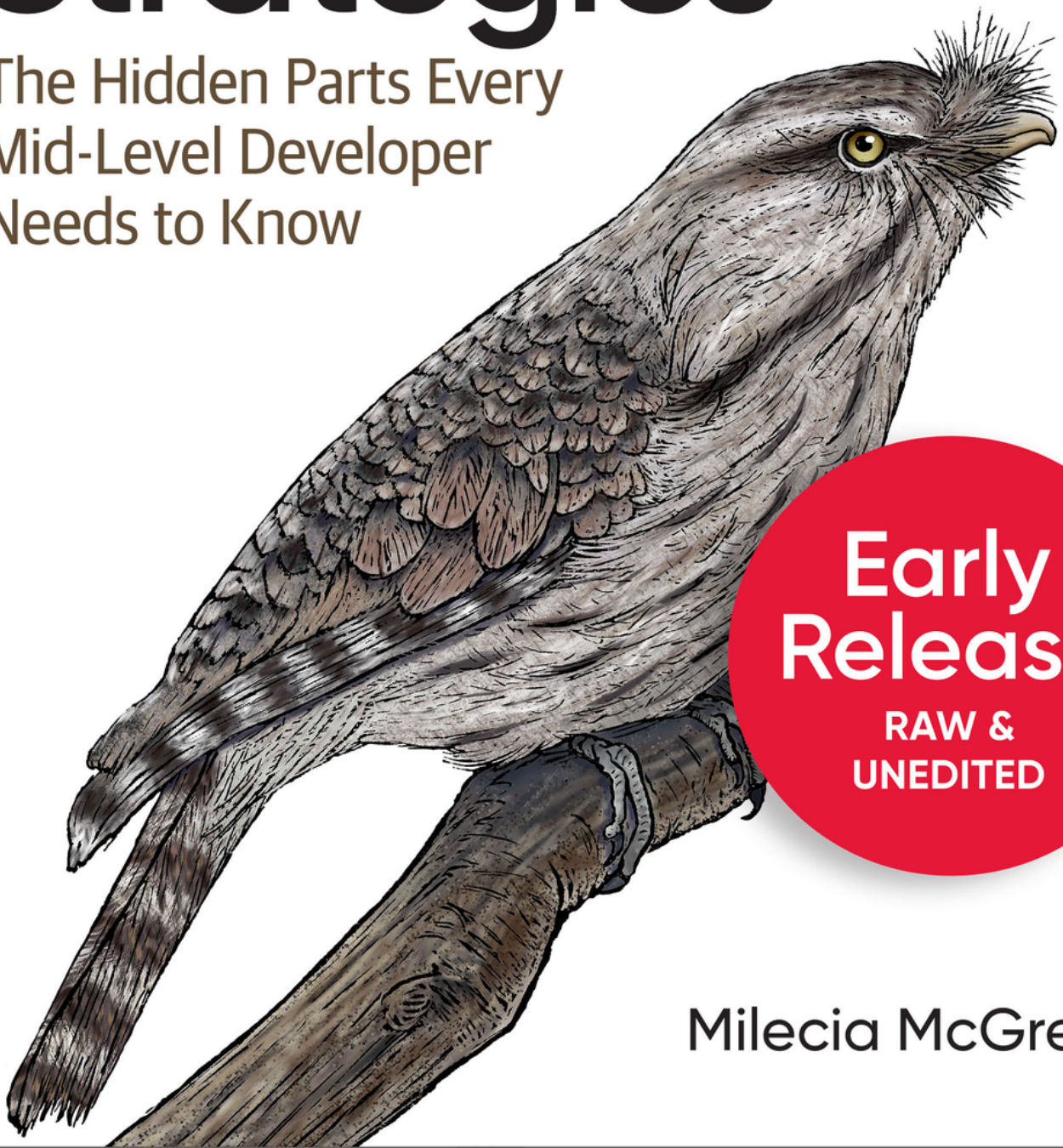
O'REILLY®

Full-Stack JavaScript Strategies

The Hidden Parts Every Mid-Level Developer Needs to Know



Early Release

RAW & UNEDITED

Milecia McGregor

Full-Stack JavaScript Strategies

The Hidden Parts Every Mid-Level Developer Needs to Know

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Milecia McGregor

Beijing • Boston • Farnham • Sebastopol • Tokyo

Full-Stack JavaScript Strategies

by Milecia McGregor

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Amanda Quinn
- Development Editor: Virginia Wilson
- Production Editor: Christopher Faucher
- Cover Designer: Karen Montgomery

- November 2024: First Edition

Revision History for the Early Release

- 2023-09-11: First Release
- 2024-01-02: Second Release
- 2024-03-11: Third Release
- 2024-05-16: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098122256 for release details.

978-1-098-12219-5

Brief Table of Contents (*Not Yet Final*)

PART 3: Building the front-end

Chapter 12: Front-End Setup (unavailable)

Chapter 13: Make React App (unavailable)

Chapter 14: State Management (unavailable)

Chapter 15: Data Management (unavailable)

Chapter 16: Styles (unavailable)

Chapter 17: Error Handling (unavailable)

Chapter 18: Front-End Testing (unavailable)

Chapter 19: Front-End Security Considerations (unavailable)

Chapter 20: Front-End Performance (unavailable)

Chapter 21: Front-End Debugging (unavailable)

PART 4: Deploying the full-stack app

Chapter 22: Full-Stack Deployment Setup (unavailable)

Chapter 23: Integration Concerns (unavailable)

Chapter 24: Building a CI/CD Pipeline (unavailable)

Preface

My goal with this book is to give you a reference, kind of a sanity check, for when you're working on either greenfield or legacy projects across the front-end and back-end and handling deployments. Some questions are relevant for both types of projects, like how you'll handle testing, performance, and security. Many applications have core commonalities that you can use, regardless of the industry you work in. When those moments come where you find yourself questioning why you've never heard of something, hopefully this book will make you feel more confident asking those "simple" questions.

Who This Book is For

If you are trying to figure out how senior devs seem to magically know how everything works and how they understand complex concepts so quickly, I'm going to show you how. At this point in your career, you've probably been working as a software developer for a few years. You know how to complete your tasks with solid code regardless of whether it's on the front-end or back-end.

While you might have some knowledge across the full-stack, it's likely you have a focus on one part of the stack over the other. On the front-end, you should be familiar with things like making responsive layouts, fetching data from APIs, and some of the frameworks like React, Astro, or Svelte. On the back-end, you've done some database migrations, built some APIs, and handled some basic authentication flows.

You also have skills like using Git with any of the repo hosting services, like GitHub or GitLab, and using different tools to test your changes. You may have worked on one project for years or you might have hopped around, but the scope of your work has typically fallen under some of the concepts mentioned.

Now you're ready to move to the next level in your career. That means learning how the whole system works and why technical decisions are made. That's what will be covered in this book.

What You Will Learn

The high level of what you're going to learn in this book is everything it takes to create a full-stack web application hosted on a cloud platform. You're going to learn how to find where the business logic for every app comes from. You're going to gain some intuition on how to decide between different architectures, third-party services, and tools to create a maintainable application.

You are going to learn the subtle, yet deep skills it takes to become a senior developer. This book is going to give you strategies for working with different teams and understanding how product decisions are made. You'll also deepen your existing skills to cover the full-stack confidently.

Regardless of whether you start on the front-end or back-end, you're going to learn about design and development principles and when to apply them. You'll learn all of these things as you go through the software development lifecycle (SDLC) of a project in this book and build a production-like application.

Think of this book as a reference you can pick up at any point in the SDLC and use it as a checklist. You will learn all of the essential parts that need to be considered for app development so that you know exactly what to do when a question comes up. By the time you finish, you'll know how, why,

and when to make technical decisions and how the business requirements evolve.

