

Practicum 13

CSCI-P 465/565 Software Engineering I

Indiana University Bloomington - Spring 2024

- Today:
 - Communication between microservices
 - Direct messaging with HTTP
 - Indirect messaging with RabbitMQ

RabbitMQ modes

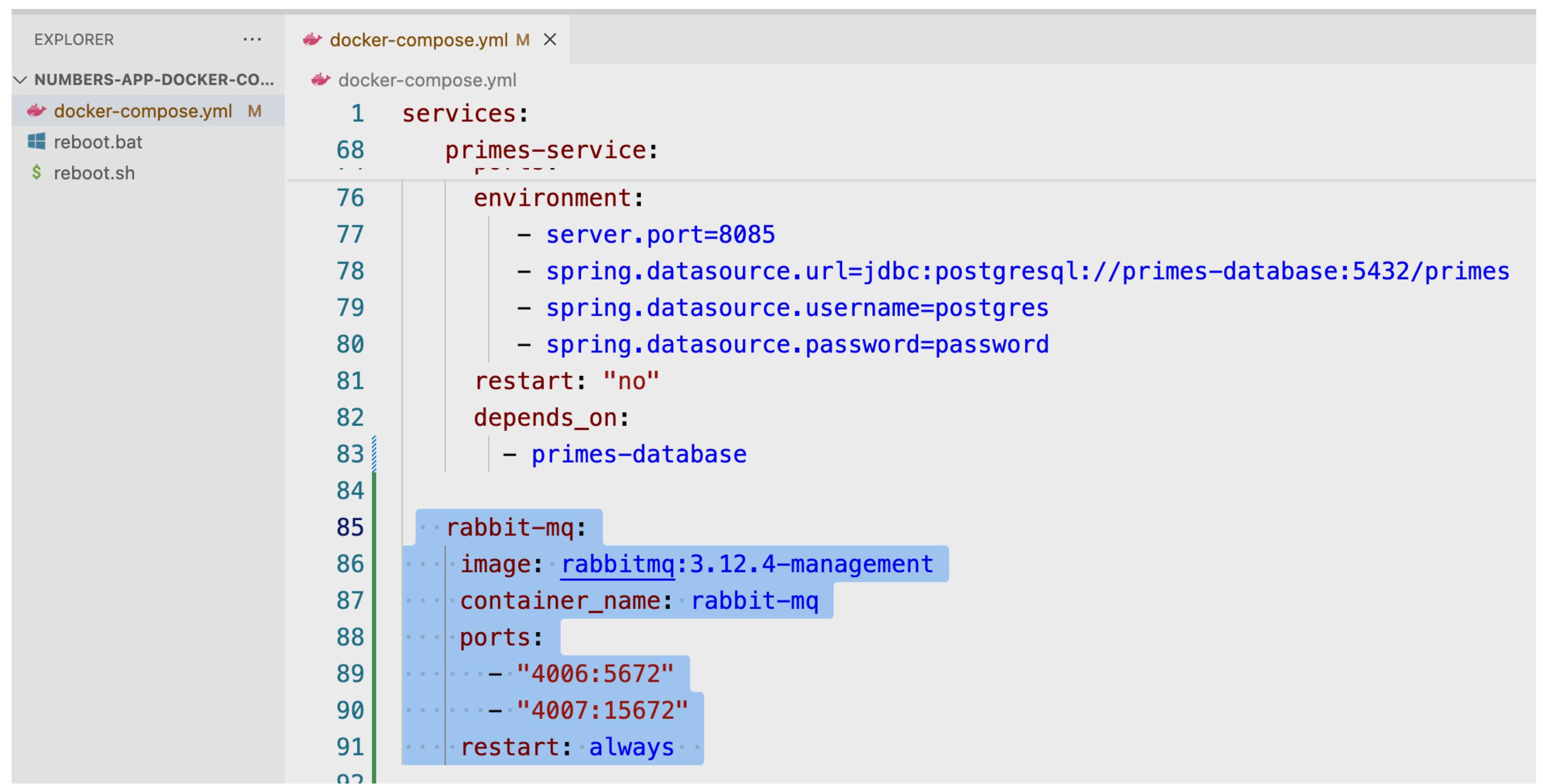
- Direct messaging
- Work queue: distribute tasks among worker nodes.
- Publish/subscribe
- Routing: receive messages selectively
- Topics: receive messages based on a pattern
- Publisher confirm: get a confirmation that the message was processed.

Communication between microservices - RabbitMQ

Add a RabbitMQ container

Steps

- Add a rabbitmq container:



The screenshot shows a code editor with a Docker Compose file open. The file defines two services: 'primes-service' and 'rabbit-mq'. The 'primes-service' is configured to use environment variables for database connection details and depends on the 'primes-database' service. The 'rabbit-mq' service uses the 'rabbitmq:3.12.4-management' image, has a container name of 'rabbit-mq', and exposes ports 4006 and 4007.

```
EXPLORER      ...
NUMBERS-APP-DOCKER-CO...
docker-compose.yml M
reboot.bat
$ reboot.sh

docker-compose.yml M ×
docker-compose.yml

1  services:
68   primes-service:
76     environment:
77       - server.port=8085
78       - spring.datasource.url=jdbc:postgresql://primes-database:5432/primes
79       - spring.datasource.username=postgres
80       - spring.datasource.password=password
81     restart: "no"
82     depends_on:
83       - primes-database
84
85   rabbit-mq:
86     image: rabbitmq:3.12.4-management
87     container_name: rabbit-mq
88     ports:
89       - "4006:5672"
90       - "4007:15672"
91     restart: always
92
```

Steps

- Reboot the docker compose and verify that you can connect to RabbitMQ management tool at <http://localhost:4007>
- The default username is: guest
- The default password is: guest

The screenshot shows two consecutive screenshots of the RabbitMQ Management Tool.

Login Screen: The first screenshot shows the login page at localhost:4007. It features the RabbitMQ logo, fields for 'Username' and 'Password', and a 'Login' button. The URL bar indicates the page is at localhost:4007/.

Overview Page: The second screenshot shows the 'Overview' page after logging in. The top navigation bar includes links for Overview, Connections, Channels, Exchanges, Queues and Streams, and Admin. The page is titled 'RabbitMQ TM' and shows system details: RabbitMQ 3.12.4 | Erlang 25.3.2.6. It displays real-time metrics like 'Queued messages' (last minute), 'Currently idle', and 'Message rates' (last minute). Below these are 'Global counts' and summary statistics for 'Connections', 'Channels', 'Exchanges', 'Queues', and 'Consumers'. The 'Nodes' section lists a single node: **rabbit@063485c3fec3**, providing detailed resource usage statistics. At the bottom, there are links for Churn statistics, Ports and contexts, Export definitions, and Import definitions.

**Modify the primes-service to
send messages to RabbitMQ**

Steps

- In primes-service backend add the dependencies:

```
<java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>

```

Steps

- In primes-service backend add the rabbitmq configuration settings to your application.yml file:

The screenshot shows a Java Spring Boot project structure on the left and the `application.yml` configuration file on the right.

Project Structure:

- java**
 - edu.iu.habahram.primesservice**
 - controller**
 - AuthenticationController
 - HomeController
 - PrimesController
 - model**
 - Customer
 - rabbitmq**
 - MQConfiguration
 - MQSender
 - repository**
 - AuthenticationDBRepository
 - AuthenticationFileRepository
 - IAuthenticationRepository
 - security**
 - Jwks
 - KeyGeneratorUtils
 - SecurityConfig
 - service**
 - AuthenticationService
 - IAuthenticationService
 - IPrimesService
 - PrimesService
 - TokenService
 - PrimesServiceApplication
 - resources**
 - static
 - templates
 - application.yml
 - test
 - target
 - .gitignore

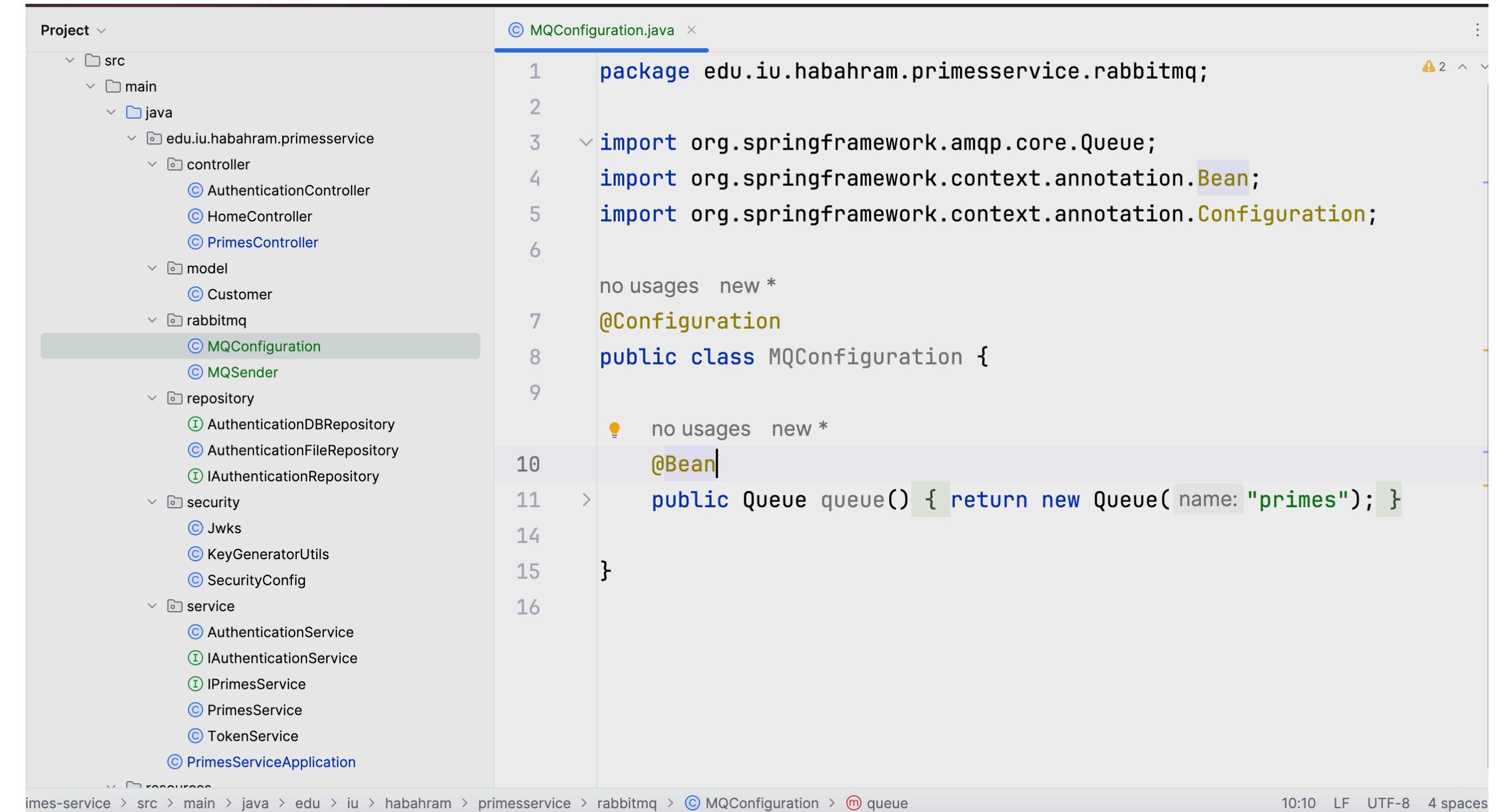
application.yml Content:

```
spring:  
  datasource:  
    url: jdbc:postgresql://localhost:4003/primes  
    username: postgres  
    password: password  
  devtools:  
    restart:  
      poll-interval: 2s  
  
  jpa:  
    hibernate:  
      ddl-auto: update  
      database-platform: org.hibernate.dialect.PostgreSQLDialect  
  
  sql:  
    init:  
      mode: always  
  
  rabbitmq:  
    host: localhost  
    port: 4006  
    username: guest  
    password: guest
```

