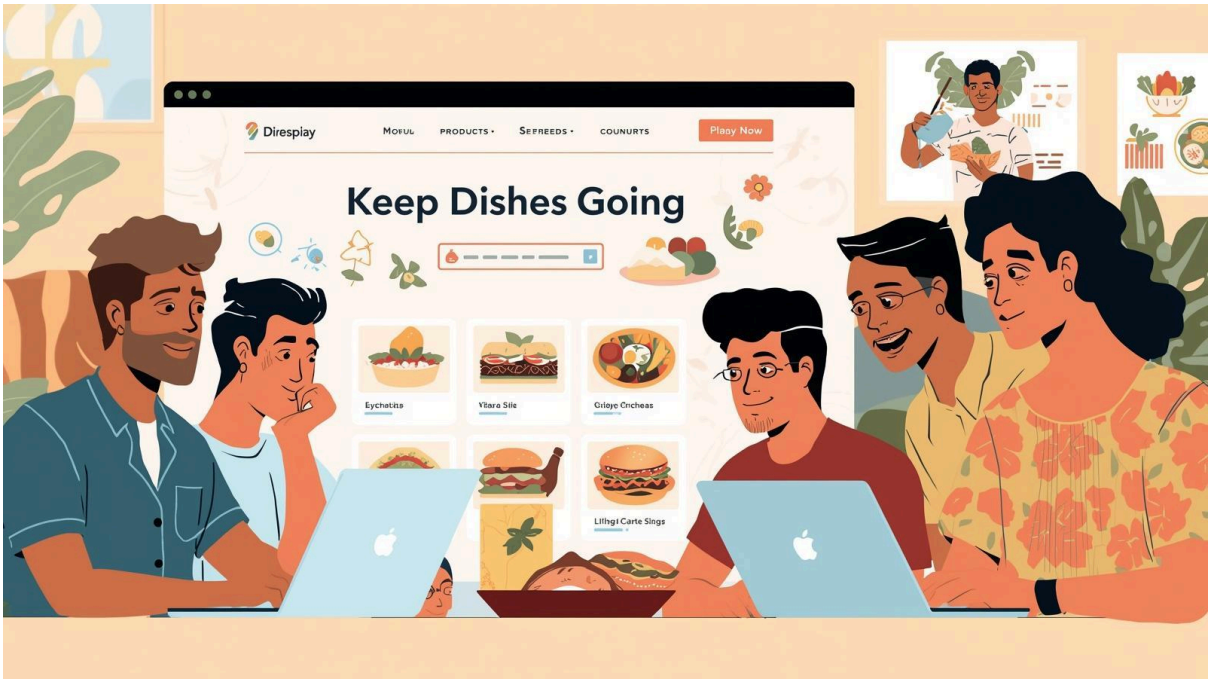


Introduction

Keep Dishes Going (KDG) is a new food ordering platform aiming to break into the market by building fast, clear, and reliable experiences. We believe fresh ideas come from young, eager developers—so this project is designed to channel your skills into a real-world scenario: restaurants control what they offer, customers order with confidence, and handoffs to delivery are smooth and on time. Your work will shape how KDG keeps menus accurate, decisions prompt, and order status transparent from checkout to delivery.

Brief

You will build a small marketplace where restaurants present what they offer and customers can place and follow orders through to handoff. The system should keep what customers see in sync with what restaurants intend to sell, make timely decisions about new orders, and clearly signal when food is ready.

The landing page should offer a clear choice to continue as an owner or as a customer.

Owners sign up/sign in through the platform. On their first successful sign-in, if no restaurant is linked to that owner, the app should present a create restaurant form. Each owner manages exactly one restaurant. The form must collect:

- Restaurant name
- Full address (street, number, postal code, city, country)
- Contact email
- Picture(s) (URL)
- Type of cuisine (Italian, French, Japanese, …)
- Default preparation time
- Opening hours (weekly schedule)

Editing these details later is out of scope for this project.