What This Book is Not

This book is not a deep dive into any specific set of tools and it will not teach you general JavaScript programming. There will be a large range of topics covered in this book with accompanying examples to demonstrate senior-level considerations, but it is expected that you know how to read code, debug issues, and find additional learning resources.

Since there are so many topics covered, there will be strategies discussed along with the code. These strategies are meant to be tools you can bring to any project you work on, although they might not work on *every* project. There isn't a single approach that would work for any two projects because everything has its own nuances. So the goal is to give you a number of options you can choose from as needed.

There will be some parts of the book that need a much deeper explanation than a chapter or section can provide and there will be links to other resources. No book can adequately cover all of the topics presented here and I want to make sure you get all of the information you need. So while some topics will be light on the full implementation details, there will always be links to complementary resources.

How This Book Is Organized

The first part of this book, *Starting your new project*, will be relatively short and cover how you translate designs into tasks and questions that you'll have to ask the Product team and other teams. This is where you get your first introduction to where business logic comes from.

The second part of the book, *Building the back-end*, focuses on creating the back-end of this project. You'll work with Nest.js as we walk through a number of considerations, like security and third-party services, for development.

After you have the back-end ready, you'll switch over to part 3, *Building the front-end* where you'll work with a React project. You'll build the user interface (UI) for this project and cover concerns associated with front-end apps, like responsiveness and performance.

Finally in part 4, *Deploying the full-stack app*, you'll dive into the details of connecting the front-end, back-end, and other systems to build and deploy a full-stack app to production. By the time you finish this book, you should feel comfortable jumping into any part of a project and asking questions that will help clarify tasks as well as provide technical advice.

You'll notice that each part of the book varies in length. That's because it's supposed to help break down where a lot of time really gets spent during

development. Some parts of a project take more time than others or have different feedback cycles. This book is trying to reflect real-world conditions as closely as possible.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`*Constant width*`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`*Constant width italic*`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/flippedcoder/dashboard-server*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does

require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Full-Stack JavaScript Strategies* by Milecia McGregor (O'Reilly). Copyright 2025 Milecia McGregor, 978-1-098-12225-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments,

and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-829-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://www.oreilly.com/about/contact.html

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Chapter 1. Kicking Off the Project

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

Welcome to the team! If you're like most readers of this book, you're a front-end or back-end developer, but you would like to be a senior level full-stack developer. Now imagine fast-forwarding to that point in your career—you're a new senior full-stack JavaScript developer. That means you're responsible for helping drive the technical direction of the project and helping coordinate efforts across different teams in order to get stuff done. Your task is to start a greenfield app, a project you get to build from scratch with no existing code, based on designs and conversations with a number of teams.

Let’s break down what you’ll learn in this chapter:

- How to work with a Product and a number of other teams to fully define features
- How to take a design and break it into small, actionable tasks
- How to determine the data you’ll need to work with

The Project You’ll Build

Here’s the description of the project you’ll be building as you go throughout the whole book. You work on an e-commerce platform that has access to a lot of user data, including some *personal identifiable information* (PII), and connects with multiple third-party services. You and your team have been tasked with creating a user dashboard so customers can see information about their order history and other actions they have taken, make purchases, and interact with their digital purchases, like ebooks. They’ll also be able to take different actions based on their permissions within the app

You’re starting with a fresh Product team and possibly a Design team that you’ll work with to make this dashboard a reality. The Product team will be responsible for talking with different stakeholders to decide what features should be built. The Design team will be responsible for making the user interface (UI) for those features and doing the first pass at the user experience.

The first part of the project will likely involve the Product team and the Design team working closely together to make some mock screens for you and possibly some behavioral docs. These mocks will be shown to you and discussed in a project kick-off meeting.

Project Kick-Off Meeting: Going through the designs

This kick-off meeting typically involves the Product team going through the mocks the Design team has made with you. There's a strong chance that these mocks will change as you go through development, but they should be about 80% of the way there. They'll take you through the user flow, show you how users interact with the app, and give you an idea of the data you need. At this point in the project, get comfortable asking a lot of questions. There are usually things that come up when you start looking at the designs from a technical perspective that the Product or Design teams haven't considered.

Remember, this whole process is very collaborative. Asking questions early and often will make things smoother as the project moves forward. Right now there are only two screens that need to be built for a proof of concept, but Product has told you the functionality will expand over time. Let's take a look at these screens and go through the user flow.

The screenshots are from a commonly used design tool called Figma.

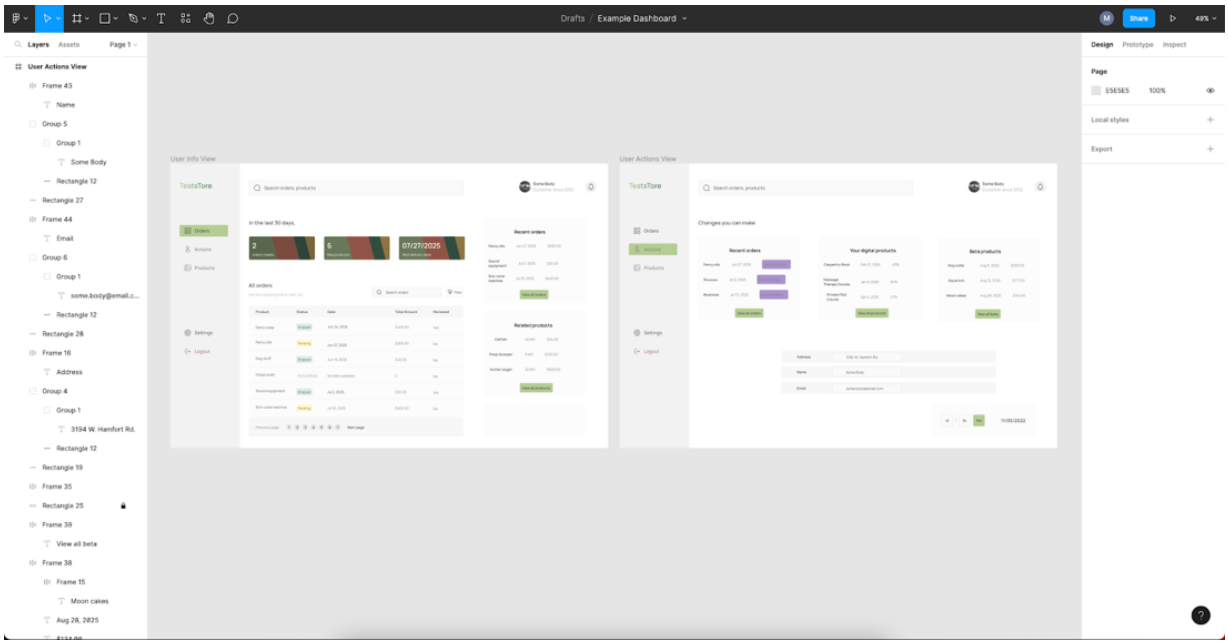


Figure 1-1. : Overall design in Figma

Figure 1-1 shows what all of the views in the application look like. There's usually a design like this somewhere that visually describes how the app flows. As you move through the engineering process, you'll focus on one of these views at a time, like the user info screen in Figure 1-2.

