming']);
INSERT INTO collection.ITEMS (id, name, summary, description, creation_year,
country, price, small_image, image, topics)
 VALUES(3, 'Juggling Juke', 'The Juggling Juke painting', 'Post modernist oil
painting of the juggling Juke', 2000, 'us', 2000000, 5, 6, ARRAY['arts',
'programming']);
```

We can place the schema.sql and data.sql files to the src/test/resources to feed the Junit tests with test data. Also, we can put the schema.sql file to src/main/resources to initialize the database upon starting the application.

To enable the H2 console, we can set spring.h2.console.enabled=true in application.properties and add the following Maven dependencies:

```
<!-- Dependencies to enable H2 console -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-jdbc</artifactId>
            </dependency>
<!-- Includes Tomcat -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
```

```
                        <artifactId>spring-boot-starter-web</artifactId>
                    </dependency>
        Then, providing running the application on port 8888, we can access the
        database administration UI at http://localhost:8888/h2-console, like it
        shown in figure 1.8.
```

Figure 1.8 H2 database administration UI

Our domain model consists of two entities: CollectionItem and Image. Image entity is associated with CollectionItem via two CollectionItem's fields: image and smallImage;
At the time of writing, Spring Data R2DBC library doesn't support entity relations, see, for example, https://github.com/spring-projects/spring-data-r2dbc/issues/99, so we need to provide our own implementation to support the one-to-one relations between CollectionItem and Image entities or to use third-part libraries, for example spring-r2dbc-relationships.
We can manage the persistence life cycle for each entity independently, with the usage of auto-generated code provided by R2DBC technology, like save() and delete() methods by extending one of the standard R2DBC interfaces, for example, R2DBCRepository interface. Also, for retrieving filtered CollectionItem objects together with the corresponding Image objects and saving newly created entities together, we can create a custom interface with all necessary methods, say CustomizedCollectionItemRepository, and provide a custom implementation for it according to Spring Data repository extension rules. Here are the components to implement the CollectionApplication persistence layer.

The domain objects are presented in listings 1.5 and 1.6. We enhance the entities with custom ORM code in the corresponding fromRow() methods.

Listing 1.5 CollectionItem.java

```
package com.example.model;

import java.math.BigDecimal;
```

```
import java.util.Map;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Column;
import org.springframework.data.relational.core.mapping.Table;

@Table(name="ITEMS", schema="collection")
public record CollectionItem (

      @Id
      long id,

      BigDecimal price,

      Image smallImage,

      Image image,

      String name,

      String summary,

      String description,

      @Column(value="creation_year")
      Short year,

      String country,

      Object[] topics
) {
      public static CollectionItem of(
                     long id, BigDecimal price, Image smallImage, Image image,
                     String name, String summary, String description, Short year,
                     String country, Object[] topics)  {
            return new CollectionItem(id, price, smallImage, image, name, summary,
                                    description, year, country, topics);
      }

      public static CollectionItem fromRow(Map<String, Object> row) {
            return of(
                        Long.parseLong(row.get("i_id").toString()),
                        (BigDecimal) row.get("i_price"),
                        Image.fromRow(row, "sim_id"),
                        Image.fromRow(row, "im_id"),
                        (String) row.get("i_name"),
                        (String) row.get("i_summary"),
                        (String) row.get("i_description"),
                        (Short) row.get("i_creation_year"),
                        (String) row.get("i_country"),
                        (Object[]) row.get("i_topics")
                  );
      }
```

Listing 1.6 Image.java

```
package com.example.model;

import java.util.Map;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Column;
import org.springframework.data.relational.core.mapping.Table;

@Table(name="IMAGES" , schema="collection")
public record Image (
            @Id
            long id,

            @Column("file_name")
            String fileName
            )
{

      public static Image of(long id, String fileName) {
            return new Image(id, fileName);
      }

      public static Image fromRow(Map<String, Object> row, String keyColumn) {
            String prefix = keyColumn.substring(0, keyColumn.indexOf('_') + 1);
            if(row.get(keyColumn) != null) {
                  return    Image.of(Long.parseLong(row.get(keyColumn).toString()),
(String) row.get(prefix + "file_name"));
            } else {
                  return null;
            }
      }
}
```

Also, we define the Image repository interface, which just extends R2dbcRepository, and the custom CollectionItem interface with the corresponding implementation, see listings 1.7 – 1.9.

Listing 1.7 ImageRepository interface

```
package com.example.repository;

import org.springframework.data.r2dbc.repository.R2dbcRepository;

import com.example.model.Image;
```

```
public interface ImageRepository extends R2dbcRepository<Image, Long> {}
```

Listing 1.8 Custom CollectionItemRepository interface

```
package com.example.repository;

import com.example.model.CollectionItem;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface CustomizedCollectionItemRepository<T> {

	Flux<CollectionItem> findAll(String sortBy);

	Flux<CollectionItem> findByCountry(String country);

	Flux<CollectionItem> findByYearInterval(int startYear, int endYear);

	Flux<CollectionItem> findByTopic(String topic);

	Mono<CollectionItem> getById(Long id);

	Mono<CollectionItem> saveWithImages(CollectionItem item);

	Mono<Long> update(CollectionItem item);
}
```

Listing 1.9 Custom CollectionItemRepository implementation

```
package com.example.repository;

import java.util.Map;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.r2dbc.core.DatabaseClient;

import com.example.model.CollectionItem;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public class CustomizedCollectionItemRepositoryImpl implements
CustomizedCollectionItemRepository {

	private static final Logger LOGGER =
LoggerFactory.getLogger(CustomizedCollectionItemRepositoryImpl.class);

	static final String ITEM_SELECT_QUERY = """
			SELECT i.id i_id,i.price i_price,i.name i_name,i.summary
i_summary,i.description i_description,i.creation_year i_creation_year,i.country
i_country, i.topics i_topics,
```

```
                           im.id im_id,im.file_name im_file_name,
                           sim.id sim_id,sim.file_name sim_file_name
                        FROM collection.ITEMS i
                        LEFT JOIN collection.IMAGES im ON im.id=i.image
                        LEFT JOIN collection.IMAGES sim ON sim.id=i.small_image
                        """;
      static final String ITEM_INSERT_QUERY = """
                  INSERT INTO collection.ITEMS (price, name, summary, description,
creation_year, country, topics, image, small_image)
                   VALUES
(:price, :name, :summary, :description, :creation_year, :country, :topics, :image,
:small_image)
                  """;

      static final String ITEM_UPDATE_QUERY = """
                  UPDATE collection.ITEMS SET price = :price, name = :name,
summary = :summary, description = :description, creation_year = :creation_year,
                        country = :country, topics = :topics, image = :image,
small_image = :small_image
                        WHERE id = :id
                  """;

      static final Map<String,String> columnMap = Map.of("price","price",
"name","name", "summary","summary", "description","description",

      "year","creation_year", "country","country", "topics","topics");

      final DatabaseClient dbClient;

      public CustomizedCollectionItemRepositoryImpl(DatabaseClient dbClient) {
            this.dbClient = dbClient;
      }

      @Override
      public Flux<CollectionItem> findAll(String sortBy) {
            if (! columnMap.containsKey(sortBy)) {
                  LOGGER.warn(String.format("Unknown property to sort by: %s",
sortBy));
                  return Flux.empty();
            }
            return dbClient.sql(String.format("%s ORDER BY i.%s",
ITEM_SELECT_QUERY, columnMap.get(sortBy)))
                        .fetch().all()
                        .map(CollectionItem::fromRow);
      }

      @Override
      public Flux<CollectionItem> findByCountry(String country) {
            return dbClient.sql(String.format("%s WHERE i.country = :cntry",
ITEM_SELECT_QUERY))
                        .bind("cntry", country)
                        .fetch().all()
                        .map(CollectionItem::fromRow);
      }
```