Document 1/1 > spring:

Steps

- In primes-service backend add a package, rabbitmq, and add two classes to it: MQConfiguration and MQSender.
- In the MQConfiguration add a bean that returns a reference to the primes queue:



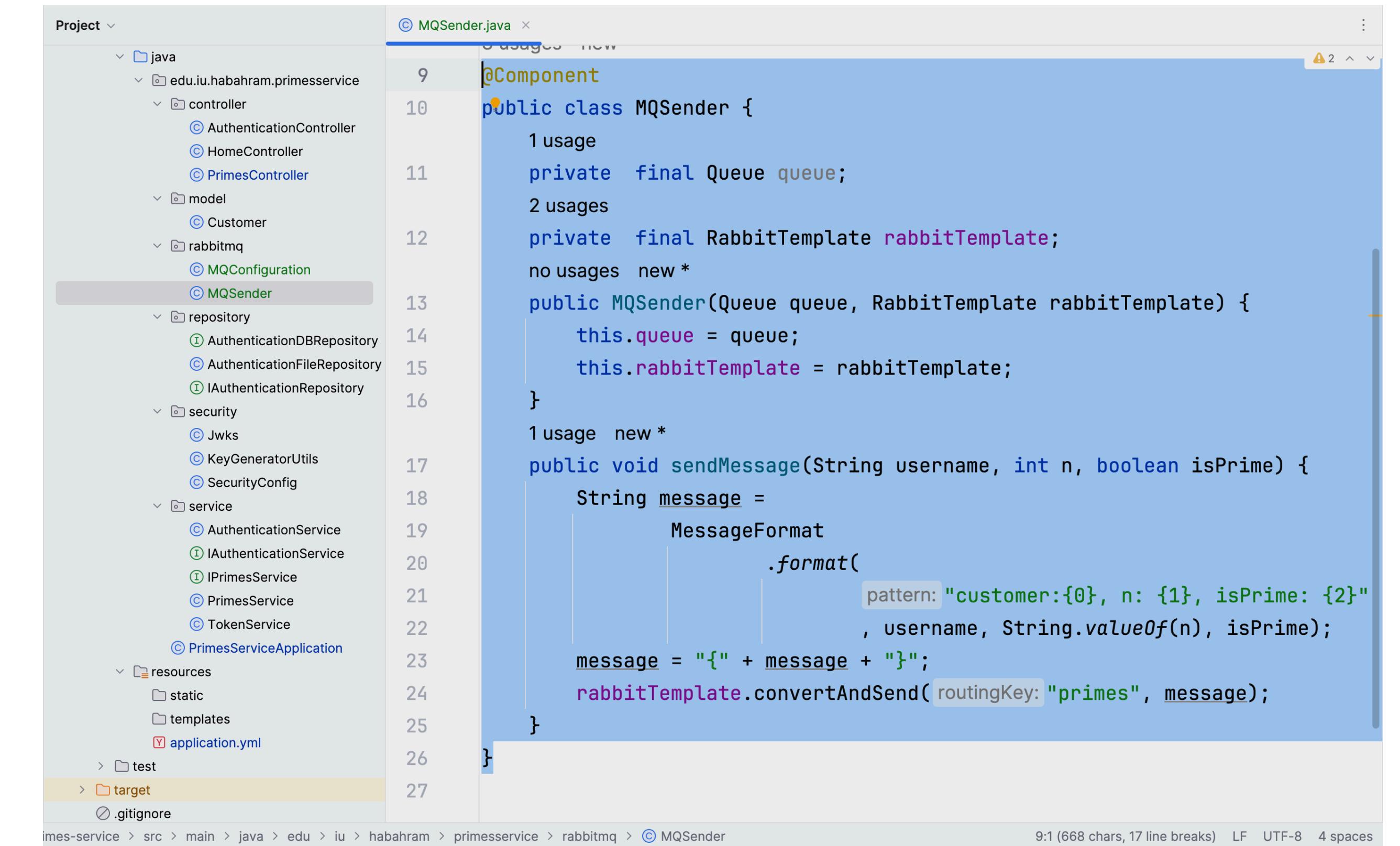
The screenshot shows a Java IDE interface with a project tree on the left and a code editor on the right. The project tree under 'src/main/java' includes packages for controller, model, repository, security, and service, along with a 'rabitmq' package which contains 'MQConfiguration' and 'MQSender'. The code editor displays 'MQConfiguration.java' with the following content:

```
1 package edu.iu.habahram.primesservice.rabbitmq;
2
3 import org.springframework.amqp.core.Queue;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7 no usages new *
8
9
10
11 > @Configuration
12 > public class MQConfiguration {
13
14     @Bean
15     public Queue queue() { return new Queue(name: "primes"); }
16 }
```

The code editor also shows a warning icon at line 10, indicating a potential issue with the '@Bean' annotation. The status bar at the bottom right shows the time as 10:10, file format as LF, encoding as UTF-8, and a 4 spaces indentation setting.

Steps

- In the MQSender add a function that sends a message (the number and a boolean indicating whether it is prime or not) to the primes queue:



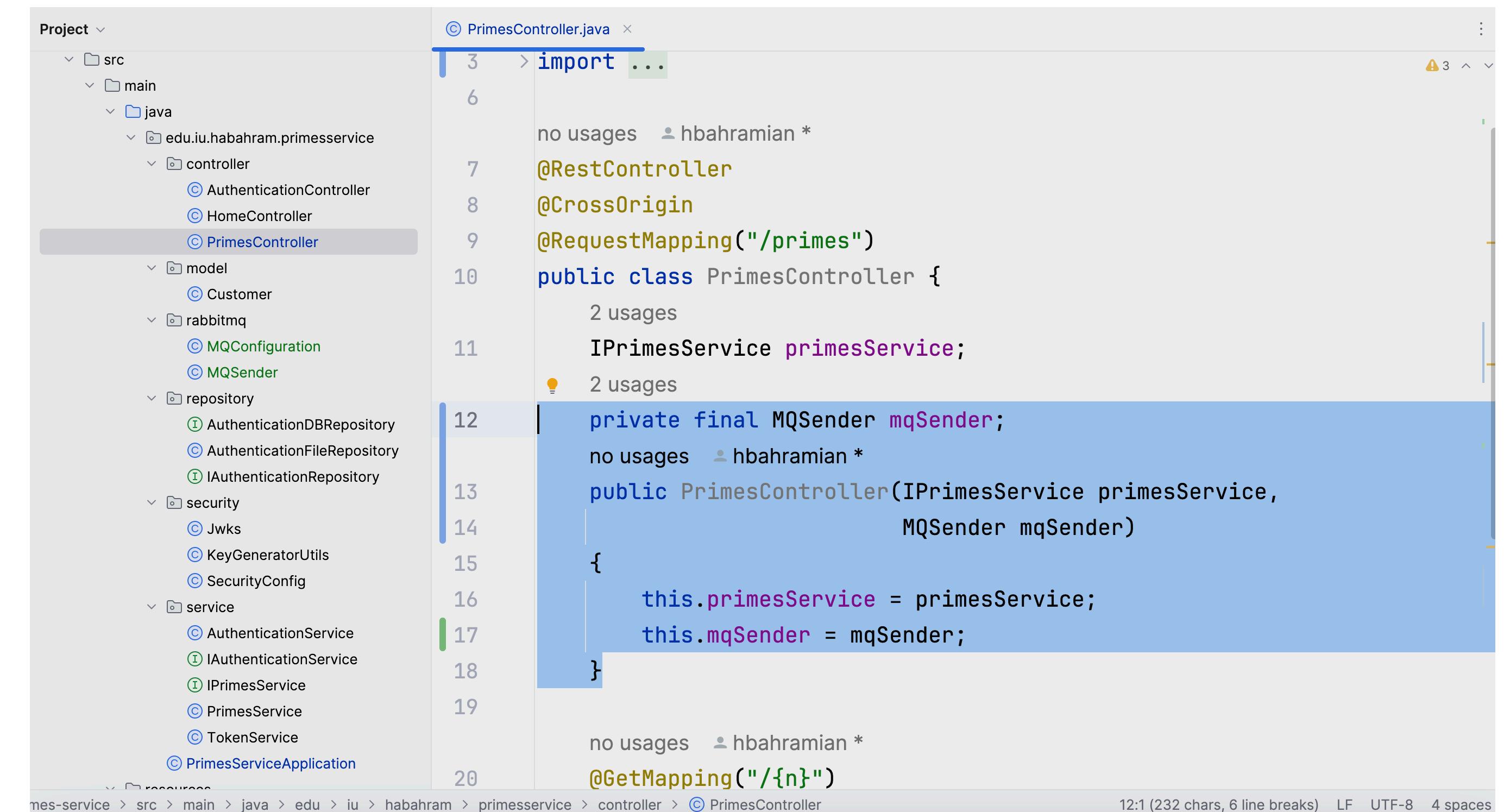
The screenshot shows a Java project structure and a code editor for the file `MQSender.java`. The project structure includes packages for `java`, `rabbitmq`, and `resources`, along with various controllers, models, repositories, services, and configuration files like `application.yml`. The `MQSender.java` file contains the following code:

```
9  @Component
10 public class MQSender {
11     private final Queue queue;
12     private final RabbitTemplate rabbitTemplate;
13     public MQSender(Queue queue, RabbitTemplate rabbitTemplate) {
14         this.queue = queue;
15         this.rabbitTemplate = rabbitTemplate;
16     }
17     public void sendMessage(String username, int n, boolean isPrime) {
18         String message =
19             MessageFormat
20                 .format(
21                     pattern: "customer:{0}, n: {1}, isPrime: {2}"
22                     , username, String.valueOf(n), isPrime);
23         message = "{" + message + "}";
24         rabbitTemplate.convertAndSend( routingKey: "primes" , message);
25     }
26 }
```