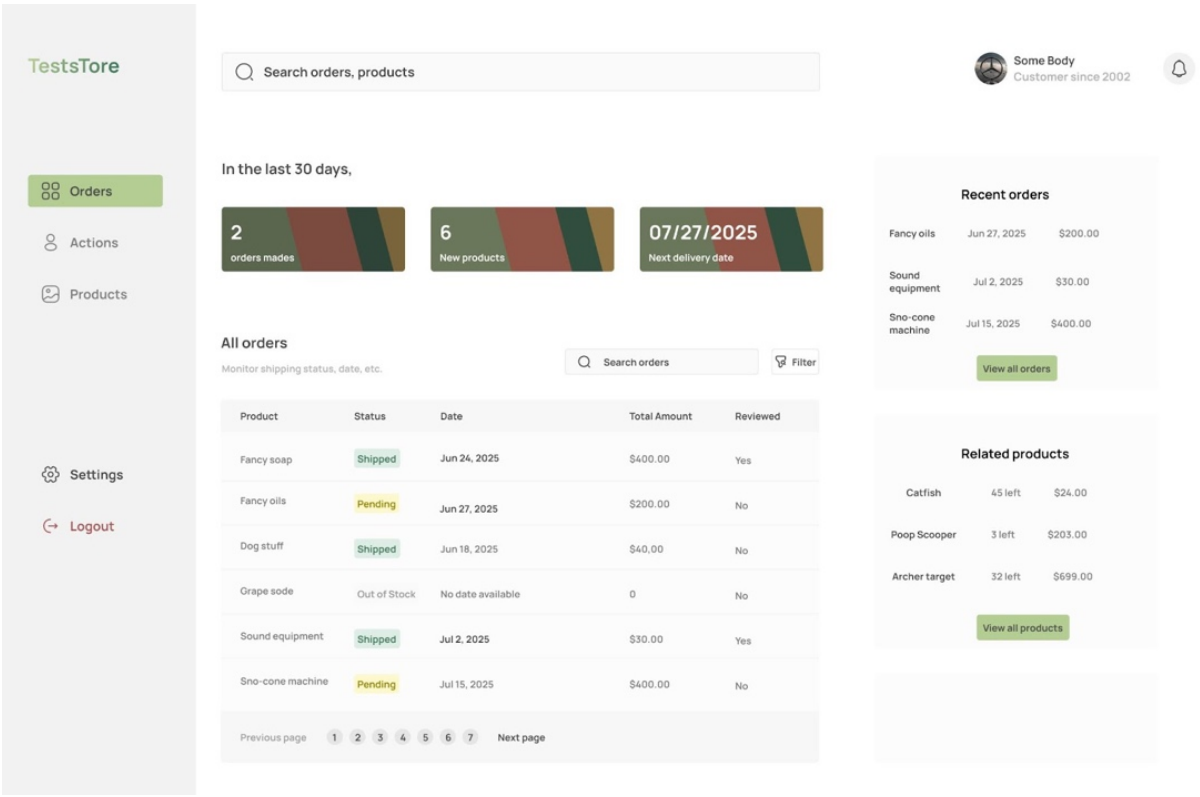


Figure 1-2. : user info screen design in Figma

Figure 1-2 is where users can see their information, like orders, suggested products, and other details.

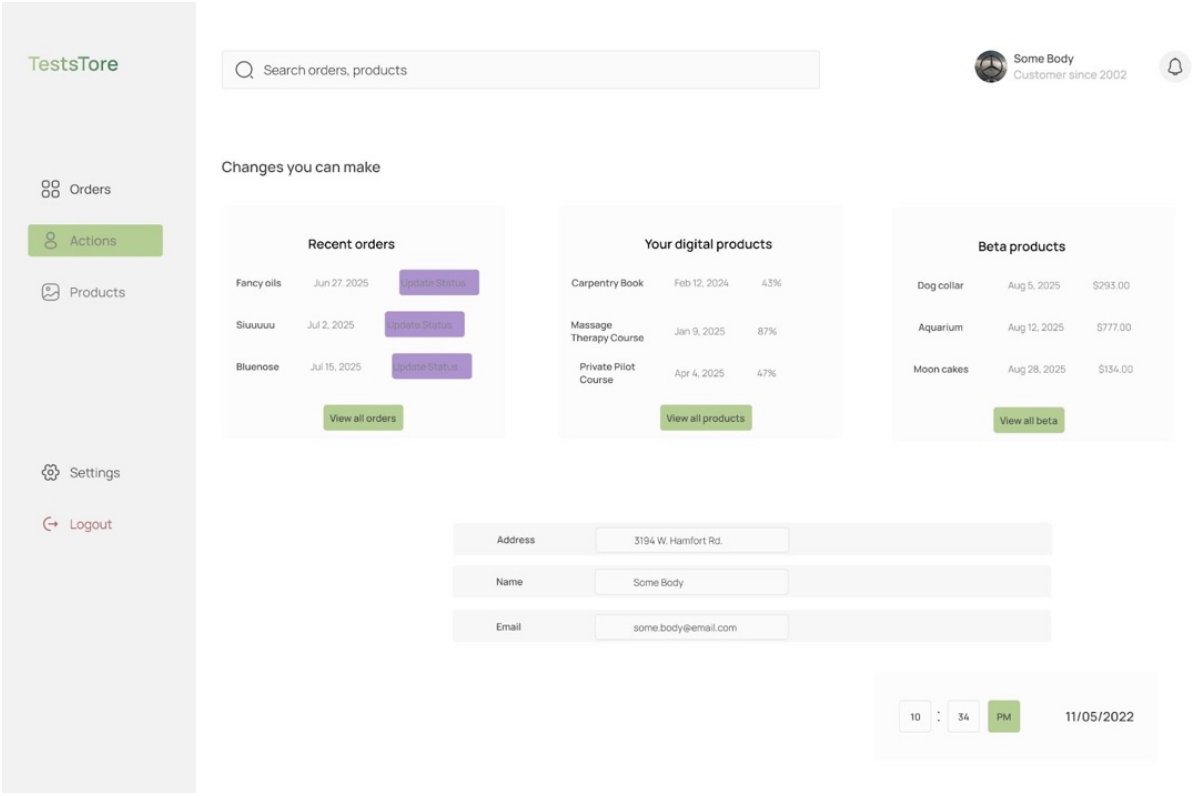


Figure 1-3. : user actions screen design in Figma

In Figure 1-3, users edit their information like address, cancel orders, and see their digital purchases.

Some Body
Customer since 2002

Search orders, products

In the last 30 days,

2
orders mades

07/27/2025
Next delivery date

All orders

Monitor shipping status, date, etc.

Search orders | Filter

Product	Status	Date	Total Amount
Fancy soap	Shipped	Jun 24, 2025	$400.00
Fancy oils	Pending	Jun 27, 2025	$200.00
Dog stuff	Shipped	Jun 18, 2025	$40,00
Grape sode	Out of Stock	No date available	0
Sound equipment	Shipped	Jul 2, 2025	$30.00
Sno-cone machine	Pending	Jul 15, 2025	$400.00

Figure 1-4. : app designs with mobile view in Figma

The design in Figure 1-4 shows how these views should look on mobile devices. The majority of projects that interact with external users will have mobile designs. If you don't see these designs, make sure you ask about them!

A lot of designers use this tool when they create the mocks for an app or a particular feature. Some other tools you might run into are inVision or the newer alternative, Penpot. These Figma designs also have the HTML and CSS in them, so they can be useful for developers when they want to check dimensions while arranging elements on the page. Although the CSS is there, it's usually best not to copy it directly into your code. It doesn't always generate style rules that fit with your existing theming.

By the time you see these mock designs, Product and Design will have had several meetings trying to iron out as much detail as they can. That doesn't mean they'll have everything figured out and that's where you come in. Based on these designs and the explanation of how a user will interact with the app, you have to work with your team to decide the technical approach you want to take.

Let's walk through these designs, similar to how you will with your Product team.

1. On the first screen, there’s a navigation bar at the top of the page with links to other areas in the app.
2. Then, there is a featured products section, along with the users most recent order.
3. There’s a sortable table that displays all of the user’s orders with several columns.
4. Finally, there’s a footer that includes links to information about other parts of the e-commerce platform.

On the second screen, there are several sections that have fields the user can interact with. There are a few settings users can toggle, fields that they can update, and actions they can take based on their purchases. This screen has more complexity than the first screen, so you’ll want to take time and really go through this with the team. As the senior developer on the team, Product and Design will often turn to you for the final say on technical decisions.