```
	@Override
	public Flux<CollectionItem> findByYearInterval(int startYear, int endYear) {
		return dbClient.sql(String.format("%s WHERE i.creation_year 
BETWEEN :startYear AND :endYear", ITEM_SELECT_QUERY))
				.bind("startYear", startYear)
				.bind("endYear", endYear)
				.fetch().all()
				.map(CollectionItem::fromRow);
	}

	@Override
	public Flux<CollectionItem> findByTopic(String topic) {
		return dbClient.sql(String.format("%s WHERE lower(:topic) = ANY 
(i.topics)", ITEM_SELECT_QUERY))
				.bind("topic", topic)
				.fetch().all()
				.map(CollectionItem::fromRow);
	}

	@Override
	public Mono<CollectionItem> getById(Long id) {
		return dbClient.sql(String.format("%s WHERE i.id = :id", 
ITEM_SELECT_QUERY))
				.bind("id", id)
				.fetch().one()
				.map(CollectionItem::fromRow);
	}

	@Override
	public Mono<CollectionItem> saveWithImages(CollectionItem item) {
		return dbClient.sql(ITEM_INSERT_QUERY)
				.filter(s -> s.returnGeneratedValues("id", "price", 
"name", "summary", "description", "creation_year", 
									"country", 
"topics", "image", "small_image"))
						.bind("price", item.price())
						.bind("name", item.name())
						.bind("summary", item.summary())
						.bind("description", item.description())
						.bind("creation_year", item.year())
						.bind("country", item.country())
						.bind("topics", item.topics())
						.bind("image", item.image() != null && 
item.image().id() != null ? item.image().id() : 0)
						.bind("small_image", item.smallImage() != null 
&& item.smallImage().id() != null ? item.smallImage().id() : 0)
						.fetch()
					.first()
					.map(CollectionItem::fromNewRow);
	}

	@Override
	public Mono<Long> update(CollectionItem item) {
```

```
            return dbClient.sql(ITEM_UPDATE_QUERY)
                        .bind("id", item.id())
                        .bind("price", item.price())
                        .bind("name", item.name())
                        .bind("summary", item.summary())
                        .bind("description", item.description())
                        .bind("creation_year", item.year())
                        .bind("country", item.country())
                        .bind("topics", item.topics())
                        .bind("image", item.image() != null && item.image().id() !
= null ? item.image().id() : 0)
                        .bind("small_image", item.smallImage() != null &&
item.smallImage().id() != null ? item.smallImage().id() : 0)
                        .fetch()
                        .rowsUpdated();
      }
}
```

Then we create CollectionItemRepository interface, which extends the custom interface and R2DBCRepository intervace, provided by the Spring Data technology. This way, we can manage relations between CollectionItem and Image entities and use all the auto-generated code, like standard count() and exists() methods.

Listing 1.10 CollectionItemRepository interface defining the application persistence layer

```
package com.example.repository;

import org.springframework.data.r2dbc.repository.R2dbcRepository;
import org.springframework.stereotype.Repository;

import com.example.model.CollectionItem;

@Repository
public interface CollectionItemRepository extends
CustomizedCollectionItemRepository, R2dbcRepository<CollectionItem, Long> {}
```

To enable custom persistence functionality, we can add the corresponding default methods directly to CollectionItemRepository interface, like it is shown in listing 1.11. This approach seems to be more compact, but in this case we introduce a dependency on the application context, which can make the Junit testing more difficult.

Listing 1.11 CollectionItemRepository interface with custom default methods

```
package com.example.repository;

import org.springframework.data.r2dbc.repository.R2dbcRepository;
```

```
import org.springframework.r2dbc.core.DatabaseClient;

import com.example.ApplicationContextUtils;
import com.example.model.CollectionItem;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface CollectionItemRepository extends R2dbcRepository<CollectionItem,
Long> {

      static final DatabaseClient dbClient =
ApplicationContextUtils.getApplicationContext().getBean(DatabaseClient.class);

      static final String ITEM_SELECT_QUERY = """
                  SELECT i.id i_id,i.price i_price,i.name i_name,i.summary
i_summary,i.description i_description,i.creation_year i_creation_year,i.country
i_country, i.topics i_topics,
                           im.id im_id,im.file_name im_file_name,
                           sim.id sim_id,sim.file_name sim_file_name
                        FROM collection.ITEMS i
                        LEFT JOIN collection.IMAGES im ON im.id=i.image
                        LEFT JOIN collection.IMAGES sim ON sim.id=i.small_image
                        """;
      static final String ITEM_INSERT_QUERY = """
                  INSERT INTO collection.ITEMS (price, name, summary, description,
creation_year, country, topics)
                   VALUES
(:price, :name, :summary, :description, :creation_year, :country, :topics)
                  """;
      static final String ITEM_UPDATE_QUERY = """
                  UPDATE collection.ITEMS SET price = :price, name = :name,
summary = :summary, description = :description, creation_year = :creation_year,
                        country = :country, topics = :topics
                        WHERE id = :id
                  """;

      static final Map<String,String> columnMap = Map.of("price","price",
"name","name", "summary","summary", "description","description",
"year","creation_year", "country","country", "topics","topics");

      default Flux<CollectionItem> findAll(String sortBy) {
            if (! columnMap.containsKey(sortBy)) {
                  LOGGER.warn(String.format("Unknown property to sort by: %s",
sortBy));
                  return Flux.empty();
            }
            return dbClient.sql(String.format("%s ORDER BY i.%s",
ITEM_SELECT_QUERY, columnMap.get(sortBy)))
                        .fetch().all()
```

```
                        .map(CollectionItem::fromRow);

      default Flux<CollectionItem> findByCountry(String country) {
            return dbClient.sql(String.format("%s WHERE i.country = :cntry",
ITEM_SELECT_QUERY))
                        .bind("cntry", country)
                        .fetch().all()
                        .map(CollectionItem::fromRow);
      }

      default Flux<CollectionItem> findByYearInterval(int startYear, int endYear)
{
            return dbClient.sql(String.format("%s WHERE i.creation_year
BETWEEN :startYear AND :endYear", ITEM_SELECT_QUERY))
                        .bind("startYear", startYear)
                        .bind("endYear", endYear)
                        .fetch().all()
                        .map(CollectionItem::fromRow);
      }

      default Flux<CollectionItem> findByTopic(String topic) {
            return dbClient.sql(String.format("%s WHERE lower(:topic) = ANY
(i.topics)", ITEM_SELECT_QUERY))
                        .bind("topic", topic)
                        .fetch().all()
                        .map(CollectionItem::fromRow);
      }

      default Mono<CollectionItem> findById(Long id) {
            return dbClient.sql(String.format("%s WHERE i.id = :id",
ITEM_SELECT_QUERY))
                        .bind("id", id)
                        .fetch().one()
                        .map(CollectionItem::fromRow);
      }

      default Mono<Long> saveWithItems(CollectionItem item) {
            return dbClient.sql(ITEM_INSERT_QUERY)
                        .filter(s -> s.returnGeneratedValues("id"))
                                    .bind("price", item.price())
                                    .bind("name", item.name())
                                    .bind("summary", item.summary())
                                    .bind("description", item.description())
                                    .bind("creation_year", item.year())
                                    .bind("country", item.country())
                                    .bind("topics", item.topics())
                                    .fetch()
```

```
                            .first()
                            .map(r -> (Long) r.get("id"));
      }

      default Mono<Long> update(CollectionItem item) {
            return dbClient.sql(ITEM_UPDATE_QUERY)
                        .bind("id", item.id())
                        .bind("price", item.price())
                        .bind("name", item.name())
                        .bind("summary", item.summary())
                        .bind("description", item.description())
                        .bind("creation_year", item.year())
                        .bind("country", item.country())
                        .bind("topics", item.topics())
                        .fetch()
                        .rowsUpdated();
      }
}
```

We can make our application more flexible and extensible introducing the service layer, which encapsulates all details of retrieving data from the persistence layer.

Listing 1.12 CollectionItemService class.java – encapsulates communication with the persistence layer, adds business logic