Restaurants control their menu dish by dish. A dish has the following attributes:

- Name
- Type (starter, main, dessert)
- Food tags (lactose, gluten, vegan,...)
- Description
- Price
- Picture (URL)

Owners edit individual dishes and save those edits as drafts. The interface should clearly show what's live versus what has pending changes, including a running count of dishes whose changes are not yet reflected on the live menu—whether those changes will publish new dishes or take dishes off the menu. For each dish, the owner can publish or unpublish; if several dishes are pending, the owner can apply all changes now or schedule changes so they go live together at a chosen time. Marking a dish out of stock (or back in stock) is always immediate and cannot be scheduled.

Dishes can be in one of three states. Published dishes are visible and can be ordered. Out-of-stock dishes remain visible but can't be added to a basket until availability returns. Unpublished dishes are invisible to customers and can't be ordered.

These are fine-dining restaurants with a deliberately tight selection: at any moment, no more than 10 dishes should be available to customers.

Customers do not need to sign up/sign in.

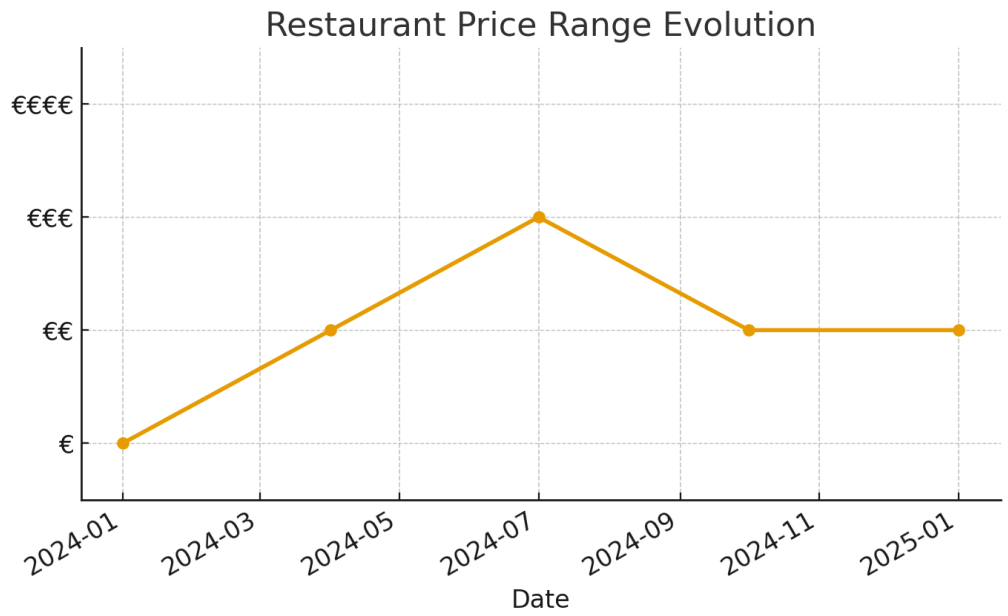
Customers can explore all restaurants either through a list of cards or on a map. Each restaurant's essential information (e.g., name, picture(s), etc.) should be presented in a clear way. These overviews can be refined by type of cuisine (such as Italian), price range, distance, and *guesstimated* delivery time.

The price ranges are as follows:

Symbol	**Description**	**Average menu price**	**Example**
€	cheap	< €10	Fast food, bakery dishes
€€	regular	€11 - €30	Comfort food
€€€	expensive	€31 - €60	Fine dining, fish menus
€€€€	premium	> €60	Michelin star level dishes

If a restaurant becomes more expensive, we'd like to see evolution over time as an extra service provided to the customers, in the form of a graph. Important is that we like to see the evolution to today's price standards. So if the average menu price categories get adjusted to inflation, the evolution graph will need to change as well.

In the diagram below, the platform admin changed the price range criteria each 2 months, you'll get the following outlook if you would plot this datawise. Important! The price range can be changed by the platform admin (and can be changed in the past as well!), so you'll have to look at techniques such as event sourcing to solve this use case.