The code uses `MessageFormat` to construct a message string with placeholders for `customer`, `n`, and `isPrime`. It then sends this message to a RabbitMQ queue named `primes`.

Steps

- In the PrimesController add a variable of type MQSender:



```
Project ▾
  src
    main
      java
        edu.iu.habahram.primesservice
          controller
            AuthenticationController
            HomeController
            PrimesController
          model
            Customer
          rabbitmq
            MQConfiguration
            MQSender
          repository
            AuthenticationDBRepository
            AuthenticationFileRepository
            IAuthenticationRepository
          security
            Jwks
            KeyGeneratorUtils
            SecurityConfig
          service
            AuthenticationService
            IAuthenticationService
            IPrimesService
            PrimesService
            TokenService
            PrimesServiceApplication
      resources
```

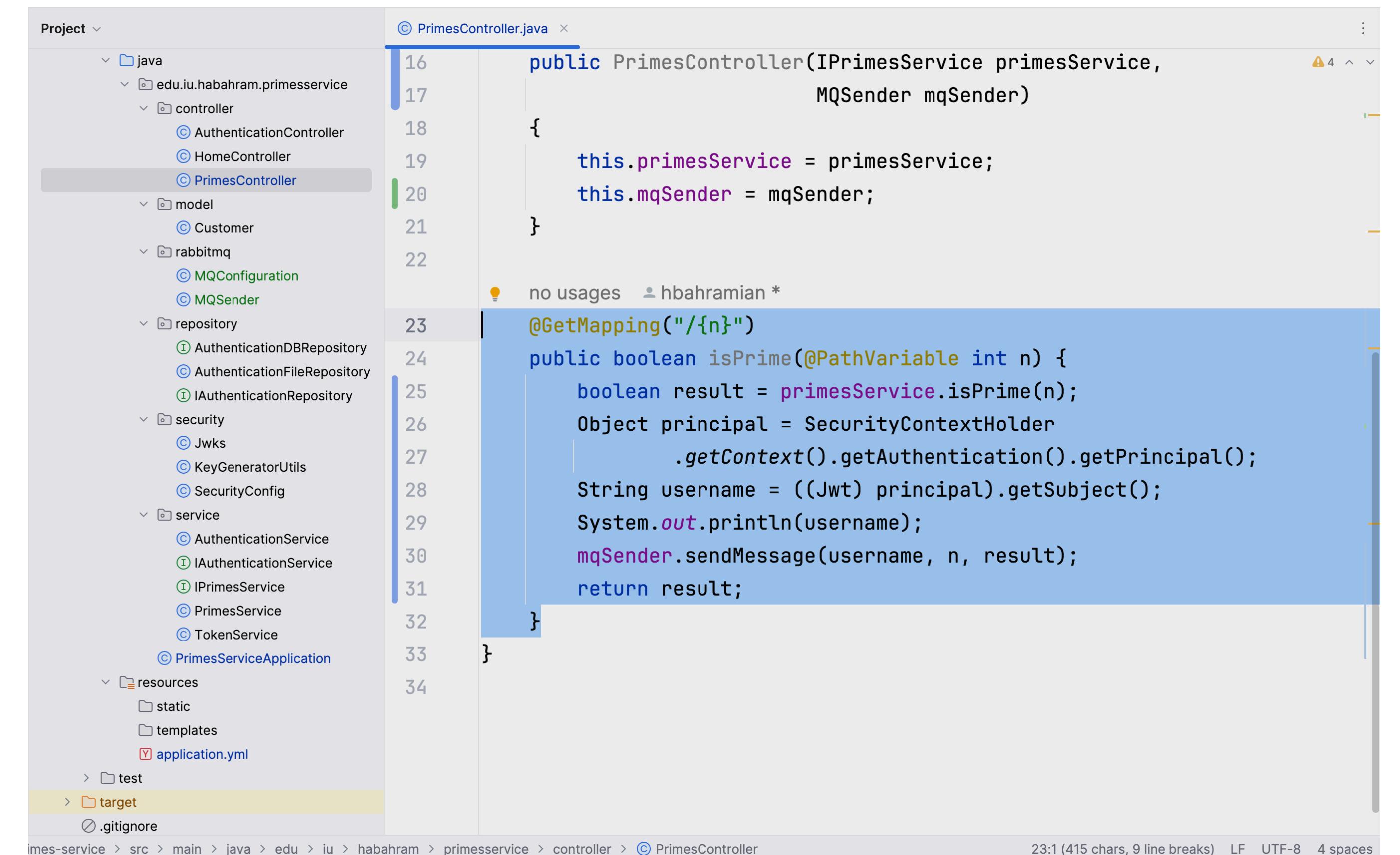
```
① PrimesController.java ×
  3 > import ...
  6
  7 no usages  ↳ hbahramian *
  8 @RestController
  9 @CrossOrigin
  10 @RequestMapping("/primes")
  11 public class PrimesController {
  12   2 usages
  13   IPrimesService primesService;
  14   2 usages
  15   private final MQSender mqSender;
  16   no usages  ↳ hbahramian *
  17   public PrimesController(IPrimesService primesService,
  18                           MQSender mqSender)
  19   {
  20     this.primesService = primesService;
  21     this.mqSender = mqSender;
  22   }
  23
  24   no usages  ↳ hbahramian *
  25   @GetMapping("/{n}")
  26 }
```

mes-service > src > main > java > edu > iu > habahram > primesservice > controller > PrimesController

12:1 (232 chars, 6 line breaks) LF UTF-8 4 spaces

Steps

- In the PrimesController modify the endpoint to send a message containing the number and its primality status to the queue:



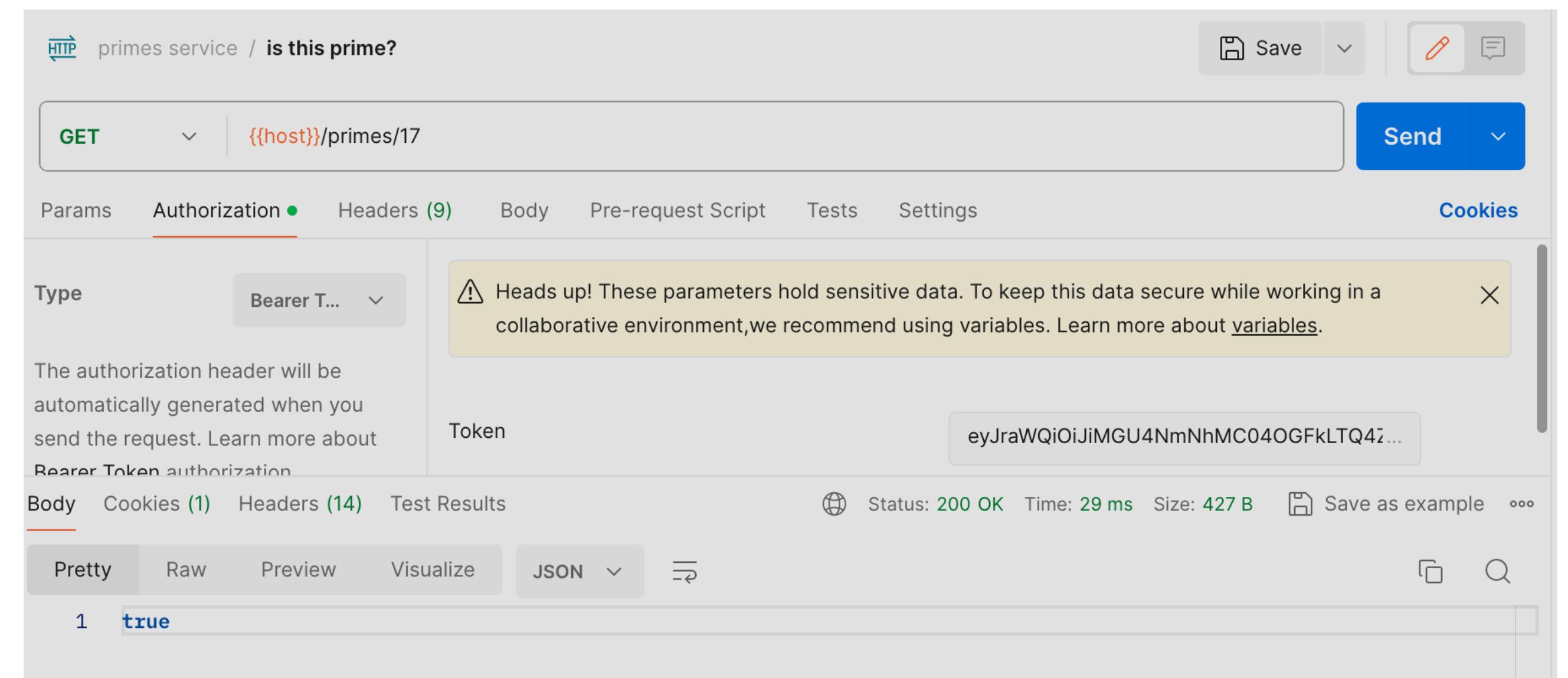
The screenshot shows a Java code editor with the file `PrimesController.java` open. The code defines a controller for handling prime number requests. It includes imports for `IPrimesService`, `MQSender`, and various authentication and security classes. The controller has a constructor that takes `IPrimesService` and `MQSender` as parameters. It also contains a method `isPrime` which checks if a number is prime, retrieves the current user from the security context, prints the user's name to the console, and sends a message to the queue containing the user's name and the prime status. The code editor shows line numbers from 16 to 34.

```
public PrimesController(IPrimesService primesService, MQSender mqSender) {
    this.primesService = primesService;
    this.mqSender = mqSender;
}

@GetMapping("/{n}")
public boolean isPrime(@PathVariable int n) {
    boolean result = primesService.isPrime(n);
    Object principal = SecurityContextHolder
        .getContext().getAuthentication().getPrincipal();
    String username = ((Jwt) principal).getSubject();
    System.out.println(username);
    mqSender.sendMessage(username, n, result);
    return result;
}
```