```
package com.example.service;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.model.CollectionItem;
import com.example.repository.CollectionItemRepository;
import com.example.repository.CustomizedCollectionItemRepository;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class CollectionItemService {

      private final CollectionItemRepository repository;

      public CollectionItemService(CollectionItemRepository repository) {
            this.repository = repository;
      }

      public Flux<CollectionItem> getSorted(String sortBy) {
            return repository.findAll(sortBy);
      }

      public Flux<CollectionItem> getByCountry(String country) {
```

```
            return repository.findByCountry(country);
      }

      public Flux<CollectionItem> getByTopic(String topic) {
            return repository.findByTopic(topic);
      }

      public Flux<CollectionItem> getByYears(short fromYear, short toYear) {
            return repository.findByYearInterval(fromYear, toYear);
      }

      public Mono<CollectionItem> getById(Long id) {
            return repository.getById(id);
      }

      @Transactional
      public Mono<CollectionItem> add(CollectionItem item) {
            return
((CustomizedCollectionItemRepository)repository).saveWithImages(item)
                        .flatMap(inserted -> {
                              return repository.getById(inserted.id())
                                    .flatMap(i -> {
                                          return Mono.just(i);
                                    });
                        });
      }

      @Transactional
      public Mono<CollectionItem> update(CollectionItem item) {
            return ((CustomizedCollectionItemRepository)repository).update(item)
                        .flatMap(n -> {
                              return repository.getById(item.id())
                                    .flatMap(i -> {
                                         return Mono.just(i);
                                    });
                        });
      }

      public Mono<Void> delete(Long id) {
            return repository.deleteById(id);
      }
}
```

To access the collection data from outside of our application, we expose the corresponding REST endpoints in CollectionItemController, using the CollectionItemService component.

Listing 1.13 CollectionItemController.java – exposing CollectionItem objects via REST endpoints

```
package com.example.api;
```

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.model.CollectionItem;
import com.example.service.CollectionItemService;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping("/collection/items")
public class CollectionItemController {

	private final CollectionItemService service;

	public CollectionItemController(CollectionItemService service) {
		this.service = service;
	}

	@GetMapping("/")
	public Flux<CollectionItem> getRows(String sortBy) {
		return service.getSorted(sortBy);
	}

	@GetMapping("/{id}")
	public Mono<CollectionItem> getById(@PathVariable("id") Long id) {
		return service.getById(id);
	}

	@GetMapping("/country/{country}")
	public Flux<CollectionItem> getByCountry(@PathVariable("country") String 
country) {
		return service.getByCountry(country);
	}

	@GetMapping("/topic/{topic}")
	public Flux<CollectionItem> getByTopic(@PathVariable("topic") String topic) 
{
		return service.getByTopic(topic);
	}

	@GetMapping("/fromyear/{fromyear}/toyear/{toyear}")
	public Flux<CollectionItem> getByYears(@PathVariable("fromyear") short 
fromYear, @PathVariable("toyear") short toYear) {
		return service.getByYears(fromYear, toYear);
```

```
      }

      @PostMapping("/add")
    public Mono<CollectionItem> add(@RequestBody CollectionItem item) {
        return service.add(item);
    }

      @PutMapping("/update/{id}")
    public Mono<CollectionItem> update(@RequestBody CollectionItem item, 
@PathVariable("id") Long id) {
        return service.update(item);
    }

      @DeleteMapping("/delete/{id}")
      @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> delete(@PathVariable("id") Long id) {
        return service.delete(id);
    }
}
```

We can add the following tests to check functionality of our application.

Listing 1.14 CollectionItemRepository integration test

```
package com.example.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import java.math.BigDecimal;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.r2dbc.DataR2dbcTest;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.annotation.DirtiesContext.ClassMode;
import com.example.model.CollectionItem;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@DirtiesContext(classMode = ClassMode.BEFORE_EACH_TEST_METHOD)
@DataR2dbcTest
public class CollectionItemRepositoryTests {

  @Autowired
  CollectionItemRepository collectionItemRepository;

    @Test
      public void testFindByCountry() {
            Flux<CollectionItem> result = 
collectionItemRepository.findByCountry("uk");
            assertNotNull(result);
            assertEquals(1, result.count().block());
            CollectionItem first = result.blockFirst();
            assertEquals("The Penny Black", first.name());
```

```
            assertEquals("history", first.topics()[0]);
      }

      @Test
      public void testFindByYearInterval() {
            Flux<CollectionItem> result =
collectionItemRepository.findByYearInterval(1993, 2000);
            assertNotNull(result);
            assertEquals(2, result.count().block());
            CollectionItem first = result.blockFirst();
            assertEquals("Juke", first.name());
          assertEquals("The Juke stature", first.summary());
          assertEquals(new BigDecimal("1000000.00"), first.price());
          assertEquals("arts", first.topics()[0]);
          //Check the item images
          assertEquals("juke.jpg", first.image().fileName());
          assertEquals("juke-small.jpg", first.smallImage().fileName());
      }

      @Test
      public void testFindByTopic() {
            Flux<CollectionItem> result =
collectionItemRepository.findByTopic("Arts");
            assertNotNull(result);
            assertEquals(2, result.count().block());
            CollectionItem first = result.blockFirst();
            assertEquals("Juke", first.name());
          assertEquals("The Juke stature", first.summary());
          assertEquals(new BigDecimal("1000000.00"), first.price());
          assertEquals("arts", first.topics()[0]);
      }

      @Test
      public void testFindById() {
            Mono<CollectionItem> result = collectionItemRepository.getById(2L);
            assertNotNull(result);
            CollectionItem testItem = result.block();
            assertEquals("Juke", testItem.name());
          assertEquals("The Juke stature", testItem.summary());
          assertEquals(new BigDecimal("1000000.00"), testItem.price());
          assertEquals("arts", testItem.topics()[0]);
          assertEquals("programming", testItem.topics()[1]);
      }

      @Test
      public void testAddNewItem() {
            String[] topics = {"programming"};
            CollectionItem newItem = CollectionItem.of(0, new
BigDecimal("10000.00"), "The Flying Tux", "Test summary", "Test description",
(short) 1991, "fi", topics);
            Mono<CollectionItem> result =
collectionItemRepository.saveWithImages(newItem);
            assertNotNull(result);
            CollectionItem addedItem = result.block();
```

```
            assertEquals("The Flying Tux", addedItem.name());
          assertEquals("Test summary", addedItem.summary());
          assertEquals("Test description", addedItem.description());
          assertEquals("fi", addedItem.country());
          assertEquals(new BigDecimal("10000.00"), addedItem.price());
          assertEquals("programming", addedItem.topics()[0]);
      }

      @Test
      public void testUpdateItem() {
            String[] updatedTopics = {"arts","programming","software"};
        CollectionItem testItem = collectionItemRepository.getById(2L).block();
            Mono<CollectionItem> result = collectionItemRepository.update(
                  CollectionItem.of(2L, new BigDecimal("2000000.00"),
testItem.image(), testItem.smallImage(), "Juke Forever!", testItem.summary(),
testItem.description(), testItem.year(), testItem.country(), updatedTopics)
            ).flatMap(n -> {
                              return
collectionItemRepository.getById(2L).flatMap(i -> {
                                    return Mono.just(i);
                              });
            });
            assertNotNull(result);
            CollectionItem updatedItem = result.block();
            assertEquals("Juke Forever!", updatedItem.name());
          assertEquals("The Juke stature", updatedItem.summary());
          assertEquals(new BigDecimal("2000000.00"), updatedItem.price());
          assertEquals("arts", updatedItem.topics()[0]);
          assertEquals("programming", updatedItem.topics()[1]);
          assertEquals("software", updatedItem.topics()[2]);
      }
}
```

