Pizza Ordering App review

This application proved to be a large review to cover within the 1 hours window. I have added my high level thoughts under the '**What do you think about the overall architecture of this app ?**' question. More fine grain comments are addressed in the **Full Review comments** section at the end.

What do you think about the overall architecture of this app ?

Overall the application architecture is pretty weak and falls down on multiple fronts:

- Security
- Error/Exception handling
- God classes and buried business logic
- Correct separation of layers (web, service, persistence)
- State management
- Scalability
- Reusability
- Readability

.

What would be the most valuable improvements ?

1. Ongoing purchases need to be moved to some sort of persistence to allow application scaling
2. Core business logic needs to be extracted from services into a dedicated package or event better a maven module
3. Testing
4. Transaction handling review
5. Exception handling fixed/ added
6. Authorization moved to REST layer
7. Introduce JWT and OAuth for security and performance improvements
8. Introduce a DTOs and VOs
9. DAO layer
10. Immutability on models (DTOs, VOst etc..) should be enforced where possible expect on the entity classes

What's missing to have the application production ready?

1. Testing (Unit, integration, E2E based on BDD)
2. Logging
3. Exception handling
4. Persistence for ongoing orders to allow scaling

5. Remove secrets from application properties
6. Environment specific properties should be read from environment variables, database url for example.
7. Database schema management needs to be added, tools like liquibase or flyway would help here. Replacing with document datastore would also simplify this.
8. Java version should be updated, 1.8 premiere support ends 2022
9. Properties should be used to manage dependency versions in pom
10. Status analysis should be added as part of build, sonar integration for example.
11. Performances and pen testing also need to be considered.
12. Hardcoded jar snapshot number in docker file is problematic as this will change when the app version is updated in pom.

Is there any framework, library, tools that you know would greatly benefit this app
and/or development ?

1. Lombok - Make use of the builder pattern
2. Some security frame, spring security
3. Logging framework, logback for example
4. Command handler pattern should be introduced to help isolate core business logic, plus extracting this to a core maven module or even a core package would help.
5. Maven Jib plugin to build the docker image

6. Consider a document store (i.e) to manage a purchase object. This will improve read write performance and greatly simplify the codebase.
7. In memory database or Test Container
    a. For maven build and execution of tests a local database or making use of test containers would make things easier, especially in terms of data clean up
8. Mockmvc for REST later testing

Full Review comments

Management Service:

For a naming perspective I have no idea what the purpose of this class is. From looking into the code it appears to allow the number of purchases to be queries, Perhaps this logic belongs in the purchase service.
Also the user authorization is validated within the get purchases count method, this authorization role check should be applied as early as possible in the app, moving to the Management Controller would be best here.

Get current user retrieves a user based on a hardcoded id, this will return the same user each time. Also throwing an illegal exception makes no sense here, an Optional type would be better.
Get current user seems like a useful piece of functionality that should be exposed for users elsewhere in the application, moving this the User details service is best.

Email Service
Properties should be passed to the service during application configuration as constructor attributes.
Use of system.out should be replaced with some logging framework.
Too much information is leaking into the email service, all it requires is the email address of the recipient.
Logging the recipient's email address is a potential security concern (PII data).
The logic to send the email should move to an email adapter/port implementation to allow easier testing of the email service.
The introduction of a Notification service where an email adapter is one of the notification implementations would allow scaling to multiple notification types in the future.
From address should be passed as one of the properties at construction time.
The email service exception handling needs work.

Purchase Service
Ongoing purchases is not thread safe and will not scale once more than one instance of the application is running.
User repository is unused
Security should be applied at the controller level.
Add pizza to purchase is to complex, perhaps a getOrCreatePurchase method at the repo level would make more sense. Also could a basket or order concept be added to the model here, to allow separation or the purchase logic and actually tracking the order.
PizzaExpection is too generic, plus some logging should be added when this occurs.

Use of linked list
- Performance issue
- Uses more memory than Array List

Linked list should be replaced with Array list

Use or security annotations should be pushed to the REST layer.
Alot of the purchase logic should be supplied with a purchase id in method arguments .
The purchase service is looking like a god class and should be considered for redesign, perhaps broken up into more understandable and testable classes.
No validation is performed on data returned from the repository layer, null pointers are just waiting to happen in the service.
Ongoing purchases should be persisted somewhere, some kind of cache would be good, perhaps a redis db would be best suited here. A queue or message broker could also be leveraged here. The message broker would also allow a scaling option for the application.

Email service should be passed in the construction at application startup.

No exception logging.
Compute amount and managing Purchase state appear to be core application logic ‘Discount Rules’ should be treated as such. With the current architecture the logic is buried in the service.

Purchase States.
A small state machine could be introduced.. The application should validate transitions between these states.
Placed to Draft or Served to draft for example are not valid state transitions and should be prevented.

General

Mixing the domain and entity model is leaking the persistence layer into the core application

Use of autowiring is discouraged for testing and mocking simplicity, these should be replace with constructor based dependency injection

Core application logic is buried in services. This logic should be isolated and easy to find. Perhaps a command handler pattern to help with this.
Clean up all unused dependencies

Application transaction logic needed to be reviewed in this application. We have multiple situations where security implementation is inconsistent, some query the database directly while others application spring based annotations. Plus security should be applied as early as possible (i.e as the REST layer)
Logging should be added .
Exception handling architecture needs to be addressed.
The service layer functionality should really be exposed behind interfaces (not required but it is good practise).
Plus services should orchestrate calls to the application core business logic.
Add a state machine to maintain the Purchase state, spring state machine perhaps.

The service layer should return model types, the REST layer should add a DTO layer for request and response.
Use of the spring service annotation is not best practise, the service layer should not be tied to any specific framework. All the application wiring can be centralized in an application configuration class/.

Repository interface should be wrapped by a DAO layer with a more readable APIs. These query specific conventions on the spring data repos should be hidden from the core and service logic.

It's good practice to perform validations on any incoming data, the service layer could do with adding such validation

Pagination on database queries and on the REST layer.
Exception handling and correct HTTPs error codes should be introduced.
Direct access to repositories is not good practise from the REST layer, service APIs should be leveraged here.