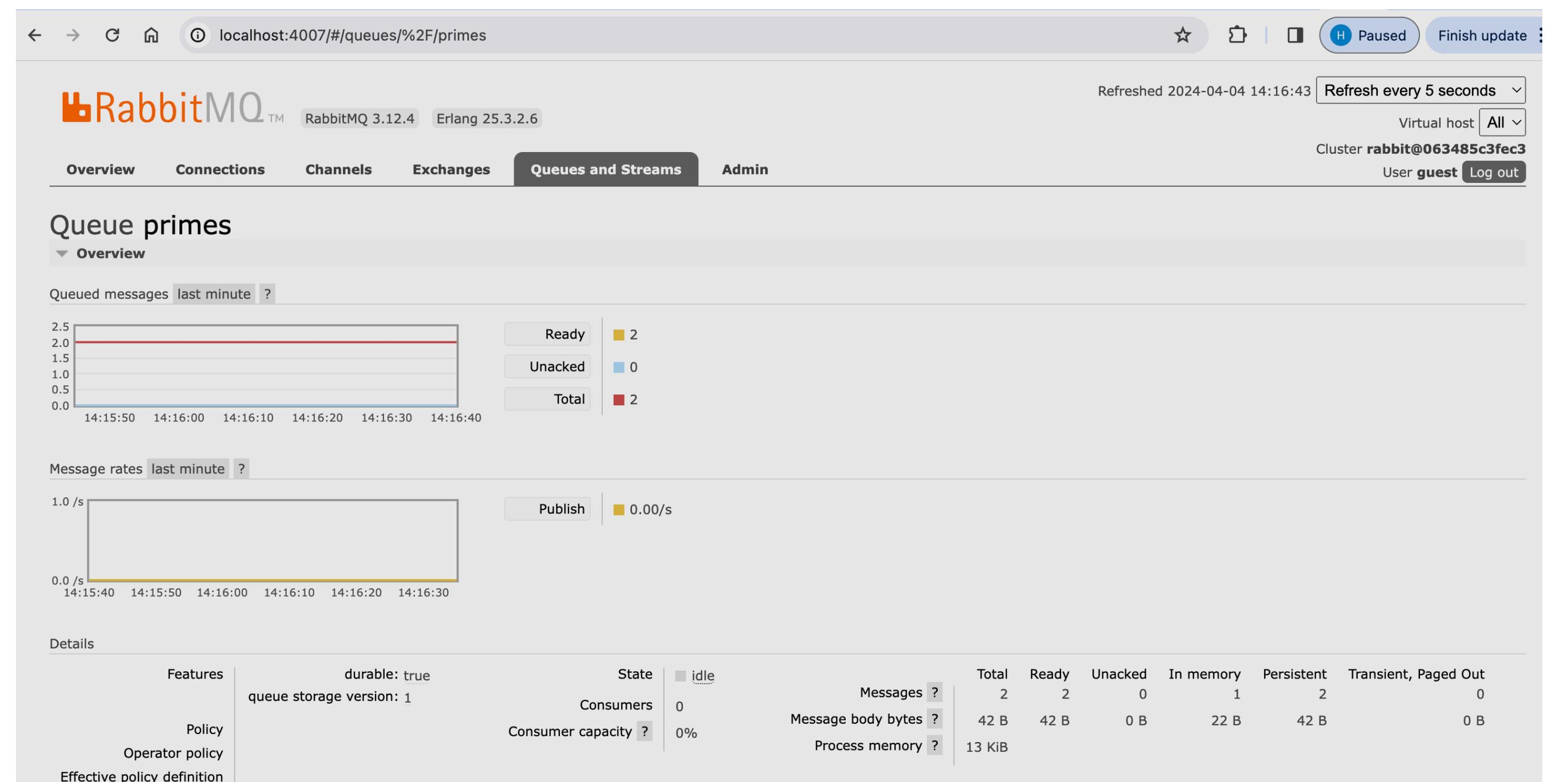
Steps

- Rerun your primes service application and using Postman call the prime check endpoint:



Steps

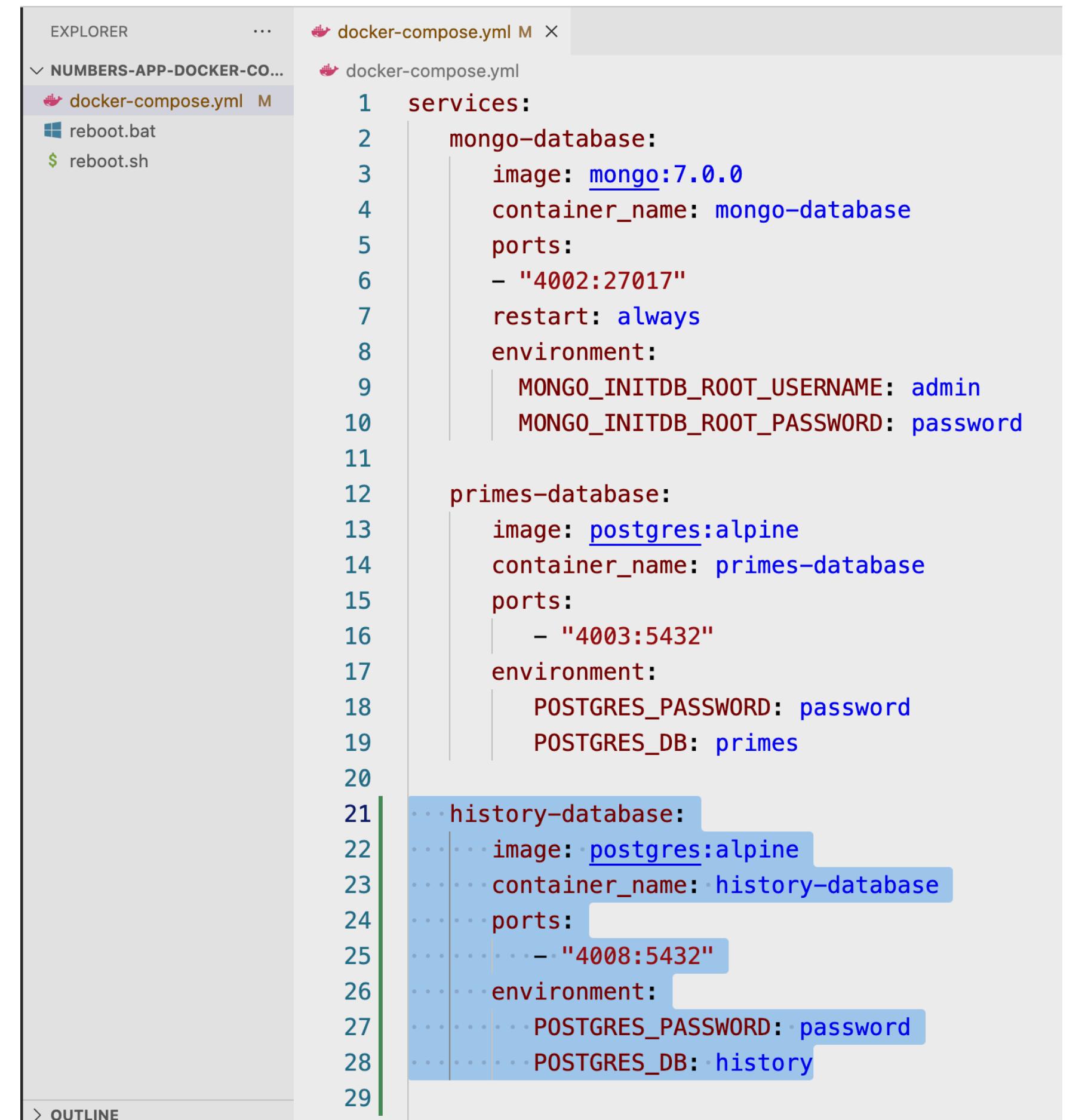
- Go to the RabbitMQ dashboard and verify that the message is received by the queue:



History Service

Steps

- First add a new database server to your docker-compose:



```
EXPLORER ... docker-compose.yml M ...
NUMBERS-APP-DOCKER-CO... docker-compose.yml M ...
reboot.bat
reboot.sh

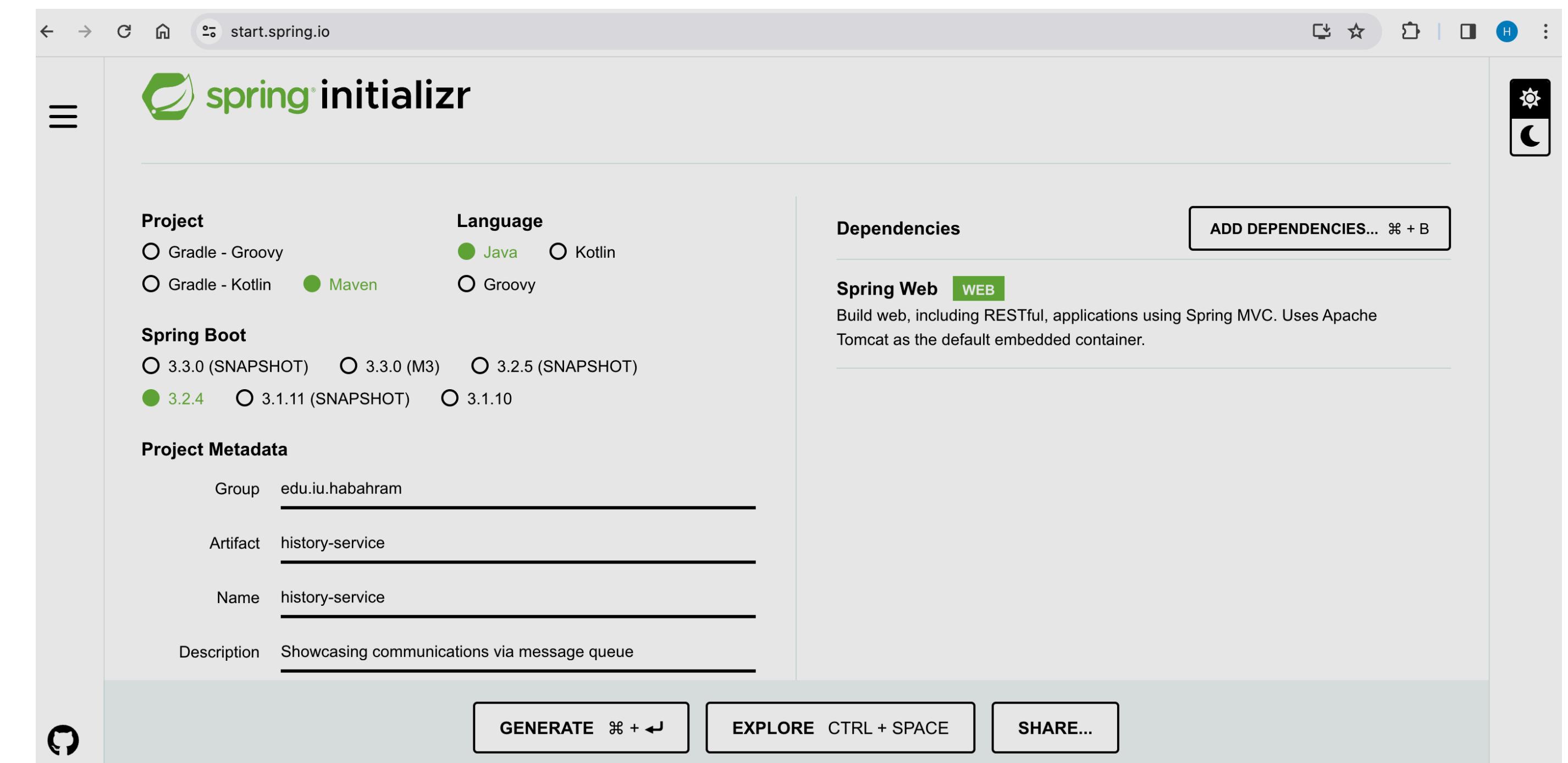
services:
  mongo-database:
    image: mongo:7.0.0
    container_name: mongo-database
    ports:
      - "4002:27017"
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password

  primes-database:
    image: postgres:alpine
    container_name: primes-database
    ports:
      - "4003:5432"
    environment:
      POSTGRES_PASSWORD: password
      POSTGRES_DB: primes

  history-database:
    image: postgres:alpine
    container_name: history-database
    ports:
      - "4008:5432"
    environment:
      POSTGRES_PASSWORD: password
      POSTGRES_DB: history
```

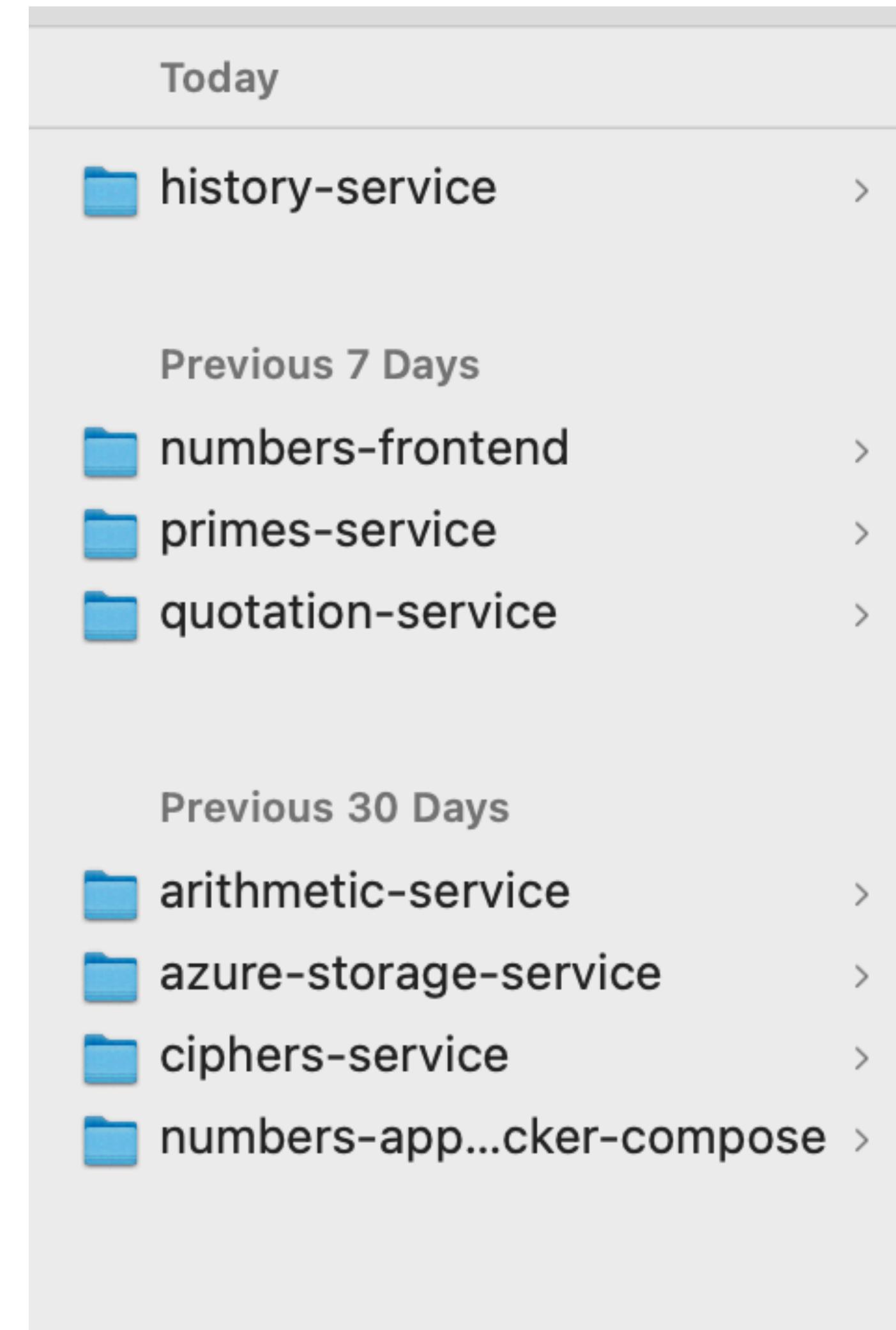
Steps

- Then create the service using spring initializer:
- Go to <https://start.spring.io/> and select Java as the language, Maven as the build tool, and add Spring Web dependency.
- Generate the project and open it using IntelliJ IDEA.