Listing 1.15 The REST API test

```
package com.example.api;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

import java.math.BigDecimal;
import java.util.List;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.HttpHeaders;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.reactive.server.WebTestClient;

import com.example.model.CollectionItem;
import com.example.service.CollectionItemService;
```

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@WebFluxTest(CollectionItemController.class)
public class CollectionItemEndpointTests {

      @Autowired
      private WebTestClient webClient;

      @MockitoBean
      private CollectionItemService collectionItemService;

      @Test
      void testGetByCountry() {
        CollectionItem testItem = CollectionItem
         .of(0, BigDecimal.valueOf(1000), null, null, "The Penny Black",
             "The very first post stamp.", null, null, null, null);
        Flux<CollectionItem> testResult = Flux.fromIterable(List.of(testItem));
        given(collectionItemService.getByCountry("uk")).willReturn(testResult);
        webClient.get()
            .uri("/collection/items/country/uk")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBodyList(CollectionItem.class)
            .value(result -> {
              assertThat(result).hasSize(1);
              assertThat(result).element(0)
                .hasFieldOrPropertyWithValue("name", "The Penny Black")
                .hasFieldOrPropertyWithValue("price", BigDecimal.valueOf(1000))
                .hasFieldOrPropertyWithValue("summary",
                                             "The very first post stamp.");
            });
            //Try to filter by non-exisiting country
            webClient.get()
            .uri("/collection/items/country/xx")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBodyList(CollectionItem.class)
            .value(result -> {
                  assertThat(result).hasSize(0);
            });
      }

      @Test
      void testGetById() {
            //Set up the test response
            CollectionItem testItem = CollectionItem
             .of(1, BigDecimal.valueOf(1000), null, null, "The Penny Black",
                 "The very first post stamp.", null, null, null, null);
            Mono<CollectionItem> testResult = Mono.just(testItem);
            given(collectionItemService.getById(1L)).willReturn(testResult);
            //Call the end point
```

```
            webClient.get()
            .uri("/collection/items/1")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(CollectionItem.class)
            .value(result -> {
               assertThat(result)
                .hasFieldOrPropertyWithValue("name", "The Penny Black")
                .hasFieldOrPropertyWithValue("price", BigDecimal.valueOf(1000))
                .hasFieldOrPropertyWithValue("summary",
                                             "The very first post stamp.");
            });
            //Try to find by non-exisiting ID
            webClient.get()
            .uri("/collection/items/0")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(CollectionItem.class)
            .value(result -> {
                  assertThat(result).isNull();
            });
      }


      @Test
      void testGetByTopic() {
            //Test implementation ...
      }

      @Test
      void testGetByYears() {
            //Test implementation ...
      }
}
```

We build the project with the following Maven command, either from `Run As ...→ Maven build …` `STS` drop down menu or from the terminal window:

```
$ mvn clean package
```

The result should be similar to the following output:

```
. . .

[INFO] Results:
[INFO]
[INFO] [1;32mTests run: 11, Failures: 0, Errors: 0, Skipped: 0[m
[INFO]
[INFO]
[INFO] [1m--- [0;32mjar:3.4.2:jar[m [1m(default-jar)[m @ [36mcollection-
reactive[0;1m ---[m
```

```
[INFO] Building jar: /home/sergiy/Documents/workspace-spring-tools-for-eclipse-4.30.0.RELEASE/CollectionApplication/target/collection-reactive-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] [1m--- [0;32mspring-boot:3.5.7:repackage[m [1m(repackage)[m @ [36mcollection-reactive[0;1m ---[m
[INFO] Replacing main artifact /home/sergiy/Documents/workspace-spring-tools-for-eclipse-4.30.0.RELEASE/CollectionApplication/target/collection-reactive-0.0.1-SNAPSHOT.jar with repackaged archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to /home/sergiy/Documents/workspace-spring-tools-for-eclipse-4.30.0.RELEASE/CollectionApplication/target/collection-reactive-0.0.1-SNAPSHOT.jar.original
[INFO] [1m------------------------------------------------------------------------[m
[INFO] [1;32mBUILD SUCCESS[m
[INFO] [1m------------------------------------------------------------------------[m
[INFO] Total time:  12.643 s
[INFO] Finished at: 2026-03-03T18:24:35+01:00
[INFO] [1m------------------------------------------------------------------------
```

Now we can run our application with the following maven command:

```
$ mvn spring-boot:run
```

or we can place the generated .jar file in place of our choice and run it as a common thick jar:

```
$ java -jar collection-reactive-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.5.7)

2025-11-18T17:28:24.057+01:00 INFO 3743 --- [CollectionApplication] [           main] com.example.CollectionApplication        : Starting CollectionApplication v0.0.1-SNAPSHOT using Java 21.0.9 with PID 3743 (/home/pi/Bookshelf/EnterpriseDevelopmentWithRPi/SourceCode/Chapter_1/CollectionApplication_Keykloak/target/collection-reactive-0.0.1-SNAPSHOT.jar started by pi in /home/pi/Bookshelf/EnterpriseDevelopmentWithRPi/SourceCode/Chapter_1/CollectionApplication_Keykloak/target)
2025-11-18T17:28:24.066+01:00 INFO 3743 --- [CollectionApplication] [           main] com.example.CollectionApplication        : No active profile set, falling back to 1 default profile: "default"
```

```
2025-11-18T17:28:25.252+01:00        INFO    3743    ---    [CollectionApplication]
[            main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring
Data R2DBC repositories in DEFAULT mode.
2025-11-18T17:28:25.543+01:00        INFO    3743    ---    [CollectionApplication]
[            main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data
repository scanning in 274 ms. Found 2 R2DBC repository interfaces.
2025-11-18T17:28:26.734+01:00        INFO    3743    ---    [CollectionApplication]
[            main] .j.s.R2dbcRelationshipsAutoConfiguration : R2DBC Relationships
auto-configuration loaded.
2025-11-18T17:28:28.368+01:00        INFO    3743    ---    [CollectionApplication]
[            main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port
8888 (http)
2025-11-18T17:28:28.410+01:00        INFO    3743    ---    [CollectionApplication]
[                    main]  com.example.CollectionApplication             :  Started
CollectionApplication in 5.129 seconds (process running for 6.018)
```

Now our Collection application provides a full-scale enterprise service, including the persistence layer and REST API. It is reactive and optimized for Cloud deployment. But it is still missing an important feature, which is critical for most of enterprise projects. It is not secured, that is, it does not support authentication and authorization when accessing its endpoints. In the next section, we will add such a security support with the usage of Keycloak technology.

1.3 Securing applications and services with Keycloak

As it is mentioned in Wikipedia, Keycloak is an open-source software product to allow single sign-on with identity and access management aimed at modern applications and services. We can run it as a Docker image or as a stand-alone service. In our projects, we are going to run the Keycloak distribution locally as a stand-alone service, to check possibilities for running Keycloak in Raspberry Pi environment.

In real projects, Keycloak authorization server is usually run on a dedicated host, either internally, inside a corporative Intranet, or in a shared location, in the Cloud.

We can download the latest stable version from the Keycloak official site. Provided having the distribution archive downloaded, we can extract its content into a directory of our choice and start Keycloak service with the kc.sh script, for example:

```
pi@raspberrypi:~/Tools/keycloak-26.4.5 $ bin/kc.sh start-dev --http-port=9090
```

Then we access the administration console on 9090 port, as it is shown in figure 1.8.

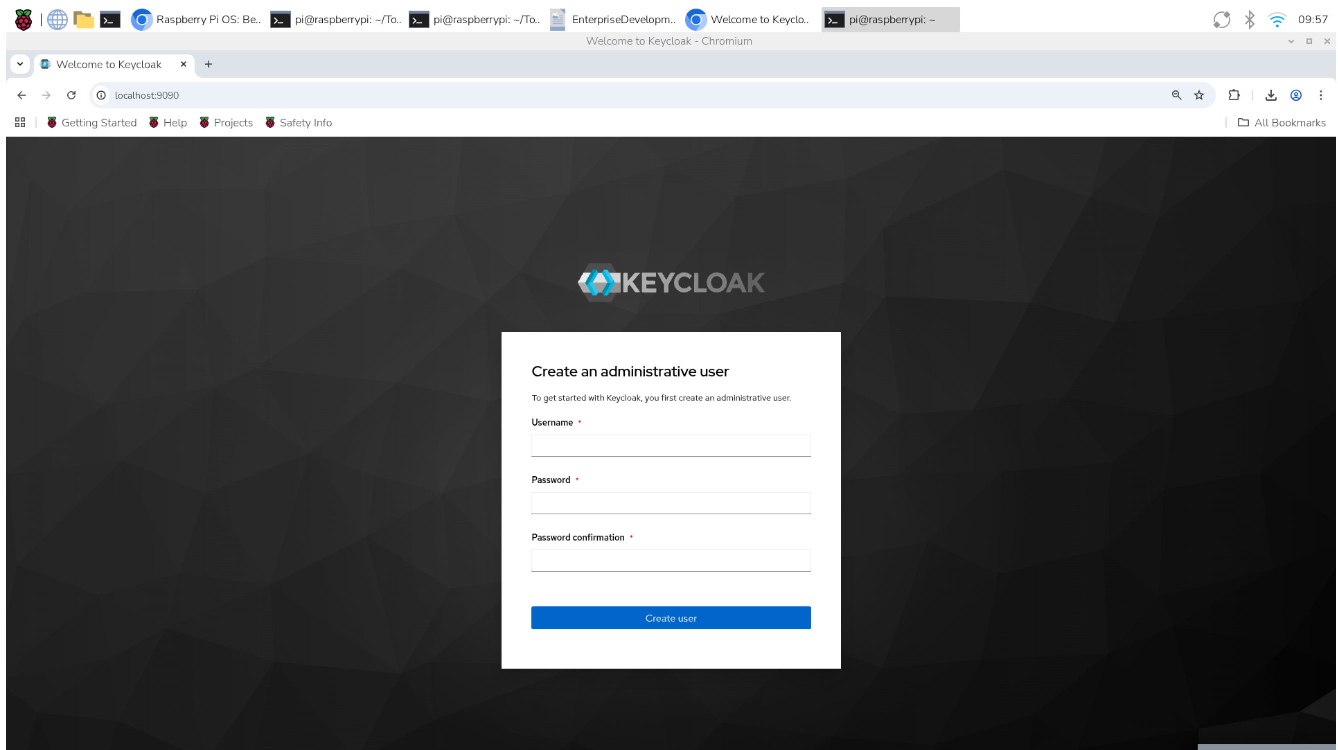


Fig. 1.9 Keycloak welcome screen

In production mode, Keycloak authorization server should communicate with its clients via SSL connection, with the usage of trusted certificates. For testing purposes, we can equip our Keycloak instance with a self-signed certificate. One way to do it is to use the following commands:

```
1. sudo keytool -genkeypair -alias keycloak -keyalg RSA -keysize 2048 -validity 365
-keystore server.keystore -dname "cn=localhost,o=MyFirm,c=CZ" -ext
"SAN=IP:192.168.49.1" -keypass secret -storepass secret
```

Put the generated server.keystore file into `<keycloak-installation-dir>/conf` directory. Then we can start the Keycloak server in "production" mode:

```
pi@raspberrypi:<keycloak-installation-dir> $ ./bin/kc.sh start --http-enabled=false
--https-key-store-password=secret --verbose
```

2. Now we can request the Keycloak certificate:

```
pi@raspberrypi:~ $ openssl s_client -connect localhost:8443 -showcerts | openssl
x509 -outform PEM > keycloak-cert.crt;
```

3. To make the Spring Boot engine use the certificate while communicating with the Keycloak server, we need to add the certificate to the Java key store:

```
pi@raspberrypi:~ $ sudo keytool -import -alias keycloak-cert -file keycloak-
cert.crt -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit
```

Keycloak provides comprehensive support for implementing OAuth2 authentication and authorization flows and fine adjustment of secured access to applications and services in various environments. We are going to implement the client credentials security flow in Collection application.
In this case, the application works as a resource server, which uses the Keycloak authorization server for authentication and authorization of requests to the REST API exposed in CollectionController.java. The implementation includes the client configuring on the Keycloak server and the Collection application enhancement and adjustment.

1.3.1 Keycloak configuring

To enable a client credential security flow, we need perform the following steps on the Keycloak server:

1. Create a special realm, collection, for our application (optional but can be really useful);
2. Within the realm, create a client with credentials and assign a predefined scope to the client, enabling the client authentication and authorization, as shown in figures 1.9 and 1.10;

Clients, users, and other resources can be imported into Keycloak, using .json descriptors like that shown in listing 1.16. We can load them at Realm settings → Actions → Partial import.

Listing 1.16 collection-client.json – a minimal client configuration to import into Keycloak realm

```
{"realm": "collection",

  "clients": [{
      "clientId": "collection-client",
      "enabled": true,
      "clientAuthenticatorType": "client-secret",
      "secret": "development",
      "standardFlowEnabled": false,
      "implicitFlowEnabled": false,
      "directAccessGrantsEnabled": false,
      "serviceAccountsEnabled": true,
      "publicClient": false,
      "protocol": "openid-connect",
```

```
      "defaultClientScopes": [
        "collection"
      ],
      "optionalClientScopes": [
      ]
    }],
    "users": [
    {
      "username": "service-account-collection-client",
      "enabled": true,
      "serviceAccountClientId": "collection-client",
      "realmRoles": [
        "default-roles-collection"
      ]
    }
  ]
}
```

For testing purposes, it is often convenient to configure long-lived access tokens, at Realm settings → Tokens → Access Token Lifespan. But, in a production system, access tokens should expire within a few minutes!

1.3.2 Application security enhancement

In our security schema, the Collection application is supposed to work as a resource server. To implement this functionality and enable Spring Boot security context, we add necessary dependencies to the application pom.xml descriptor, as shown in listing 1.17.

Listing 1.17 Keycloak Maven dependencies

```
. . .
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
. . .
```

Also, we create a security configuration bean implementing the authentication and authorization mechanism, as shown in listing 1.18.

Listing 1.18 Application security configuration

```
package com.example.security;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.oauth2.jwt.NimbusReactiveJwtDecoder;
import org.springframework.security.oauth2.jwt.ReactiveJwtDecoder;
import org.springframework.security.web.server.SecurityWebFilterChain;

@Configuration
public class SecurityConfig {

    @Value("${spring.security.oauth2.resourceserver.jwt.jwkSetUri}")
    private String jwkEndpoint;

    @Bean
    SecurityWebFilterChain springSecurity(ServerHttpSecurity http) {
      http
            .authorizeExchange((exchange) -> exchange
                  .pathMatchers("/api/**").hasAuthority("SCOPE_collection")
                  .anyExchange().authenticated())
                  .oauth2ResourceServer(oauth2 ->
oauth2.jwt(Customizer.withDefaults()));
      return http.build();
    }

    @Bean
    ReactiveJwtDecoder jwtDecoder() {
        return NimbusReactiveJwtDecoder.withJwkSetUri(jwkEndpoint).build();
    }
}
```

We can test the secured API endpoints with the following test:

Listing 1.19 Testing the Collection application secured API

```
package com.example.api;

import static org.mockito.BDDMockito.given;
import static
org.springframework.security.test.web.reactive.server.SecurityMockServerConfigurer
s.mockJwt;

import java.math.BigDecimal;
import java.util.List;
```

```
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.webflux.test.autoconfigure.WebFluxTest;
import org.springframework.context.annotation.Import;
import org.springframework.http.HttpHeaders;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.reactive.server.WebTestClient;

import com.example.model.CollectionItem;
import com.example.security.SecurityConfig;
import com.example.service.CollectionItemService;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@WebFluxTest(CollectionItemController.class)
@Import(SecurityConfig.class)
public class CollectionItemEndpointTests {