NOTE

Never make promises without talking to your team first, no matter how hard you get pushed. The Product team may pull you into a meeting to try and get an idea of how complicated it will be to implement a feature they have in mind. This can lead to them expecting a quick commitment on a timeline. This is something you will have to get comfortable with handling as you move to senior dev level.

You have to firmly state that you can't give an accurate estimate until you talk to the rest of the team. The other devs will be able to bring up concerns that you might not realize and they'll be working on the feature eventually anyways. Stand your ground and simply don't commit if they keep pressing for a date. That way *you* aren't the one who put the team on a tight deadline.

With these designs in hand and a few documents on how the screens are supposed to work, it's time to develop an understanding of the business logic before you go to the team. Having a solid understanding of the business decisions behind the designs is a huge help when you're trying to explain things to other developers.

As you go through this information, there are a lot of things you'll need to consider based on the designs you have.

Design Considerations

This is a stage where you will find yourself asking a lot of questions and that's totally normal. Now is the time to really dive into the details and get nitpicky. One of the ways you can get more experience with evaluating

designs for technical purposes is by getting exposure to different design sets. For example, the Design team you're working with is likely doing design work for other apps in the company.

If you have questions about how a dropdown is supposed to behave, ask if there are other designs you can use as a reference. Many times there are and you should take the time to thoroughly understand how they work. Some of the best questions come from experience with different implementations. Another way to get more exposure to different designs is to start paying close attention to the apps you use! Have you ever *really* thought about the user flow for your bank app or the flow for an exercise or meditation app? These are all examples of product designs that are in production that you can make comparisons with.

NOTE

It can be a little tricky getting time to do exploratory work. One way you can do that is by writing a ticket that details what you'll be looking into, some of the thoughts you've had about the issue so far, and some open questions. Go ahead and add points to the ticket too to make it look more official.

Think about how designs look for other products and how interfaces behave when you're breaking them down into the details. As you think of differences, take notes about the questions they bring up.

WARNING

A word of caution for when you're compiling your list of questions: As you think of a question, look through the documentation you were given. Sometimes the answers are already there and it saves everyone time when you read the docs first. Take about an hour to go through any engineering docs the team has for other projects, go through the docs on third-party services that are being used, and any packages that you're thinking about using.

You might be wondering what some good questions are and they always depends on your designs and the business logic. Here are some that can get you started as you look at the screens you have available:

- Are there translations to other languages we need account for?
- What are the brand guidelines, like colors, fonts, and responsive sizes, that should be used throughout the app?
- What permissions do users need to see their information?
- What permissions do users need to take actions in the app?
- How many different types of user will there be?
- How long do we expect users to be active, so we know how long an access token should live?
- What is all of the data we expect to handle for users?
- Do we want to show this data for different time ranges?
- Should the table be sortable by each column or only specific columns?
- How do we know which orders should be featured?
- What type of data do the input fields accept?

- When a user saves changes, do we want to redirect them to the previous page?
- When a user saves changes, what should we show them?
- If there's an API error or client-side, what should the user see or be able to do?
- Does the user need to enter more credentials before taking sensitive actions, like changing PII?
- Should input errors be shown inline with the field or by the submit button?
- Do we need to consider tablet-sized devices?
- Will the back-end of this app be used by any other apps?
- Will the front-end need to be integrated into another app?
- Are there any compliances or regulations this app falls under or will get audited by?

You see, this list of questions can get very long and I'm sure you thought of other questions as well. It's ok if the list is long. Of course, it'll be hard to get all of these questions addressed so you should definitely prioritize this list based on the feature. It's better to ask as many of these things in the beginning compared to half-way to the deadline. If there are small things then you can probably use your experience to make judgement calls, like with form handling.

The questions can be all over the place, but these are questions the Product or Design team can answer. There isn't anything technical in here yet,

although the answers to all of these questions will drive technical decisions. Getting answers to these sorts of questions early is why senior developers tend to deeply understand the business logic.

In order to make the best technical decisions you can, you have to understand the purpose of functionality and how it fits in the overall business. This can give you some foresight on how to build the app in the most maintainable way because you see the direction the business is going. Some of this foresight will come from the data you know you're going to work with.

Data-driven Design

Data drives design. Everything the designs do is built around how to best deliver data and let users interact with it. Take the table screen for example. The data doesn't have to be displayed as a table. It could be a collapsable list or a graph. That's why you have to dig into these questions about the business logic and what you will be working with. It's one of the reasons that front-end development is usually dependent on back-end development. We'll talk about this more in the next chapter though.

As you go through the designs, it's also a good idea to take notes on the data you see you'll need for each screen. On the settings screen, you can see that you'll need the user's name, shipping address, and language preference

in one section. Take notes on potential variable names and data types for these that you feel comfortable sharing with the team. It can help spark the initial conversations around how to approach the project. Also confirm with the Product team that these values are correct and what's expected by the app. Always check with the Product team that you have the correct data and relationships between the data.

They won't explain it in technical terms, but they'll walk you through different scenarios so you can paint the big picture. This is like the discovery phase of the project for your development team. Once you've gone through these designs and you have your questions and notes, get ready to talk about specific features.

Breaking Down Designs into Tasks

You've made it through the first round of refinement for the project! In this next round, you can assume that Product has answered all of your team's questions and you have updated designs. Now you'll be working with Product to carve features out of this info. This is when you'll first come to the a kanban system, like Jira, Trello, ClickUp, or Shortcut.