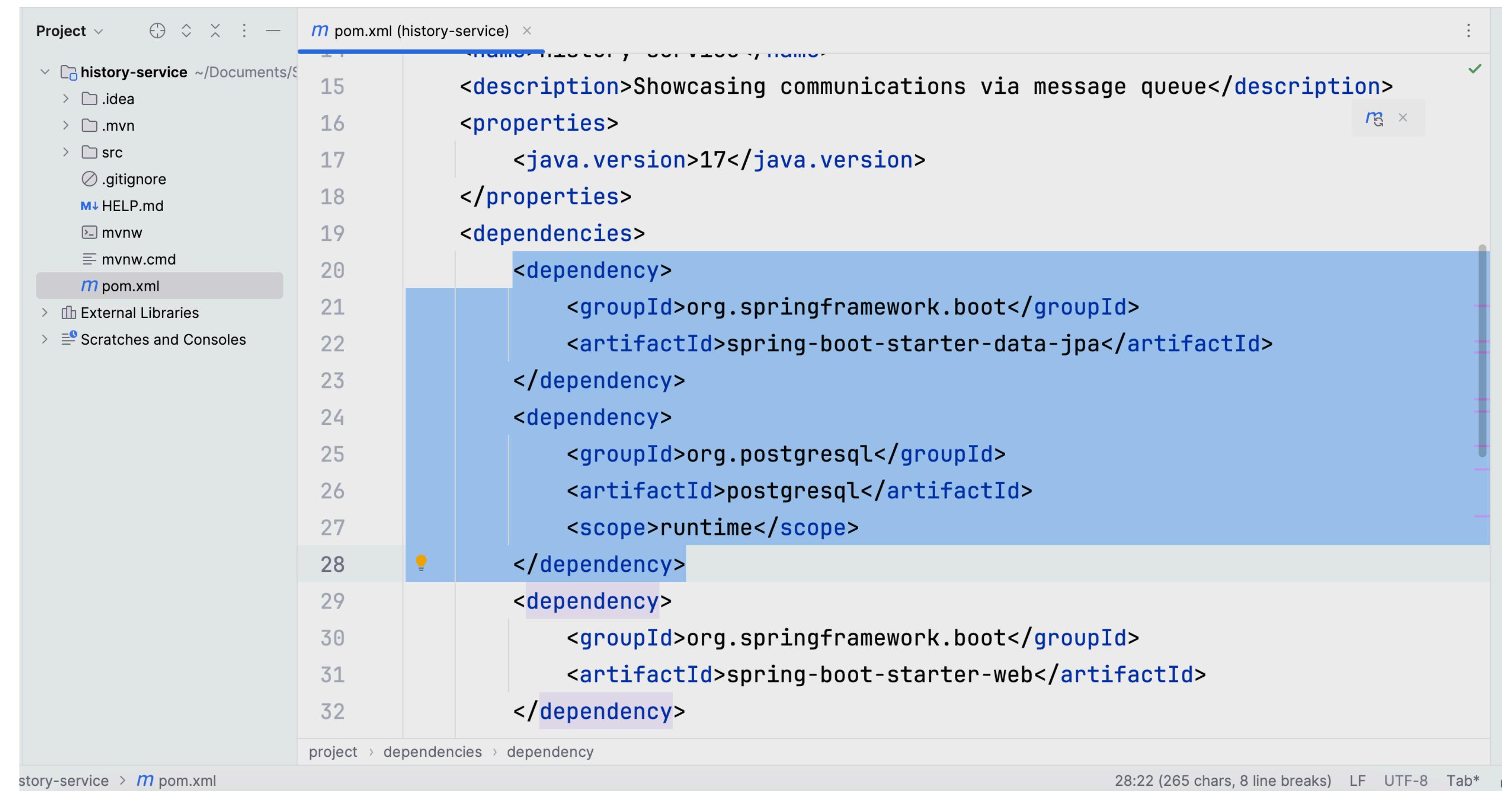
Steps

- Put the history service folder in the same folder where all the other services are located:



Steps

- Add JPA and Postgres dependencies to the project:



```
15 <description>Showcasing communications via message queue</description>
16 <properties>
17   <java.version>17</java.version>
18 </properties>
19 <dependencies>
20   <dependency>
21     <groupId>org.springframework.boot</groupId>
22     <artifactId>spring-boot-starter-data-jpa</artifactId>
23   </dependency>
24   <dependency>
25     <groupId>org.postgresql</groupId>
26     <artifactId>postgresql</artifactId>
27     <scope>runtime</scope>
28   </dependency>
29   <dependency>
30     <groupId>org.springframework.boot</groupId>
31     <artifactId>spring-boot-starter-web</artifactId>
32   </dependency>
```

Steps

- Add a model, `PrimesRecord`, to store the history of primality checks:

The screenshot shows the IntelliJ IDEA interface with the project tree on the left and the code editor on the right. The project tree shows a structure under the `history-service` root, including `.idea`, `.mvn`, `src` (containing `main`, `java`, `rabbitmq`, `repository`, `resources`, `test`, and `target`), and files like `.gitignore`, `HELP.md`, `mvnw`, and `mvnw.cmd`. The code editor displays the `PrimesRecord.java` file with the following content:

```
8 usages
8 @Entity
9 @Table
10 public class PrimesRecord {
11     2 usages
12     @Id
13     @GeneratedValue
14     private int id;
15     2 usages
16     private String customer;
17     2 usages
18     private String n;
19     2 usages
20     private boolean isPrime;
21
22     no usages
23     public int getId() { return id; }
24
25     no usages
```

The code editor shows several annotations and fields. Lines 8 and 9 are annotated with `@Entity` and `@Table` respectively. Lines 12 and 13 show the `@Id` and `@GeneratedValue` annotations. Lines 16 and 17 show the `private String customer;` and `private String n;` fields. Line 20 shows the `private boolean isPrime;` field. Lines 23 and 25 show the `public int getId() { return id; }` method. Lines 18 and 25 are marked as having 'no usages'.

Steps

- Add the repository:

The screenshot shows a Java IDE interface with a project tree on the left and a code editor on the right.

Project Tree:

- Project
- |.idea
- |.mvn
- src
- main
- java
- edu.iu.habahram.historyservice
- controllers
- PrimesHistoryController
- model
- PrimesRecord
- rabbitmq
- MQReceiver
- repository
- PrimesHistoryRepository
- HistoryServiceApplication
- resources
- static
- templates
- application.yaml
- test
- target
- .gitignore
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml
- External Libraries

Code Editor (PrimesHistoryRepository.java):

```
import edu.iu.habahram.historyservice.model.PrimesRecord;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

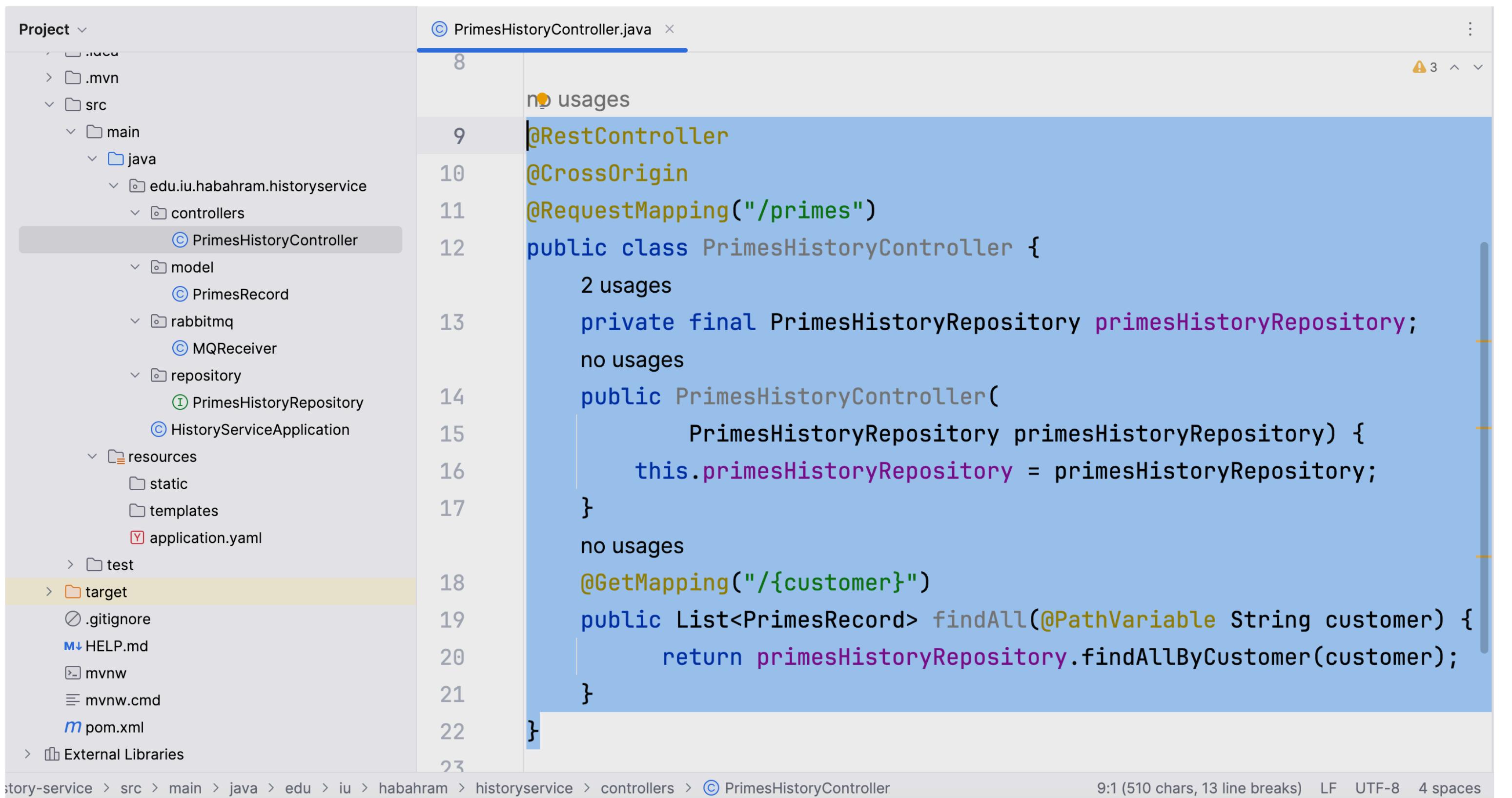
import java.util.List;

@Repository
public interface PrimesHistoryRepository
    extends CrudRepository<PrimesRecord, Integer> {
    List<PrimesRecord> findAllByCustomer(String customer);
}
```

The code editor shows the `PrimesHistoryRepository.java` file. The `@Repository` annotation is highlighted in yellow. The `findAllByCustomer` method is also highlighted in yellow. The code editor status bar at the bottom indicates the file has 9:1 (171 chars, 4 line breaks) and is in LF, UTF-8, 6 spaces format.