	@Autowired
	private WebTestClient webClient;

	@MockitoBean
	private CollectionItemService collectionItemService;

	@Test
	void testGetByCountry() {
		CollectionItem testItem = CollectionItem.of(0,
BigDecimal.valueOf(1000), null, null, "The Penny Black", "The very first post
stamp.", null, null, null, null);
		Flux<CollectionItem> testResult =
Flux.fromIterable(List.of(testItem));

	given(collectionItemService.getByCountry("uk")).willReturn(testResult);
		webClient
		.mutateWith(mockJwt().jwt((jwt) -> jwt.claim("scope", "collection")))
		.get()
		.uri("/api/collection/items/country/uk")
		.header(HttpHeaders.ACCEPT, "application/json")
		.exchange()
		.expectStatus().is2xxSuccessful()
		.expectBodyList(CollectionItem.class)
		.value(result -> {
			Assertions.assertThat(result).hasSize(1);
			Assertions.assertThat(result).element(0)
				.hasFieldOrPropertyWithValue("name", "The Penny Black")
				.hasFieldOrPropertyWithValue("price",
BigDecimal.valueOf(1000))
				.hasFieldOrPropertyWithValue("summary", "The very first
post stamp.");
		});
		//Try to filter by non-exisiting country
```

```
            webClient
            .mutateWith(mockJwt().jwt((jwt) -> jwt.claim("scope", "collection")))
            .get()
            .uri("/api/collection/items/country/xx")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBodyList(CollectionItem.class)
            .value(result -> {
                  Assertions.assertThat(result).hasSize(0);
            });
      }

      @Test
      void testGetById() {
            //Set up the test response
            CollectionItem testItem = CollectionItem.of(1,
BigDecimal.valueOf(1000), null, null, "The Penny Black", "The very first post
stamp.", null, null, null, null);
            Mono<CollectionItem> testResult = Mono.just(testItem);
            given(collectionItemService.getById(1L)).willReturn(testResult);
            //Call the end point
            webClient
            .mutateWith(mockJwt().jwt((jwt) -> jwt.claim("scope", "collection")))
            .get()
            .uri("/api/collection/items/1")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(CollectionItem.class)
            .value(result -> {
                  Assertions.assertThat(result)
                        .hasFieldOrPropertyWithValue("name", "The Penny Black")
                        .hasFieldOrPropertyWithValue("price",
BigDecimal.valueOf(1000))
                        .hasFieldOrPropertyWithValue("summary", "The very first
post stamp.");
            });
            //Try to find by non-exisiting ID
            webClient
            .mutateWith(mockJwt().jwt((jwt) -> jwt.claim("scope", "collection")))
            .get()
            .uri("/api/collection/items/0")
            .header(HttpHeaders.ACCEPT, "application/json")
            .exchange()
            .expectStatus().is2xxSuccessful()
            .expectBody(CollectionItem.class)
            .value(result -> {
                  Assertions.assertThat(result).isNull();
            });
      }

      @Test
```

```
    void testGetByTopic() {
        //Test implementation ...
    }

    @Test
    void testGetByYears() {
        //Test implementation ...
    }
}
```

The server access parameters are set in `application.properties` like follows:

```
. . .
spring.security.oauth2.resourceserver.jwt.jwkSetUri=https://localhost:8443/realms/
collection/protocol/openid-connect/certs
. . .
```

Now we can request an access token for the client:

```
pi@raspberrypi:~ $curl -k -POST
"https://localhost:8443/realms/collection/protocol/openid-connect/token" -H
"Content-Type: application/x-www-form-urlencoded" -d
"grant_type=client_credentials" -d "scope=collection" -d "client_id=collection-
client" -d "client_secret=development"
```

If the response looks like the following:

```
{"access_token":"eyJhbGciOiJSUzI1NiIs...4uzcCyxQ","expires_in":60,"refresh_expires_
in":0,"token_type":"Bearer","not-before-policy":0,"scope":"collection"}
```

it means that the client is configured correctly and can be used by our application.

Then we can use the token to access the application secured REST endpoints:

```
$ curl -k -H "Authorization: Bearer eyJhbGciO . . . ETxwAA"
http://localhost:8080/api/collection/items/country/uk
```

Provided having all the stuff configured correctly, the response should like that:

```
[{"id":1,"price":1000.00,"smallImage":{"id":3,"fileName":"penny-black-
small.png"},"image":{"id":4,"fileName":"penny-black.png"},"name":"The Penny
Black","summary":"The very first stamp","description":"The very first post stamp
but surprisingly not the most expensive one","year":1840,"country":"uk","topics":
["history"]}]
$
```

To get the result in more readable format, we can pipe the response to jq command:

```
$ curl -k -H "Authorization: Bearer ..."
http://localhost:8080/api/collection/items/country/uk | jq
[
  {
    "id": 1,
    "price": 1000.00,
    "smallImage": {
      "id": 3,
      "fileName": "penny-black-small.png"
    },
    "image": {
      "id": 4,
      "fileName": "penny-black.png"
    },
    "name": "The Penny Black",
    "summary": "The very first stamp",
    "description": "The very first post stamp but surprisingly not the most
expensive one",
    "year": 1840,
    "country": "uk",
    "topics": [
      "history"
    ]
  }
]
```

In Raspberry Pi OS, we can install the jq tool with the usage of APT package manager:

```
$ sudo apt install jq
```

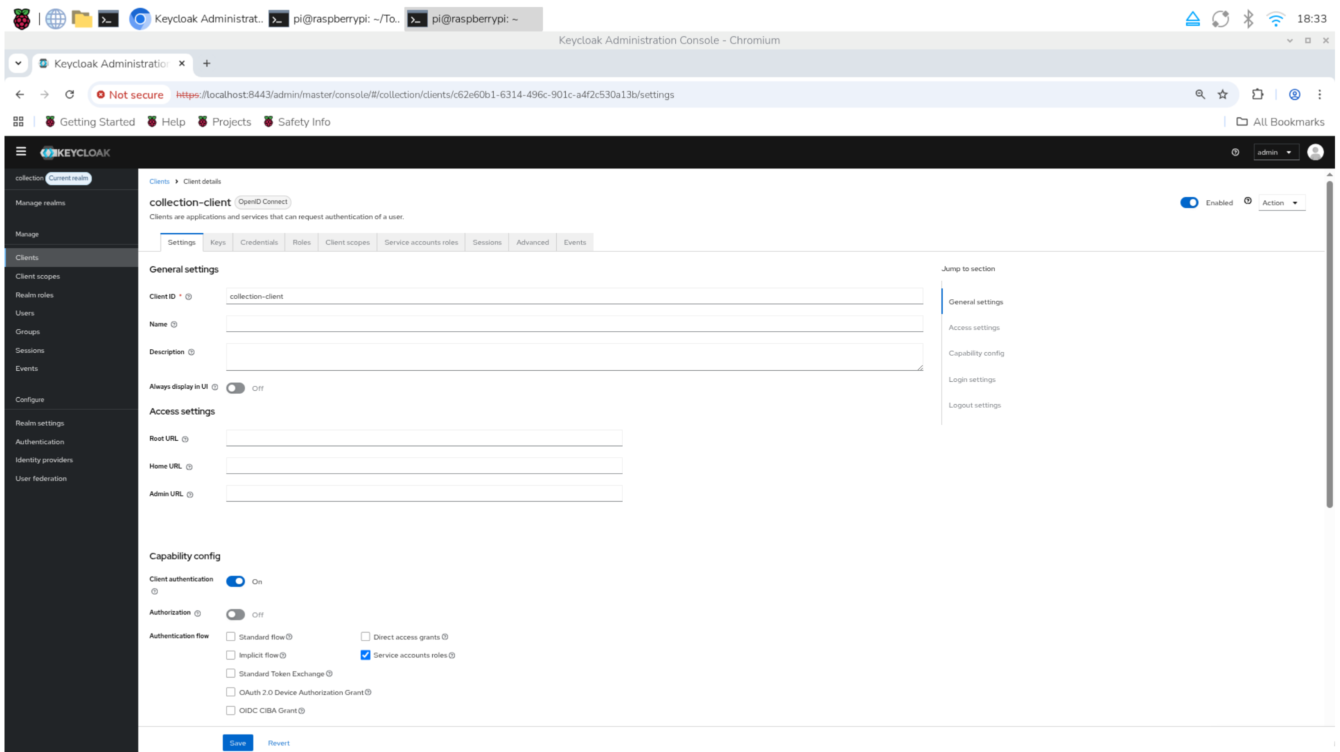


Figure 1.10 Creating the Collection application client

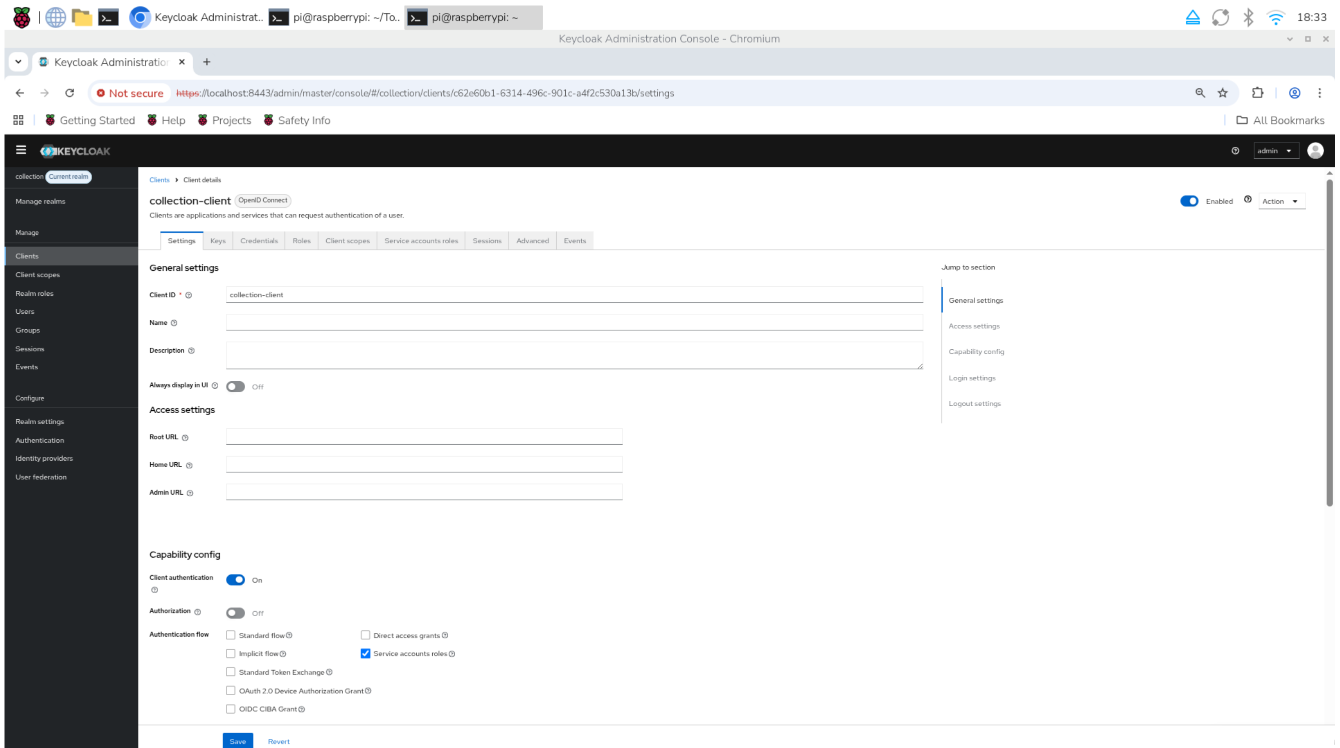


Figure 1.11 Creating collection client scope

Now our application provides CRUD and advanced search functionality for the collection domain objects. This functionality is exposed via a set of secured REST endpoints. Eventually, we would want to add more services, like analytical and marketing ones, as well as adapters to communicate with third-part services and tools. To make such a system more dynamic and flexible, we can use messaging and data streaming and we would like to have is a single centralized system that allows for publishing generic types of data, which will grow as your business grows.. One of options in this case is to use Apache Kafka, which is an open-source distributed event streaming platform for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.
In the next section, we discuss possibilities of using Kafka technology on the Raspberry Pi platform.

1.4 Event streaming with Apache Kafka

For testing purposes, we are going to use a standalone Kafka server, which provides basic data streaming functionality. Then, if necessary, we can extend this system with additional clusters and data centers.

Another common option for enterprise applications is to run Kafka in the Cloud. This way, we can use the cloud infrastructure to achieve the amount of data retention and performance required for our messaging service. Main cloud platforms provide support for using Kafka, like Amazon Managed Streaming for Apache Kafka (Amazon MSK), Google Cloud Managed Service, and Azure Event Hub.

We can download the latest Kafka distribution as a tar archive and extract it to a directory of our choice, for example:

```
/home/Tools$ tar -xzf kafka_2.13-4.2.0.tgz
/home/Tools$ cd kafka_2.13-4.2.0
/home/Tools/kafka_2.13-4.2.0$
```

Then we generate a cluster ID:

```
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-storage.sh random-uuid
```

Copy the UUID — you’ll need it in the next step.

```
The configuration file is available at
<KAFKA-INSTALLATION-DIR>/config/server.properties
```

The next step is to format the storage directory. This initializes the Raft metadata log. Kafka will not start without it.

```
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-storage.sh format --standalone -t <your-
cluster-id> -c config/server.properties
Bootstrap metadata:
BootstrapMetadata(records=[ApiMessageAndVersion(FeatureLevelRecord(name='metadata.v
ersion', featureLevel=29) at version 0),
ApiMessageAndVersion(FeatureLevelRecord(name='eligible.leader.replicas.version',
featureLevel=1) at version 0),
ApiMessageAndVersion(FeatureLevelRecord(name='group.version', featureLevel=1) at
version 0), ApiMessageAndVersion(FeatureLevelRecord(name='share.version',
featureLevel=1) at version 0),
ApiMessageAndVersion(FeatureLevelRecord(name='streams.version', featureLevel=1) at
version 0), ApiMessageAndVersion(FeatureLevelRecord(name='transaction.version',
featureLevel=2) at version 0)], metadataVersionLevel=29, source=format command)
Formatting dynamic metadata voter directory /tmp/kraft-combined-logs with
metadata.version 4.2-IV1.
```

Now we can start Kafka in KRaft mode:

```
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-server-start.sh config/server.properties
```

Then we get a long comprehensive logging out put, which would be ended with the following lines:

```
[2026-02-17 19:30:10,058] INFO [BrokerServer id=1] Transition from STARTING to
STARTED (kafka.server.BrokerServer)
[2026-02-17 19:30:10,061] INFO Kafka version: 4.2.0
(org.apache.kafka.common.utils.AppInfoParser)
[2026-02-17 19:30:10,061] INFO Kafka commitId: a18251bae0b825c6
(org.apache.kafka.common.utils.AppInfoParser)
[2026-02-17 19:30:10,061] INFO Kafka startTimeMs: 1771353010058
(org.apache.kafka.common.utils.AppInfoParser)
[2026-02-17 19:30:10,065] INFO [KafkaRaftServer nodeId=1] Kafka Server started
(kafka.server.KafkaRaftServer)
```

If it is the case, it means that the Kafka cluster is up and running. To test the cluster operation, we can create a topic:

```
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-topics.sh --create --topic test-topic \
  --partitions 1 \
  --replication-factor 1   --bootstrap-server localhost:9092
Created topic test-topic.
```

We start the console producer :