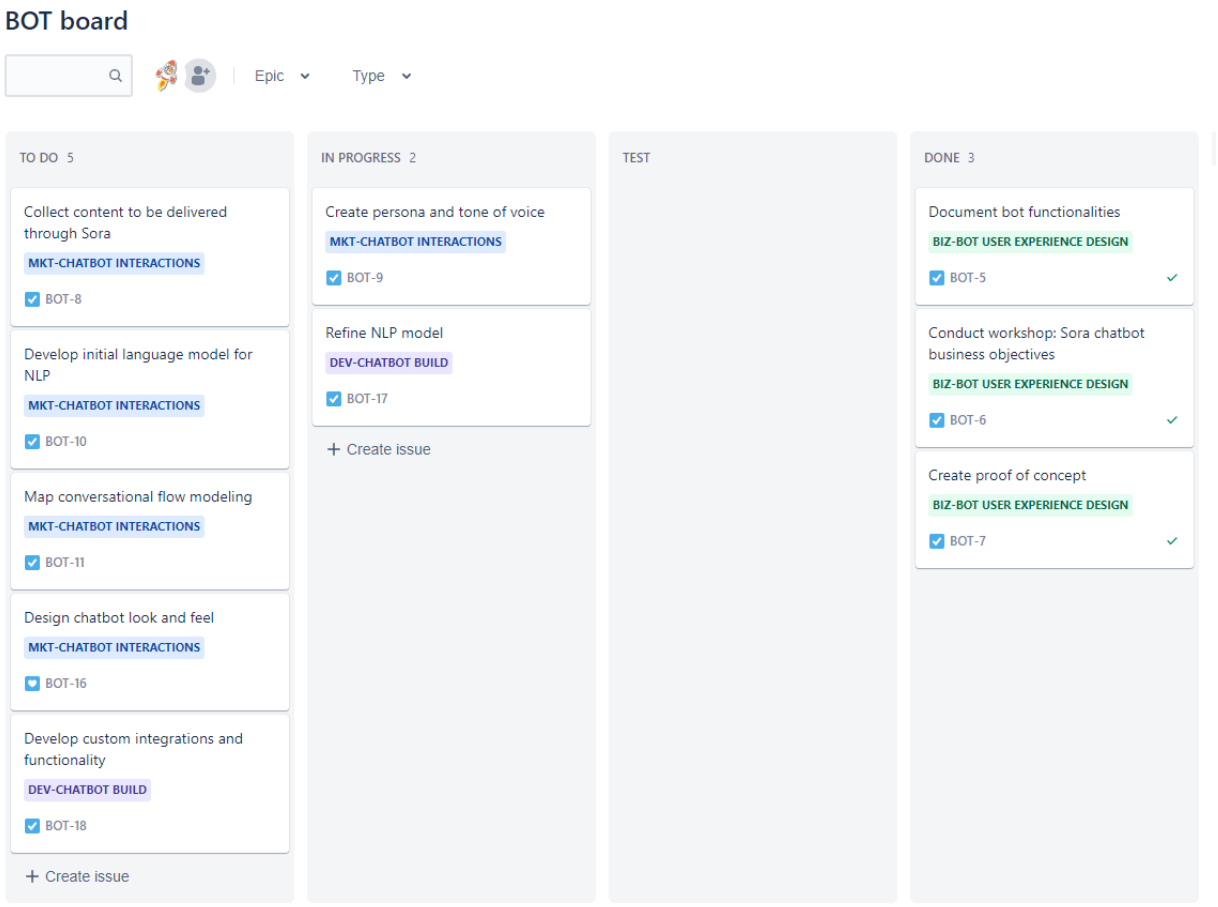


Figure 1-5. : example of a backlog in Jira

The tickets you make here are going to go through refinement once you start bringing in more developers and other teams. The goal at this stage is to make action items for segments of the designs. The user info screen for example has several pieces that can be broken out into features. The featured section and the table are individual features. You're trying to get people thinking about how the designs will actually work and start bringing in that technical perspective. As a senior full-stack dev, you might try to group features based on data relationships.

This will help make it easier for back-end changes to be made and deployed as the front-end is being built. This is also the point where you can still call out any big issues you run into as you start thinking through the tech stack. Any time you have a thought for a feature or technical requirements, make sure a ticket gets made. If there isn't a ticket, the work won't get prioritized.

Refining Tasks from Feature Requirements

You're at the point in the project where Product should have some user stories written for you to refine with technical details. These user stories are part of the documentation you get that define the features you're developing. They're typically written from the perspective of the user to give you some of the hidden context behind how they will interact with the functionality. This doesn't happen all the time. Many times you'll be working with Product to fully define features, but they'll have some details to get started with from the previous round.

You can assume that your team follows a normal agile process with two week sprints and the standard meetings like daily standup, backlog grooming, sprint planning, and retro. At this point, you'll go to a backlog grooming meeting and start going through the tickets in the Jira backlog. This meeting should include the development team, QA team, and Product team so that everyone can ask questions and bring up concerns. Feature tickets should be broken down into the smallest piece of work a developer can do.

This is one of the parts where you have to use some art vs science. The “smallest” piece could mean having a ticket to write an endpoint with all the tasks associated with it and have a separate ticket for triggering events. One way to think about it is from a QA testing perspective. What is the smallest grouping of work that makes sense to test together? There are entire books and numerous resources written about agile and these kinds of project decisions. One resource you might look at are the Atlassian articles on agile.

In order to keep expectations aligned with what the team can output, estimates are usually made on every ticket for how complex the task is. When you’re going through tickets during backlog grooming, don’t hesitate to slow things down. If you spend the entire time on one ticket, then that shows Product where they need to improve and it can be brought up in the retro meeting.

Make sure every ticket has acceptance criteria that’s agreed on by Product, QA, and your team. If the ticket has UI elements, make sure you have access to the designs for both desktop and mobile. Every ticket should also have background information on how the functionality fits in the overall app so developers have context. All of the developers should be in agreement on estimates for each ticket.

This might seem like a lot of fuss over tickets, but doing all of this upfront saves your team time. Getting everyone on the same page in the very

beginning helps keep your team unblocked. With this level of documentation in each ticket, there shouldn’t be ambiguity in the task when anyone on the project gets to it. Here’s an example of info you will find in a well-defined ticket.

[BACK-END, QA] CREATE ENDPOINT FOR TABLE DATA ON USER INFO SCREEN

Description: As a user, I need to see a table with my purchase info. This info should include the product name, price, estimated arrival date, and quantity in each row on the table. This table should also be sortable by clicking on the column headers. It should have pagination so I can go through my purchase history for the previous 12 months. It should match the designs attached. (See fig 1.1-2 and fig 1.1-4)

Acceptance criteria:

- Update data schema to include purchase history definition: product name, price, estimated arrival date, quantity
- Run migration on database
- Create endpoint that responds with user purchase history for past 12 months
- Ensure authentication is working correctly
- Implement validation on request data
- Write tests for new endpoint
- Write docs for how the endpoint works so front-end can use it

Points: 5

It should be very clear to any engineer what needs to be done, although it shouldn’t tell them how to implement it. That’s something they can decide

after the team’s discussed it. Having tickets written out like this will spark deeper technical discussions that will help drive the overall architecture of your project.

This is when you start to take the tasks from Product and refine them even more as a development team. It’s always a good idea to include any highlights that come out of team meetings in the tickets so everyone remembers why things were done that way. You might learn that one of your third-party services only accepts data in a certain format and it leads to you needing to write some hacky code. But you absolutely have to have this because of a product requirement. That’ll be important context to have for a code review and for testing.

It might seem like this is a long process, but usually you can define the tickets in the upcoming sprint in a few days. Then once the sprint starts you can jump in. But before you get to pick your own tickets and start coding, there are a few more conversations you’ll need to have.

Discussing Timelines with Product and Other Teams

At this point in the project, you can assume you have well-defined tickets and the details are as ironed out as possible. Now it’s time for one of the

higher level conversations. After all of the tickets have been estimated, Product will be ready to set a launch date.

NOTE

Sometimes Product will come to you with a launch date before you've had a chance to estimate tickets. If you find that their timeline is shorter than engineering needs to adequately get through the tickets, push back. Always speak up when timelines are tight. This will save everyone from unnecessary stress.

This is going to involve more than just your team. You might need to coordinate with developers on other projects, the DevOps team, the Support team, and the QA team. It just depends on how far the reach of your changes goes. Don't commit to a launch date until you've talked with these other teams. You'll want to see how your goals fit into theirs and what you can do to make the process easier for them.

Talking to Other Development Teams

Your project might depend on another team implementing something in their code that your app consumes. You'll need to give them plenty of heads-up and details so they can include this in their sprint. Development teams aren't always aware of when feature releases for one app crosses over into theirs. Talk to them about what you need and keep it as simple as possible. Mention what your target launch date is and see if that's something they can help you reach.

On the other hand, the new features in your app might cause breaking changes for other teams and you need to let them know what updates they need to make and by when. Don’t hesitate to reach out to any teams you think will be affected just to double check. As you find out which teams are affected, update the team docs to show them as a consumer of your app.

Here are some things you’ll need to have ready before you talk to them:

- Well-defined technical requirements. Give them a task that’s just as defined as if your team were doing it. Also give them time to ask questions if they need more definition.
- A general idea of where in their code the changes need to be made. You don’t have to go in their code and implement the changes, but havinng a clue of where to get started helps.
- Context behind the change you’re requesting. A seemingly small change to you could change the functionality of something important in their app. Make sure they understand why this change is needed.
- A deadline. Of course you’ll ask how long it’ll take them to get to it, but have a flexible date in mind.
- Willingness to do the work. If it would be faster for you to implement the change and the team review it, be open to that as well.

When you know you have everyone across the development teams on the same page, then you can talk to the next set of people. These discussions

don’t need to happen in this order, but they do need to happen. Whoever is available to talk first can be the first conversation you have.

Coordinating with DevOps

When it’s time to release to different environments, you’ll need to check in with the DevOps team. That’s assuming there *is* a DevOps team. Sometimes at smaller companies, you will also be responsible for handling deployments. Throughout this book, you can assume that you’re working with at least one DevOps engineer that handles infrastructure and pipeline changes for you.