'Guesstimated' delivery time is calculated as follows:

- *Delivery time*: calculation using the restaurant location and the customer's (browser-provided) location
- *Default preparation* time of a dish: provided by owner (restaurant-wide)
- *Busyness factor*: number of orders not yet ready for pickup for this restaurant

  Guesstimated delivery time = (*delivery time* + *default preparation time*) * *busyness factor*

Customers can navigate to a specific restaurant to view its details and dishes. The dishes can be filtered by type (such as starter or dessert), and by food tags (lactose, gluten, vegan,...). Provide sensible sorting options like price.

Customers build a basket from a restaurant's published dishes and review it before placing an order. A basket contains dishes from one restaurant only. If something changes while the customer is browsing—a dish is removed or becomes out of stock—the basket should make that obvious and block checkout until the basket is fixed. What the customer agrees to at checkout should not change afterwards. Once an order is placed, its contents and price are fixed.

When checking out, customers provide:

- Customer name
- Delivery address (street, number, postal code, city, country)

- Contact email

After checkout (this includes a payment provider integration), customers should be able to track the progress of their order by returning to a tracking view that reflects status changes.

Opening hours and closures determine when orders can be taken. The owner can also manually mark the restaurant as open or closed at any moment (for example, opening early or closing early). The customer experience should follow this immediately: if the restaurant is closed, customers can't place an order; if it opens early, ordering becomes available right away. Any order a restaurant accepts is assumed to be feasible within opening times.

When an order is submitted, the restaurant is asked to decide promptly. The five-minute window starts at the moment the customer submits the order. If no decision is made within five minutes, the order is automatically declined so the customer isn't kept waiting. If the restaurant rejects sooner, the customer should see why and can adjust the basket or try later. If the restaurant accepts, the kitchen commits to prepare the order.

After acceptance, the system waits for the kitchen to mark the order ready for pickup. That moment matters because responsibility can move toward delivery. An external delivery service operated by another company will rely on these updates to coordinate pickup, and will also inform us when the courier has picked up and when the order has been delivered.

Delivery integration (RabbitMQ)

To work smoothly with the external delivery service, we need a simple, shared way to name and route messages. Below is the agreement we'll use so both sides know where to publish and how to subscribe.

- We use a single topic exchange named kdg.events on RabbitMQ.
- Routing keys are versioned and follow this pattern:
  <context>.<restaurantId>.<resource>.<action>.v1
  Example: restaurant.42.order.ready.v1

Bindings and routing keys (delivery service consumes from us)

- Bind to kdg.events with: restaurant.*.order.accepted.v1
- Bind to kdg.events with: restaurant.*.order.ready.v1

Bindings and routing keys (we consume from delivery service)

- Bind to kdg.events with: delivery.*.order.pickedup.v1
- Bind to kdg.events with: delivery.*.order.delivered.v1
- Bind to kdg.events with: delivery.*.order.location.v1

Out of scope

You do not need to take into account: delivery zones, inventory/stock counts, discounts, loyalty programs, or multi-currency.

Use cases

#	Description
1	As an owner, I want to sign up/sign in to access my restaurant management area.
2	As an owner, I want to create my restaurant by submitting name, full address, contact email, picture(s) url, default preparation time, type of cuisine, and opening hours.
3	As an owner, I want to edit a dish as a draft without affecting the live menu.
4	As an owner, I want to publish a dish so it becomes available to customers.
5	As an owner, I want to unpublish a dish so it is no longer available to customers.
6	As an owner, I want to apply all pending dish changes in one action.
7	As an owner, I want to schedule a set of publishes/unpublishes to go live together at a chosen time.
8	As an owner, I want to mark a dish out of stock or back in stock immediately.
9	As an owner, I want to set opening hours and manually open/close the restaurant at any moment.
10	As an owner, I want to accept or reject new orders and provide a reason on rejection.
11	As an owner, I want orders without a decision within five minutes to be automatically declined.
12	As an owner, I want to mark an accepted order ready for pickup.
13	As a customer, I want the landing page to let me continue as a customer.
14	As a customer, I want to explore restaurants in a list or on a map.
15	As a customer, I want to view a restaurant’s details and dishes.