```
bin/kafka-console-producer.sh \
  --topic test-topic \
  --bootstrap-server localhost:9092
```

```
>
```

and the console consumer:

```
bin/kafka-console-consumer.sh \
  --topic test-topic \
  --from-beginning \
  --bootstrap-server localhost:9092
```

If we can enter a message in the producer window and see it in the consumer window, it means that the Kafka service works normally. By default, it supports various alphabets, so our communication can look like that:

```
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-console-producer.sh \
  --topic test-topic \
  --bootstrap-server localhost:9092
>Hello!
>Привіт!
>¡Hola!
>こんにちは！
>
/home/Tools/kafka_2.13-4.2.0$ bin/kafka-console-consumer.sh \
  --topic test-topic \
  --from-beginning \
  --bootstrap-server localhost:9092
Hello!
Привіт!
¡Hola!
こんにちは！
```

Now we can enhance the collection application with data streaming functionality. Suppose we want to notify all interested subscribers about adding new items to our collection. We enhance the Collection service with functionality to publish the corresponding messages and let the Kafka broker to sent the messages to all registered consumers. We can utilize the Spring Boot support for Kafka messaging, which is implemented in the form of KafkaTemplate component.

First of all, we add necessary Maven dependencies:

```
<dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
</dependency>
. . .
<dependency>
      <groupId>org.springframework.kafka</groupId>
```

```
      <artifactId>spring-kafka-test</artifactId>
      <scope>test</scope>
</dependency>
...
```

Then we create a configuration bean, which includes settings for sending data to the Kafka broker, KafkaProducerService.java and JsonSerializer.java classes, which implement the sending mechanism.

Listing 1.20 Configuration settings for the Kafka messaging

```
package com.example.service.kafka;

import java.util.HashMap;
import java.util.Map;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;

import com.example.model.CollectionItem;
import com.example.model.Image;

@Configuration
public class KafkaProducerConfig {


    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;

    @Bean
    ProducerFactory<String, CollectionItem> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    Map<String, Object> producerProps() {
      Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
```

```
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        return props;
    }

    @Bean
    KafkaTemplate<String, CollectionItem> kafkaTemplate() {
        return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(producerProps()));
    }

    @Bean
    KafkaTemplate<String, Image> kafkaImageTemplate() {
        return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(producerProps()));
    }
}
```

Listing 1.21 Sending data to the Kafka broker

```
package com.example.service.kafka;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import com.example.model.CollectionItem;
import com.example.model.Image;

@Service
public class KafkaProducerService {

      private static final Logger LOGGER =
LoggerFactory.getLogger(KafkaProducerService.class);

      @Value("${kafka.producer-topic}")
    private String  topic;
    ;
    private final KafkaTemplate<String, CollectionItem> kafkaTemplate;
    private final KafkaTemplate<String, Image> kafkaImageTemplate;

    public KafkaProducerService(KafkaTemplate<String, CollectionItem>
kafkaTemplate, KafkaTemplate<String, Image> kafkaImageTemplate) {
        this.kafkaTemplate = kafkaTemplate;
            this.kafkaImageTemplate = kafkaImageTemplate;
    }

    public CollectionItem sendItem(CollectionItem item) {
        kafkaTemplate.send(topic, item);
        LOGGER.info("Collection update sent: " + item);
            return item;
    }
```

```
    public Image sendImage(Image image) {
      kafkaImageTemplate.send(topic, image);
        LOGGER.info("Collection update sent: " + image);
            return image;
    }
}
```

Listing 1.22 Domain object serialization for the Kafka messaging

```
package com.example.service.kafka;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.common.serialization.Serializer;

public class JsonSerializer<T> implements Serializer<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public byte[] serialize(String topic, T data) {
        try {
            return objectMapper.writeValueAsBytes(data);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

After that, we can enhance our service beans with the data streaming ability, like it is shown in listing 1.23.

Listing 1.23 CollectionItemService class with the Kafka messaging enabled

```
package com.example.service;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.model.CollectionItem;
import com.example.repository.CollectionItemRepository;
import com.example.repository.CustomizedCollectionItemRepository;
import com.example.service.kafka.KafkaProducerService;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class CollectionItemService {

      private final CollectionItemRepository repository;
      private final KafkaProducerService producerService;
```

```
	public CollectionItemService(CollectionItemRepository repository,
KafkaProducerService producerService) {
		this.repository = repository;
		this.producerService = producerService;
	}

	public Flux<CollectionItem> getSorted(String sortBy) {
		return repository.findAll(sortBy);
	}

	public Flux<CollectionItem> getByCountry(String country) {
		return repository.findByCountry(country);
	}

	public Flux<CollectionItem> getByTopic(String topic) {
		return repository.findByTopic(topic);
	}

	public Flux<CollectionItem> getByYears(short fromYear, short toYear) {
		return repository.findByYearInterval(fromYear, toYear);
	}

	public Mono<CollectionItem> getById(Long id) {
		return repository.getById(id);
	}

	@Transactional
	public Mono<CollectionItem> add(CollectionItem item) {
	    return((CustomizedCollectionItemRepository)repository)
	                .saveWithImages(item).flatMap(inserted -> {
	                    return repository.getById(inserted.id()).flatMap(i -> {
	                                producerService.sendItem(i);
	                                return Mono.just(i);
	                            });
	                    });
	}

	@Transactional
	public Mono<CollectionItem> update(CollectionItem item) {
		return ((CustomizedCollectionItemRepository)repository).update(item)
				.flatMap(n -> {
					return repository.getById(item.id())
					         .flatMap(i -> {
							producerService.sendItem(i);
							return Mono.just(i);
						  });
				});
	}

	public Mono<Void> delete(Long id) {
		return repository.deleteById(id);
	}
}
```

Now we can send a new collection item request to the REST endpoint, like that:

```
curl -H "Content-Type: application/json" --json '{"price":1000.00,"name":"The
Flying Tux","summary":"Watercolor of the flying Tux","description":"An expressive
image of Tux flying in the sky.","year":1991,"country":"fi","topics":
["programming"]}' http://localhost:8888/collection/items/add
```

and see the inserted data in the Kafka consumer console:

```
$ bin/kafka-console-consumer.sh   --topic collection-update   --from-beginning   --
bootstrap-server localhost:9092
{"id":1,"price":1000.00,"smallImage":null,"image":null,"name":"The Flying
Tux","summary":"Watercolor of the flying Tux","description":"An expressive image of
Tux flying in the sky.","year":1991,"country":"fi","topics":["programming"]}
```

Then we can update the newly-created collection item:

```
$ curl -X PUT -H "Content-Type: application/json" --json '{"id": 1,
"price":3000.00,"name":"The Flying Tux","summary":"Watercolor of the flying
Tux","description":"An expressive image of Tux flying in the
sky.","year":1991,"country":"fi","topics":["programming", "ornithology"]}'
http://localhost:8888/collection/items/update/1
```

and see how the corresponding message was added in the topic:

```
. . .
{"id":1,"price":1000.00,"smallImage":null,"image":null,"name":"The Flying
Tux","summary":"Watercolor of the flying Tux","description":"An expressive image of
Tux flying in the sky.","year":1991,"country":"fi","topics":["programming"]}
{"id":1,"price":3000.00,"smallImage":null,"image":null,"name":"The Flying
Tux","summary":"Watercolor of the flying Tux","description":"An expressive image of
Tux flying in the sky.","year":1991,"country":"fi","topics":["programming",
"ornithology"]}
```

Our application notifies all the topic subscribers about adding and modifying collection items. This functionality can be used for the system service orchestration and B2B integration.

In our example application, we use a toy database with 3 items. In real enterprise projects, it is often necessary to process huge amounts of data with high speed and low latency. So, we need to use powerful relational database (RDB) management systems. In the next chapter, we are going to check how to use such systems in the Raspberry Pi environment.