Since this will be a completely new app, you’ll need the infrastructure to be set up for QA to do testing, for production, and any other environments. You may be able to help set up the continuous integration and continuous deployment (CI/CD) pipeline using tools like CircleCI, Jenkins, or GitHub Actions. Although when it comes to provisioning resources and handling infrastructure concerns, that will be completely up to the DevOps team.

They’ll give you important information like which cloud provider the app will be hosted on. You’ll give them important information like the programming language and frameworks being used. The goal for these two teams on a new project is to at least have a couple of environments up for testing before prepping for production releases. Getting DevOps involved early is going to make things go smoother as development progresses.

Remember, all of these teams you're interacting with already have their own priorities. So you have to take into account that they may not get to you right away. That's why you want to coordinate with DevOps as soon as you have a technical direction. The last thing you want is to put DevOps in a crunched position right before an important deadline. Be ready to jump into a call to go over the project with them so they understand what performance and usage is expected from this app. During your call, there are several things you want to cover:

- Work on developing playbooks, the strategies and steps you implement to resolve issues, for outages and other issues in production. In the moment, it's hard to think and figure out best steps. That's why you and DevOps need to think about this ahead of time.
- Determine the right number of environments for the app. Some projects can have three environments while others have an environment for every Git branch in existence. Find the right balance of environments that is maintainable on the infrastructure side and flexible for testing and other uses.
- Figure out the best deployment strategy. It's important for DevOps to know when they need to trigger scripts to run and how often they will run. Some parts will be automated and other parts will need to be run manually. The best strategy is one that works for these two teams.
- Decide how to handle CI/CD maintenance. Most CI/CD tools use YAML files to define the configurations for a pipeline and these config files live in the project's repo. Sometimes developers make the

changes, DevOps makes the changes, or both feel comfortable making changes. Just get everyone on the same page.

- Agree on the versions of the JavaScript runtime, like Node, Deno, or Bun, that will be supported. Servers aren't always running the latest version of your runtime and that can cause weird bugs in your app. Know the oldest version of the runtime your project will work with, just in case.
- Find out when credentials are cycled. Some apps are dependent on keys and secrets that are maintained as part of the infrastructure, like NPM tokens. Add a reminder ticket to Jira so you can plan to update them before they expire and cause production issues.
- Choose a location to store assets. For some apps, this will be included as part of the cloud provider and for others a third-party service will be used. Even if you don't come to an absolute answer right now, make another Jira ticket to come back to this.
- Set up time to test the app together before deploying. You know how the app should behave so DevOps will lean on you to handle some of the environment testing. They can usually get the app running, but you have to verify it's in the correct state.
- Reserve some time for after the initial app deployment. Things always come up with the first few deploys that need to be fixed. Go ahead and put that meeting on the calendar. If nothing happens, you just reserved celebration time for everyone.

One thing to remember while you're talking to DevOps is that you won't understand everything they're talking about and you don't need to. You'll start to pick up on things as you work with them and it wouldn't hurt to pair with them a few times to see what they do. But even if you don't want to dive into DevOps world, that's fine. They'll handle everything on their side from here.

Now you need to talk with the QA team.

Working with QA

While developers, especially seniors, are expected to thoroughly check their work, sometimes bugs slip through. That's why QA teams exist: to make sure those bugs don't reach production. Weird integration issues happen, PRs overwrite functionality, a number of things cause bugs that developers don't catch. Getting new features and bug fixes to QA in a timely manner is important to staying on time for releases. You need to get the new code to them, give them time to test and report back defects, then give you time to fix the defects, and send it back to QA.

NOTE

There is a chance that you might not work with a QA team at all. It really depends on the company. Usually in early-stage startups, you won't have a dedicated QA team. It'll just be you and the other developers, maybe even the CEO. In other companies, you might see the Product team do some manual QA on features. Regardless of whether there's a QA team or not, you can use the steps in this section to help do your own QA more thoroughly.

You see this loop can take quite a while. So to shorten that time and make sure the highest priority tickets are tested first, include QA in many of your development calls. There will be questions they have about features and bugs from a QA perspective that need to be understood. All of the test cases they write are basically like the history of the app. Many features will have test coverage, whether it's automated or manual. These tests will exist even when the feature doesn't, so QA keeps record of changes in a different way than other teams.

Set up a call between the QA team and your dev team and make sure you've talked about:

- Automated integration tests. This is a responsibility that's usually owned by QA, but since the tests live in the project's repo in many cases, developers can write these tests as well. Also, make the distinction between integration tests and unit tests. Developers should always be the ones writing unit tests.

- Level of detail in reproduction steps. There’s a balance between not enough detail and way too much. Discuss what works best for developers and QA engineers so that everyone knows what to expect. You’ll end up working with a QA engineer to hone in on the issue anyways, but it helps to have a starting point.
- Include them in relevant calls like backlog grooming and stand ups. This way they’ll be up to date with any upcoming deadlines, shifted priorities, or releases and they can give updates if they’re running into issues. Try to put any updates for them at the beginning of these calls, like tickets they can test, so they can use their time the best.
- Make it clear that QA should feel comfortable reaching out to the developer that worked on a ticket they’re testing. Developers should be willing to pair with QA to help them determine if an issue is a bug or something else like an environment difference.

QA is everybody’s best friend because they find a lot of bugs that would have caused issues on production. It’s not good for anyone when there are issues on production. They aren’t there to tell the developer they’re wrong or that their code is bad. They just want to make sure absolutely nothing makes us bring out those playbooks for production fires. Be patient with them as they report defects because they’re just doing their job.

You’ll interact with a number of teams once you’re ready to deploy changes and that will really depend on the company. You might interact with the

sales team so they understand what the product is capable of and they can set realistic customer expectations.

There’s also the integration or customer success team. These teams usually work with customers 1-1 to help them integrate the product into their systems. They definitely need to be aware of any breaking changes or new features.

Some companies have a dedicated security team as well, especially if the product is in the FinTech, healthcare, e-commerce, or manufacturing. This team may come in right after QA and run security tests and there might also be continuous security scans happening on the app throughout your development pipeline.

One team that will be around in some form or another is the Support team. They are the next and probably last team you need to talk with.

Planning with Support

These are the people who are in direct contact with users who are experiencing problems. Some of the bugs you will fix come directly from Support. They might even link their support ticket to your bug ticket so they can keep users updated. The Support team is also a source of inspiration for new features. If they keep getting the same requests from different users, it’s worth looking into a solution.

Support does a difficult job of dealing with upset customers. You want to make sure that happens as little as possible. When you're talking with Support about an upcoming feature release, they need to know how it affects users. Product should have already talked to them and discussed timelines. You're trying to see what they need from the development team.

Keep in mind that the Support team usually doesn't have highly technical backgrounds. They understand how the app works from a user perspective and as an app admin, but they won't necessarily understand the weeds of the technical details. Understanding where they are coming from will help your communications exponentially. Your job with Support is to translate the technical stuff into the information they need.

Here are some common things that will help:

- Have documentation Support can refer to in order to use the new feature. Describe anything that has changed compared to existing functionality. Highlight values users will need to enter since user input causes many of the issues they receive.
- Listen when they bring up complaints with the app. Take in the feedback they give you from bugs or other requests and work with Product to prioritize it. That doesn't mean you implement everything they ask for, but it does mean you do a little research into why they're asking about it and how hard it would be to fix.

- Always sync up on release dates and times. Support will sometimes set aside special time around releases because there are going to be user questions. They need to know exactly when changes are going out so they aren’t blindsided with an increase in issues.
- Do a demo of the finished feature right before release. After the changes have been approved for release, do a quick live demo with Support (and really everybody else). That way they can see how it’s supposed to work and ask any questions.
- Dedicate extra developer time to Support around release days. If they start getting overwhelmed with issues, you might need to rollback the changes. Just be ready to pay close attention to them.

Discussing these things with Support is another way to bring up issues that can be addressed early in the process. It’s also a great way to build a relationship between Engineering and Support because you work closer together than it seems. This is the last team you need to coordinate with. Now it’s time to bring back all of your findings and notes to make a cohesive plan.