Steps

- Add the controller:



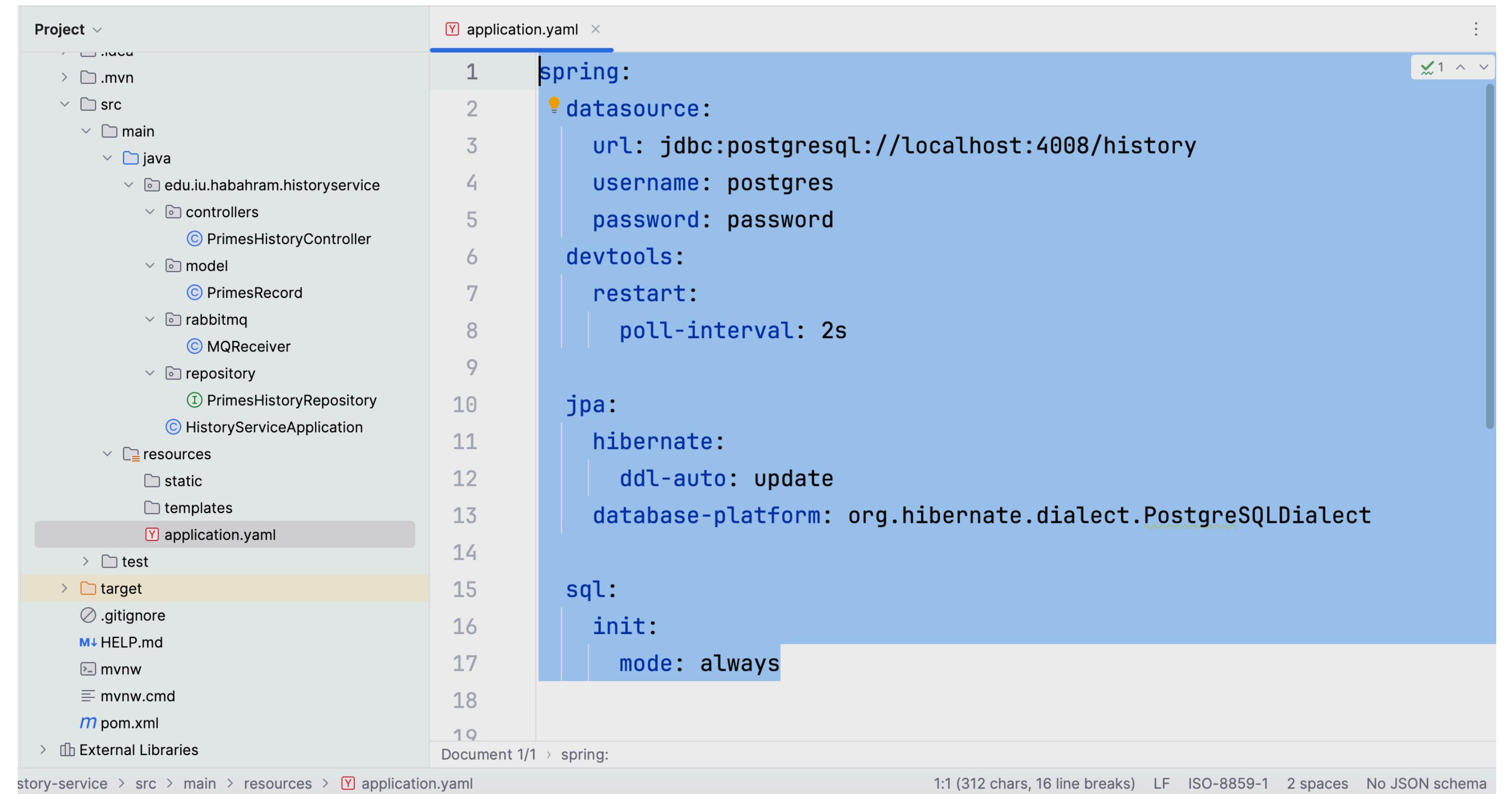
The screenshot shows a Java code editor with the file `PrimesHistoryController.java` open. The code defines a REST controller for handling prime numbers. The controller has a constructor that takes a `PrimesHistoryRepository` dependency and initializes it. It also contains a method to find all prime records for a given customer.

```
8  no usages
9  @RestController
10 @CrossOrigin
11 @RequestMapping("/primes")
12 public class PrimesHistoryController {
13     2 usages
14     private final PrimesHistoryRepository primesHistoryRepository;
15     no usages
16     public PrimesHistoryController(
17         PrimesHistoryRepository primesHistoryRepository) {
18         this.primesHistoryRepository = primesHistoryRepository;
19     }
20     no usages
21     @GetMapping("/{customer}")
22     public List<PrimesRecord> findAll(@PathVariable String customer) {
23         return primesHistoryRepository.findAllByCustomer(customer);
24     }
25 }
```

The left side of the interface shows the project structure, which includes a `.mvnw` file and a `target` folder containing `.gitignore`, `HELP.md`, `mvnw`, `mvnw.cmd`, and `pom.xml`.

Steps

- Add the configuration properties:



The screenshot shows a code editor with a project structure on the left and the `application.yaml` file on the right.

Project Structure:

- `.idea`
- `.mvn`
- `src`
 - `main`
 - `java`
 - `edu.iu.habahram.historyservice`
 - `controllers`
 - `PrimesHistoryController`
 - `model`
 - `PrimesRecord`
 - `rabbitmq`
 - `MQReceiver`
 - `repository`
 - `PrimesHistoryRepository`
 - `HistoryServiceApplication`
 - `resources`
 - `static`
 - `templates`
 - `test`
 - `target`
 - `.gitignore`
 - `HELP.md`
 - `mvnw`
 - `mvnw.cmd`
 - `pom.xml`
 - `External Libraries`

Steps

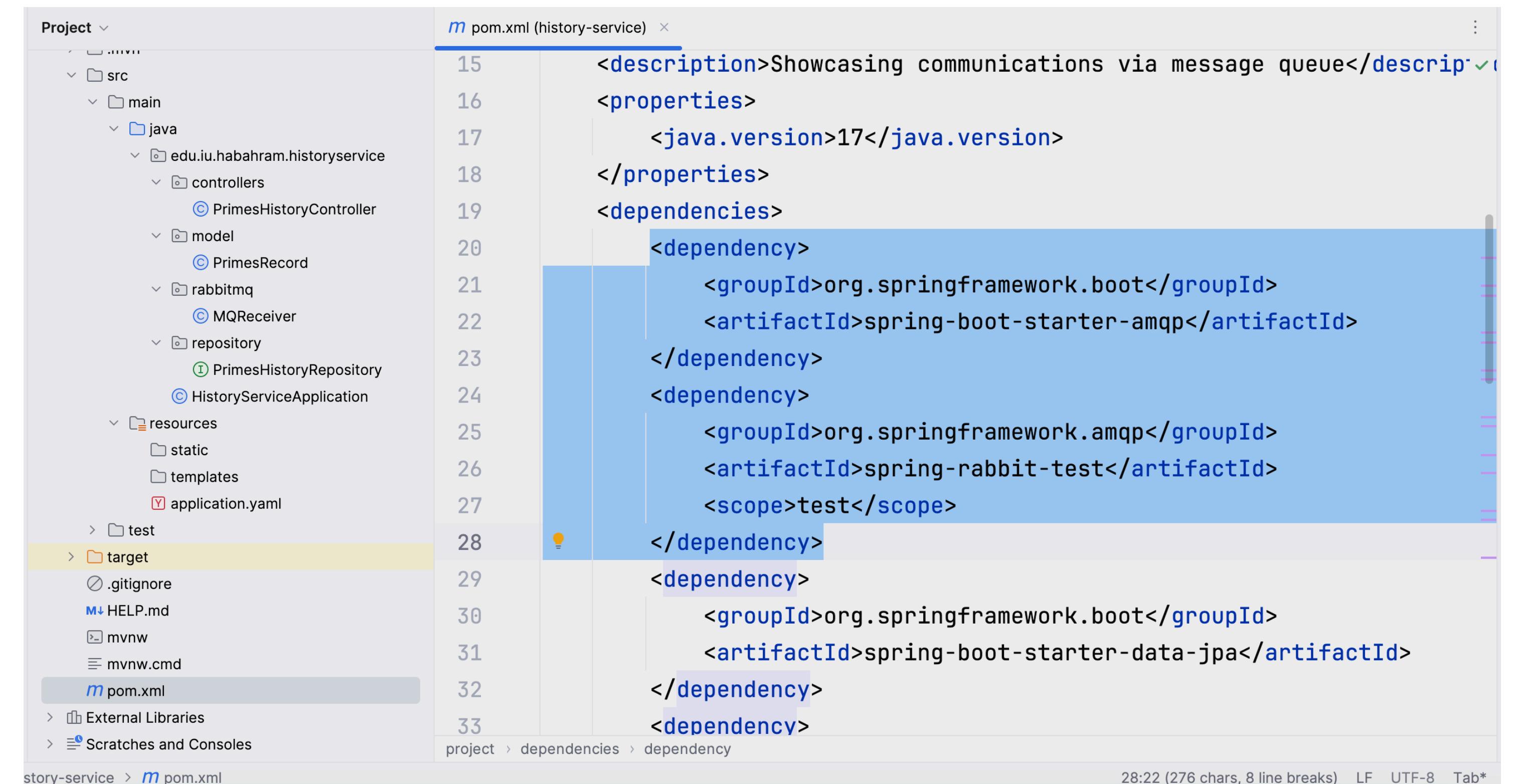
- Using postman verify that the history service is working. For now it should return an empty list.

The screenshot shows the Postman application interface. On the left, the sidebar lists services: 'azure file storage', 'history service' (which is expanded), 'primes service' (also expanded), and 'quotations-service'. Under 'history service', there are three methods: 'GET get prime records' (selected), 'GET greetings', and 'GET is this prime?'. Under 'primes service', there are two methods: 'POST register' and 'POST login'. The main workspace shows a request configuration for 'GET get prime records'. The URL field contains `http://{{host}}/primes/john`. Below the URL, tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', 'Tests', and 'Settings' are visible. A 'Cookies' tab is also present. The 'Params' tab is selected, showing a table with one row:

Key	Value	Description
Key	Value	Description

Steps

- Add the rabbitmq dependencies to the pom.xml:

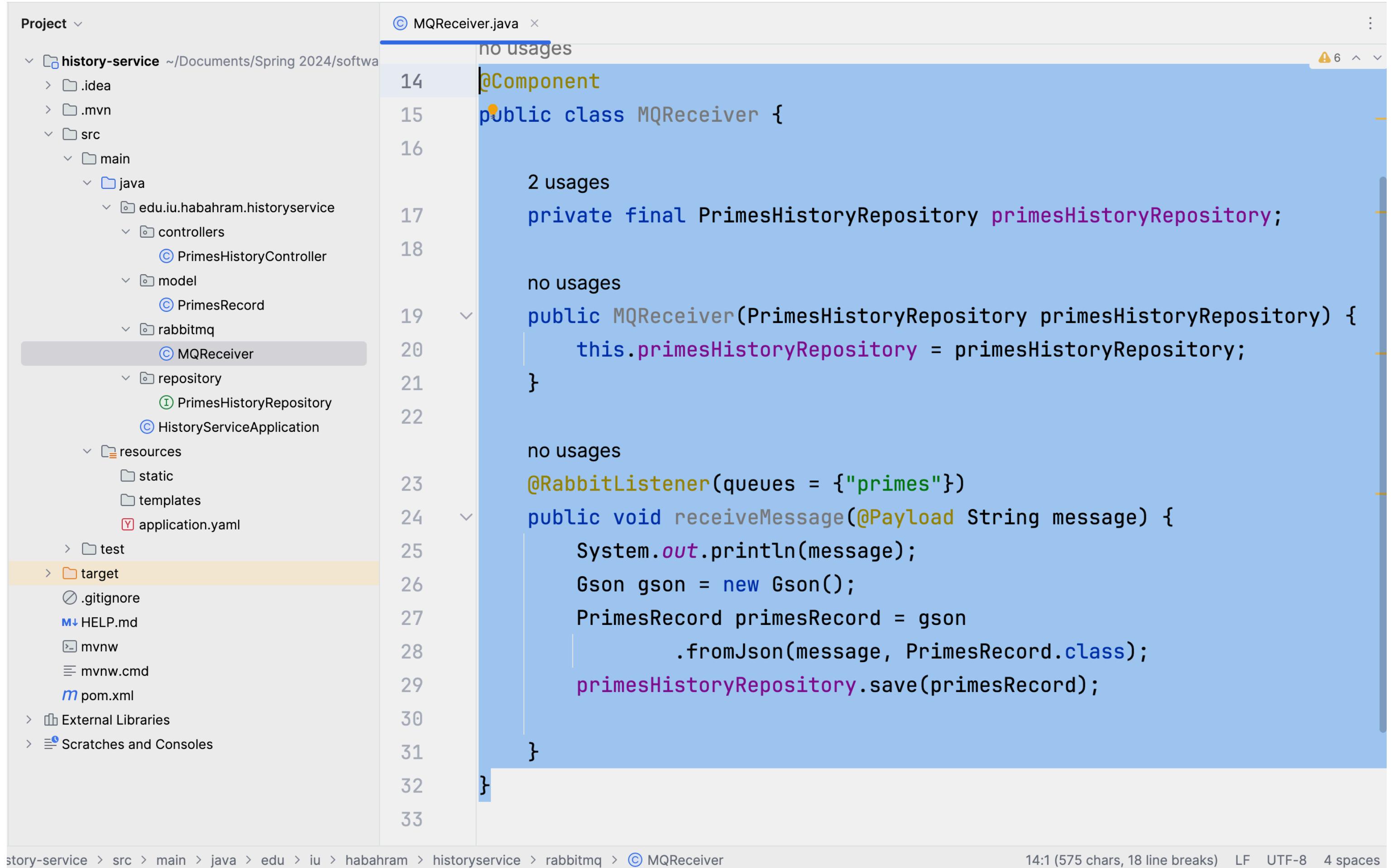


The screenshot shows a Java project structure on the left and its corresponding pom.xml file on the right. The project structure includes src/main/java, src/test, target, and various configuration files like .gitignore, HELP.md, mvnw, mvnw.cmd, and application.yaml. The pom.xml file lists several dependencies, including org.springframework.boot:spring-boot-starter-amqp, org.springframework.amqp:spring-rabbit-test (with scope test), org.springframework.boot:spring-boot-starter-data-jpa, and org.springframework.boot:spring-boot-starter-web.

```
15 <description>Showcasing communications via message queue</description>
16 <properties>
17   <java.version>17</java.version>
18 </properties>
19 <dependencies>
20   <dependency>
21     <groupId>org.springframework.boot</groupId>
22     <artifactId>spring-boot-starter-amqp</artifactId>
23   </dependency>
24   <dependency>
25     <groupId>org.springframework.amqp</groupId>
26     <artifactId>spring-rabbit-test</artifactId>
27     <scope>test</scope>
28   </dependency>
29   <dependency>
30     <groupId>org.springframework.boot</groupId>
31     <artifactId>spring-boot-starter-data-jpa</artifactId>
32   </dependency>
33   <dependency>
```

Steps

- Add a rabbitmq receiver that receives the messages from the queue and stores them in the Postgres database:



The screenshot shows a Java code editor in an IDE. On the left is a project tree for a Maven project named "history-service". The "src/main/java" directory contains packages for "edu.iu.habahram.historyservice.controllers", "model", and "rabbitmq". The "rabbitmq" package contains the file "MQReceiver.java", which is currently open. The code in "MQReceiver.java" is as follows:

```
no usages
14 @Component
15 public class MQReceiver {
16
17     private final PrimesHistoryRepository primesHistoryRepository;
18
19     public MQReceiver(PrimesHistoryRepository primesHistoryRepository) {
20         this.primesHistoryRepository = primesHistoryRepository;
21     }
22
23     @RabbitListener(queues = {"primes"})
24     public void receiveMessage(@Payload String message) {
25         System.out.println(message);
26         Gson gson = new Gson();
27         PrimesRecord primesRecord = gson
28             .fromJson(message, PrimesRecord.class);
29         primesHistoryRepository.save(primesRecord);
30     }
31 }
32
33
```

The code editor shows syntax highlighting for Java keywords and annotations like @Component and @RabbitListener. Line numbers are visible on the left. The status bar at the bottom indicates the file has 14:1 (575 chars, 18 line breaks) and is in LF, UTF-8, 4 spaces format.

Steps

- Test the sender/receiver:
- Using your primes services send a prime check request.
- Using your history service verify that the prime check request is stored in the database.

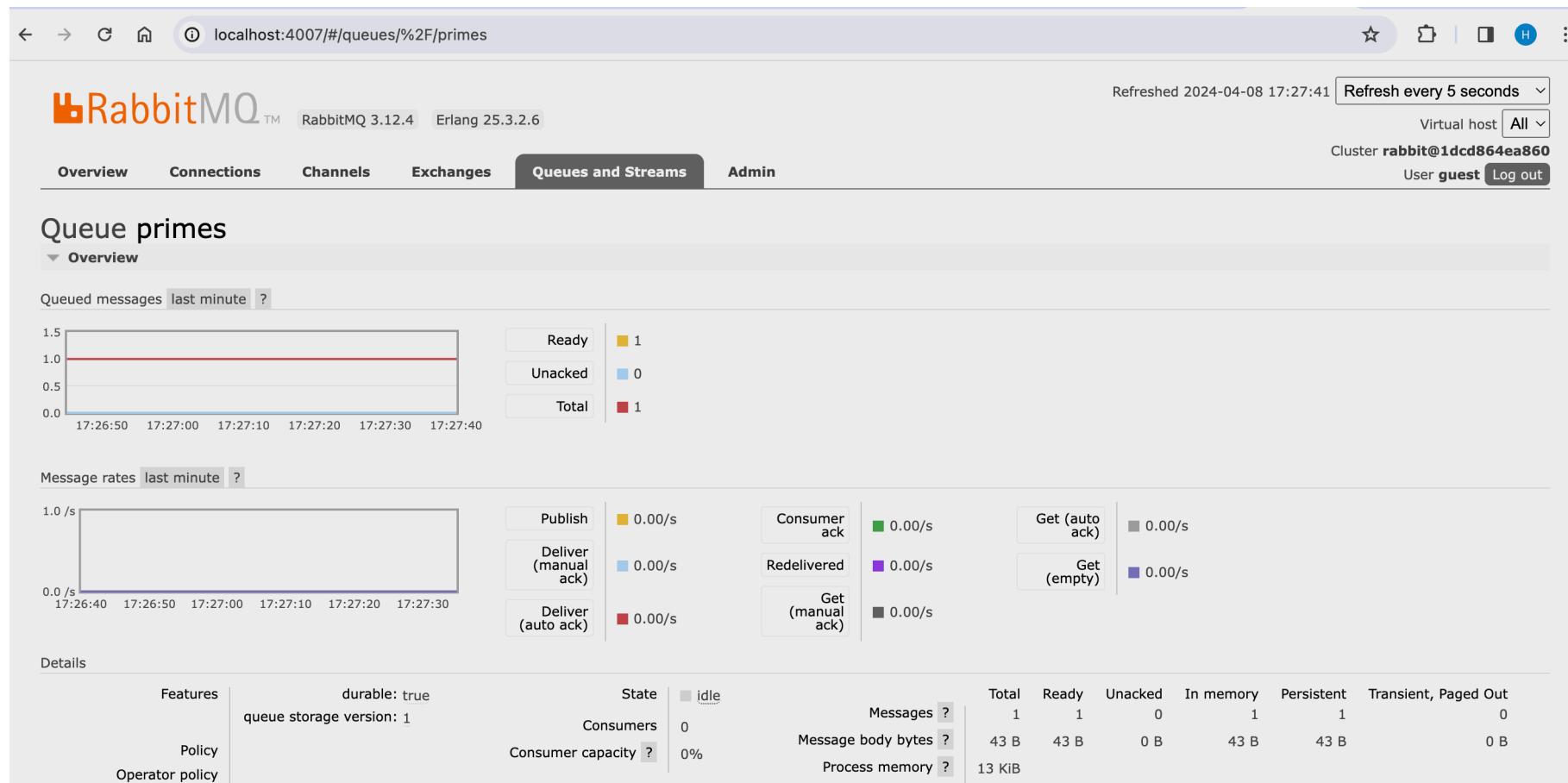
The screenshot shows a Postman request to the URL `HTTP://primes service / is this prime? {{host}}/primes/567913`. The request method is GET. In the Authorization tab, it is set to Bearer Token, and a warning message states: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about [variables](#)". The token value is eyJraWQiOiIOWZlODBmYl0zODU3LTQ3ZD... . The response status is 200 OK, time: 28 ms, size: 428 B. The body contains the JSON object:

```
1 false
```

.

The screenshot shows a Postman request to the URL `HTTP://history service / get prime records {{host}}/primes/john`. The request method is GET. In the Headers tab, there are 7 items. The response status is 200 OK, time: 18 ms, size: 357 B. The body contains the JSON array:

```
1 [ 2 { 3 "id": 4, 4 "customer": "john", 5 "n": "17", 6 "prime": true 7 }, 8 { 9 "id": 5, 10 "customer": "john", 11 "n": "567913", 12 "prime": false 13 } 14 ]
```



The END

- That was it for today! Congratulations!
- Submit the url of your primes service repository.
- Submit the url of your history service repository.