16	As a customer, I want to filter restaurants by type of cuisine, price range, distance, and guesstimated delivery time.
17	As a customer, I want to filter dishes by type (e.g., starter or dessert) and by food tags (lactose, gluten, vegan, …).
18	As a customer, I want sensible sorting options (e.g., price) when viewing dishes.
19	As a customer, I want to see a guesstimated delivery time based on location, default preparation time, and busyness.
20	As a customer, I want to build a basket from a single restaurant's published dishes.
21	As a customer, I want checkout to be blocked if any dish in my basket becomes out of stock or unpublished while I'm browsing.
22	As a customer, I want to provide my name, delivery address, and contact email at checkout.
23	As a customer, I want to pay using the payment provider during checkout.
24	As a customer, I want a confirmation with a link so I can return to track my order.
25	As a customer, I want to track the progress of my order as its status changes.
26	As KDG, I want to be able to see the price range evolution of a restaurant
27	As KDG, I want no more than 10 dishes to be available to customers at any moment.
28	As KDG, I want to publish messages for the delivery service when an order is accepted and when it is ready for pickup.
29	As KDG, I want to consume delivery service messages for picked up, delivered, and courier locations to update orders.
30	As KDG, I want customers to be able to order without signing up or signing in.
31	As KDG, I want each owner to manage exactly one restaurant.
32	As KDG, I want to adjust the price ranges criteria

Technologies and techniques

All technologies and concepts handled in this course.

Backend:

- Concepts of Domain Driven Design with a rich domain, entities, aggregates, value objects, domain events
- A clear context mapping technique between bounded contexts
- Hexagonal Architecture with clear separation, boundaries, infrastructure agnostic uses cases
- Commands/events
- Event sourcing
- Projected data (derived state)
- At least 1 snapshotted event sourced aggregate
- Apply security to REST endpoints

Frontend:

- Use React with Functional Components (hooks) and Typescript
- Use an asynchronous state library like React Query
- Use routing for separate pages
- Use a component framework (like Material UI, etc.)
- Organise your code using a clean folder structure, components/pages, hooks and (API) services
- Secure the application with OAuth.
- Keep views that need frequent updates up-to-date using polling.

Project setup

A Gitlab group will be created for you with two empty repos (frontend, backend). You are required to work in this group. Your Git repos will be used as a single source of truth and should contain:

- A clean commit history (very important!)
- User stories and clean progress
- Wireframes
- Lightweight, to the point, documentation
    - Domain models
    - Command and event catalog
    - Make sure it is easy to read and understand

Backend repo

There will be a documentation folder containing your documentation (event and command catalog). A compose.yaml (Docker Compose) file will be provided as well in the infrastructure folder.

The compose.yaml file consists of the following services:

- A database for your application (PostgreSQL):

  This is a PostgreSQL database. Use this database for your application. The PostgreSQL database port will be mapped to 54321, check the service postgres_kdg_db for more information on how to connect with what credentials.

  Volumes are mapped under the infrastructure folder, make sure to not commit them by setting up a .gitignore file.

  This service connects via a custom docker network called backend.

- A message oriented middleware (RabbitMQ):

  This is RabbitMQ, the management port is mapped to port 15672, check the service service_kdg_rabbitmq. It will become clear during lectures how we are going to integrate with this service.

  Volumes are mapped under the infrastructure folder, make sure to ignore them in your .gitignore file.

  This service connects via a custom docker network called backend.

- An identity provider (Keycloak):

  This is an identity provider. Detailed documentation can be found here: https://www.keycloak.org

  This service is called idp_keycloak and it will be clear during lectures how we are going to integrate with this service. A special docker network is set up called kc.

  The management console is exposed at http://localhost:8180/auth/ - a nice starting point for documentation as well.

- A database for all things related to Keycloak:

  This is a PostgreSQL database as well, but cannot be used for direct interactions. It links to another docker network especially for keycloak called kc network.

  This service is called postgres_idp_db.

Frontend repo

- Create a new React project using ‘vite’ (see instructions during class)
- Configure git on the project and set remote to gitlab repo
- Start working: commit and push frequently (at least after each work session)