Bringing It All Together

You’ve made it to the final step of project kick-off. You have all of your notes from the different team meetings and it’s time to distill it all and give the highlights to Product. All of these conversations will help you come up

with a reasonable deadline. This is where some negotiation will start. One thing to keep in mind with deadlines is that something in development will go wrong.

Give yourself and your team room to breathe and possibly over-deliver. Remember, under-promise and over-deliver. Unless the feature you need to implement is time-sensitive, like a third-party service being updated, add a few days of padding to account for weird happenings. After you've settled on a date with Product taking all of the other teams' concerns in mind, summon everyone together for one final, short call.

NOTE

It's important to understand that setting deadlines is extremely tricky in software development. Any number of things can pop up at any time and derail everything. If someone gets sick or a natural disater takes out someone's electricity for a week, some tickets aren't going to get done. You might get deep into development and realize that a tool you're trying to integrate with is way more complex on the back-end than you accounted for. There might be work your waiting to receive from another dev team and they fall off schedule, putting you behind as well.

All of that is ok. The best thing you can do is communicate these things with everyone as soon as you find out about them. Then you can start making plans for what to do next. Life happens and you have to try your best to not let this stress you out every time it happens. Because it will happen a lot!

This call will have someone from all of the teams you've talked to. The only things you want to confirm at this point are everyone's action items and deadlines. Having this call is about establishing accountability and

triple-checking that everyone is on the same page. As prep for this meeting, take all of this info out of your notes and make a short doc that everyone can reference leading up to the release.

At this point, you have many of the Jira tickets you'll be working on and they're well-defined. You've documented the cross-team connections. Project kick-off is complete. You have everything you need to finally get started writing code. In the next part of this book, you'll build a scalable back-end in TypeScript using the Nest.js framework.

Chapter 2. Setting Up the Back-end

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

You've got all of your tickets ready to go and you can assume you've been tasked with building the bones of the back-end. The framework of choice at your company is Nest.js. It's ok if you've never seen Nest.js or even heard of it before. The docs for it are pretty good and you'll have other code you can reference. Reach out to other developers to see why certain architecture and design decisions have been made on existing back-ends.

In this chapter, you'll learn more about:

- How to make architecture decisions for the back-end

- Checking the app to make sure it runs after initial setup
- Writing initial documentation

These are all crucial to ensuring that development is able to run smoothly as both the product and the team grow and change over time. Everything that you build on the back-end will start from the foundation you set here, so take your time to thoroughly discuss options with your team.

Why Nest.js?

Before we dive too deep into the weeds, let’s discuss why Nest.js is a solid choice. It’s basically like a back-end architecture in a box. As soon as you initialize the app, you immediately have access to things like validation, authentication, routing, controllers, data schema, and a lot of other functionality. Instead of piecing everything together, Nest.js gives you everything you need to make a scalable back-end from the beginning. That does mean it’s a little less flexible in what you can do with it, but it’s the trade-off for all of the up-front scaffolding.

This is when you get to have some fun as a senior dev as you set up the initial repo for the back-end. The decisions you make here drive the direction for the future of this project. As you come to questions that have options, ask your team for their technical input. They might be aware of something you don’t know about. This not only helps you build a better app, it also helps everyone on the team learn.

There are so many considerations for building a back-end that having other input will help you make decisions faster and more confidently. It will also help to keep architectural decision records (ADRs). These help keep track of why and when important architectural decisions were made and you can see some examples in this GitHub repo.

It can be a little overwhelming being the one to open the empty terminal and bootstrap a completely new project. There's nothing to build on except examples and boilerplate code. It will go whatever direction you and your team decide to take it. This is a fun time because you can really set up the code base to be in a position that's clean to update, test, and expand.

Choosing a Project Approach

This project will be a monolithic back-end with a couple of serverless functions. This will allow you to build and deploy quickly and you'll get to work with a hybrid back-end architecture. You could have also decided to take a microservice approach. Then all of your APIs could be serverless functions, for example. Your choice of back-end architecture is dependent on how the app is going to grow over time. Microservices can take more resources and more engineers to maintain compared to monoliths. Although monoliths can make it more difficult to make larger changes as quickly as you can with microservices.

You'll also be using TypeScript. The reason you're starting the project off with TypeScript is to help prevent debugging because it's harder to write incorrect code. By strictly enforcing typing, you'll catch some errors before you even run the code. It also helps make it easier to see what fields and data types you're working with and where they come from, which is wonderful for keeping types consistent across the front-end and back-end. Having some typing in place is like having documentation for how things are connected and it lets you see all of the fields you have available.

Setting Up Nest

You'll walk through all of the steps to bootstrap this whole project. To start, there are a few environment things you need to have in place. You can follow along with the demo repo here: https://github.com/flippedcoder/dashboard-server

---

NOTE

Keeping track of these types of version concerns is something a senior dev might silently do. As we go through building this app, you'll be introduced to some leadership best practices, like keeping docs updated and making sure everyone's local environment is running correctly. You'll also have to consider things like linting with something like ESLint and formatting rules with a common tool like Prettier for the repo and any number of environment configs to keep everything consistent in the code for everyone.

You'll need to install the Nest CLI. The current version of Nest as of this writing is 9.2.0. Open a terminal and run the following command:

```
[source, bash]
npm i -g @nestjs/cli
```

You can create a new Nest project with the command:

```
[source, bash]
nest new <your-project-name>
```

For this project though, you'll use this command:

```
[source, bash]
nest new dashboard-server
```

It will ask you which package manager you want to use. Select npm and hit ENTER.

```
[source, bash]
? Which package manager would you ❤  to use? (Us
❯ npm
  yarn
  pnpm
```

After that the installation will start and it might take a few minutes. This process is installing all of the project dependencies, scaffolding the entire architecture, adding boilerplate code, and more. It even comes with TypeScript preconfigured. Take some time to go through this repo before you start making changes. Whether you're working on a brand new app like this or a legacy project, always take some time in the beginning to get a high-level understanding of how everything works.

Key things to look at are:

- The `package.json` so you can see which packages are used in the project.
- All of the config files so you know how things work under the hood.
- The test files because these give you a good idea of the major functionality of the app.
- Some of the actual code to see how developers are expected to write things and if anything can be improved.

After you've poked around a bit, go ahead and run the app with these commands:

```
[source, bash]
cd dashboard-server
npm start
```

You'll see an output in the terminal like this that will let you know the app is running successfully:

```
[source, bash]
> dashboard-server@0.0.1 start
> nest start
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [N
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [I
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [R
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [R
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [N
```

This is where one of your first senior decisions comes up. How exactly are you going to test this back-end? Having a consistent environment for development will change how quickly you can test code and how much frustration the team experiences as the product grows and the team grows too.

NOTE

A standard choice is to use Docker because you can setup everything from a Postgres database instance to the server that the app will run on in a container (or two).These containers can hold the exact same data and versions and be setup on any local environment, regardless of the operating system (OS). Another option would be using some functionality in VS Code called Dev Containers. This can be a good choice depending on the team and the project.

Testing the Back-end Locally

Some popular tools you'll see include Postman and Paw. You'll work with Postman in this book (and many times in real life) because it has a free version. This is how you'll be able to make requests to your endpoints with different headers, body values, and other values and see what response you get. That's typically how you test the back-end before there is a UI available.

Remember, tools like this are here to help you move faster. Don't get caught up in all of the tool options. At the end of the day, it doesn't matter what you use for your local development as long as you're able to make meaningful progress. Now that you have Postman available, make a GET request to +http://localhost:3000+. You should see +Hello world!+ as the response.

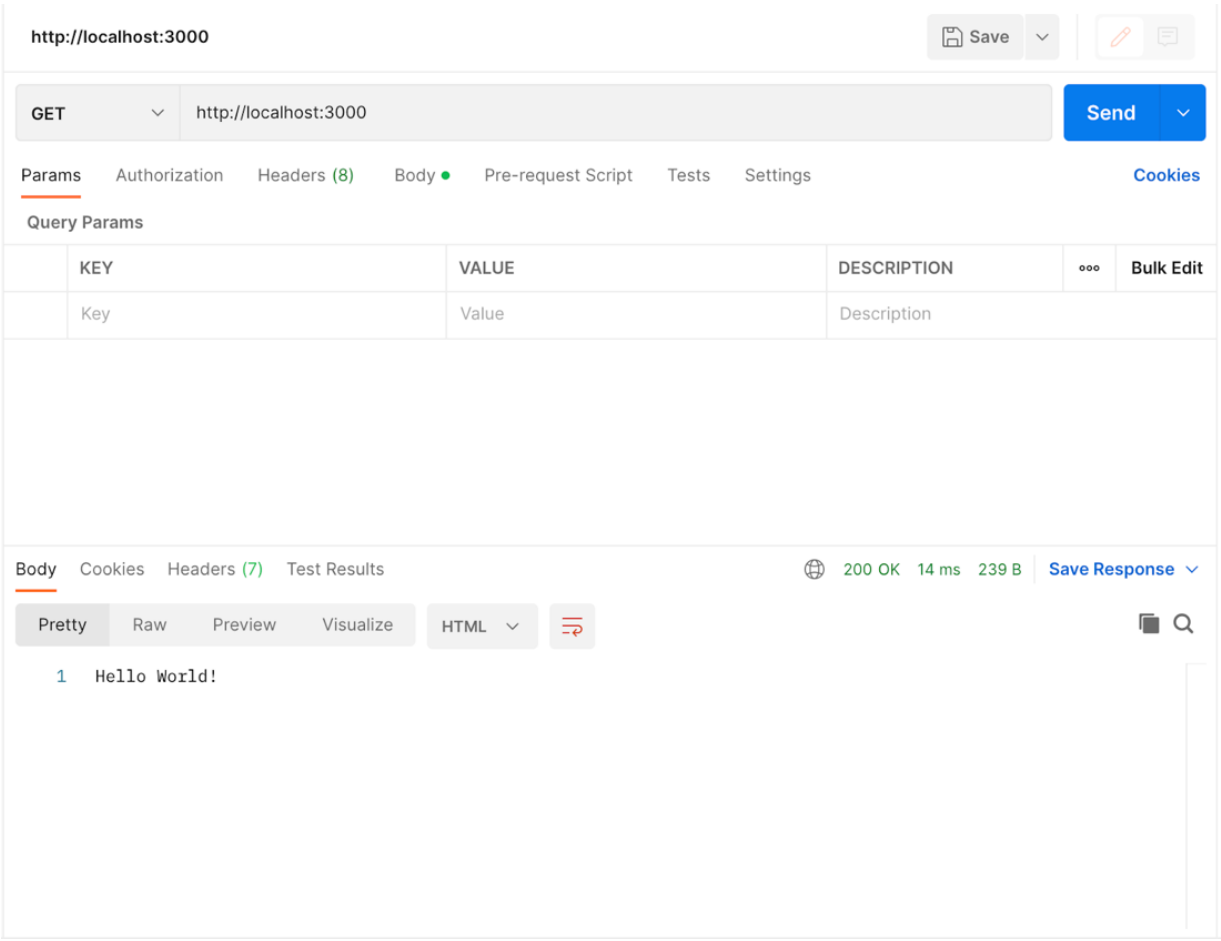


Figure 2-1. : GET request and response in Postman

Everyone will have a different preference for the tools they like to use for back-end testing and it typically doesn't matter. However, it's easier to troubleshoot when all of the devs on the team are in agreement on this particular tool. You can share your endpoint tests with other devs and that helps everyone find issues faster and more consistently. Eventually your back-end tests can serve as the documentation for the front-end. This will be something worth making a Jira ticket for so you can get it included in a sprint.

Now that you know the app is working, you can start making changes to get it ready for the team.

Updating the README

Start by updating the +README.md+ with specific instructions on how to set up and run the app. There’s already a lot of good stuff in there, so you can trim it up a bit and add a few things. A basic update can look like:

````
[source, markdown]
# Dashboard Server
## Description
Back-end to support customers built on [Nest](htt
## Installation
```bash
$ npm install
```
## Running the app
```bash
# development
$ npm run start
# watch mode
$ npm run start:dev
# production mode
$ npm run start:prod
```
````

```
## Test
```bash
# unit tests
$ npm run test
# e2e tests
$ npm run test:e2e
# test coverage
$ npm run test:cov
```
```

It doesn’t have to be a huge doc that expands on your philosophical thoughts about code and the project. Give enough guidance that any dev could clone this repo and get it running. Other docs for the project will be stored somewhere else. As the repo grows, adding more details to the README will be helpful. This is a living doc that everyone should feel comfortable updating as part of their PRs when it’s needed.

Adding a CHANGELOG

As a best practice, it’s good to include a +CHANGELOG.md+ file in your repo. This will help you keep a close record of everything in each release over time. Having this record will help you determine which version of the app contains which updates and that comes in handy when there are issues with production. Here’s what the beginnings of this file can look like:

```
[source, markdown]
# CHANGELOG
## Guide
- Major releases include breaking changes and the
- Minor releases include new features, but no bre
- Patch releases include bug fixes and performanc
### 0.0.1
- Initial release
```

You might also consider setting specific rules on commit messages. If there's a certain format you prefer, set up a tool like commitlint to enforce it early. This means you commit messages can include details like whether this is a fix or a chore. It can include ticket numbers if needed. You can also make it enforce things like the message starting with a verb to state what the change does.

That commit message might look like:

```
[souce, bash]
fix: update modal to send API call once
```

As a senior dev, you might be responsible for preparing the release PRs so it's important to make sure updating this file gets included in that process. A good way to remember it is when you update the app version in +package.json+, update this file too.

In many projects, you may need to set up ESLint, Prettier, and TypeScript config files. This app comes with a lot of configs set by default, so you should go through them and add or remove values as needed. That’s all for the initial app setup. Now you can turn your attention to building the data schema.

Monolith and Microservice Architectures

This is a crucial point in the project because you’re setting the foundation for how everything will be built for the foreseeable future. You’ll need to think about the future of the project and how it might grow and what blockers that growth might run into. Take some time to make an architecture diagram. It doesn’t have to be fancy, but it does need to represent how the back-end interacts with different pieces of the entire application.

This includes everything the app touches ranging from the front-end to cron jobs. Cron jobs are tasks that run in the background of your system on a schedule. They can be written in TypeScript as well. The back-end is like your control center. It drives the development of everything else because it interacts with a number of systems. That’s why it’s so important to take your time and make this calculated decision.

The two based architectures you'll run into are monoliths and microservices and I'll cover the differences here. You can think of this as a guide to help you choose one or the other.

A *monolith* is when all of your back-end functionality is contained in a single codebase. You'll see this a lot with enterprise apps. There's one database and one server for the application. It gets deployed as a single unit that is connected to all of the services that you work with. All of the endpoints, maybe some cron jobs, and some third-party integrations will be handled here.

With a monolith, you get the benefit of all the code being in one place. That way you can get exposure to how the entirety of the system works and where changes need to be made. Setting up the infrastructure is straight-forward because you'll only be deploying one app. Debugging is also easy because you can check the back-end holistically.

Building a monolith is a simpler approach, but it comes with drawbacks. For example, if some bug gets released to production and it causes errors at runtime that didn't happen locally, the whole app is down for users. It can also be hard to appropriately scale the app. You might only have one endpoint that gets a large amount of traffic, but with a monolith, you have to scale the whole back-end which can cost a lot.

Figure 2-2 shows an example of what a monolithic back-end can look like:

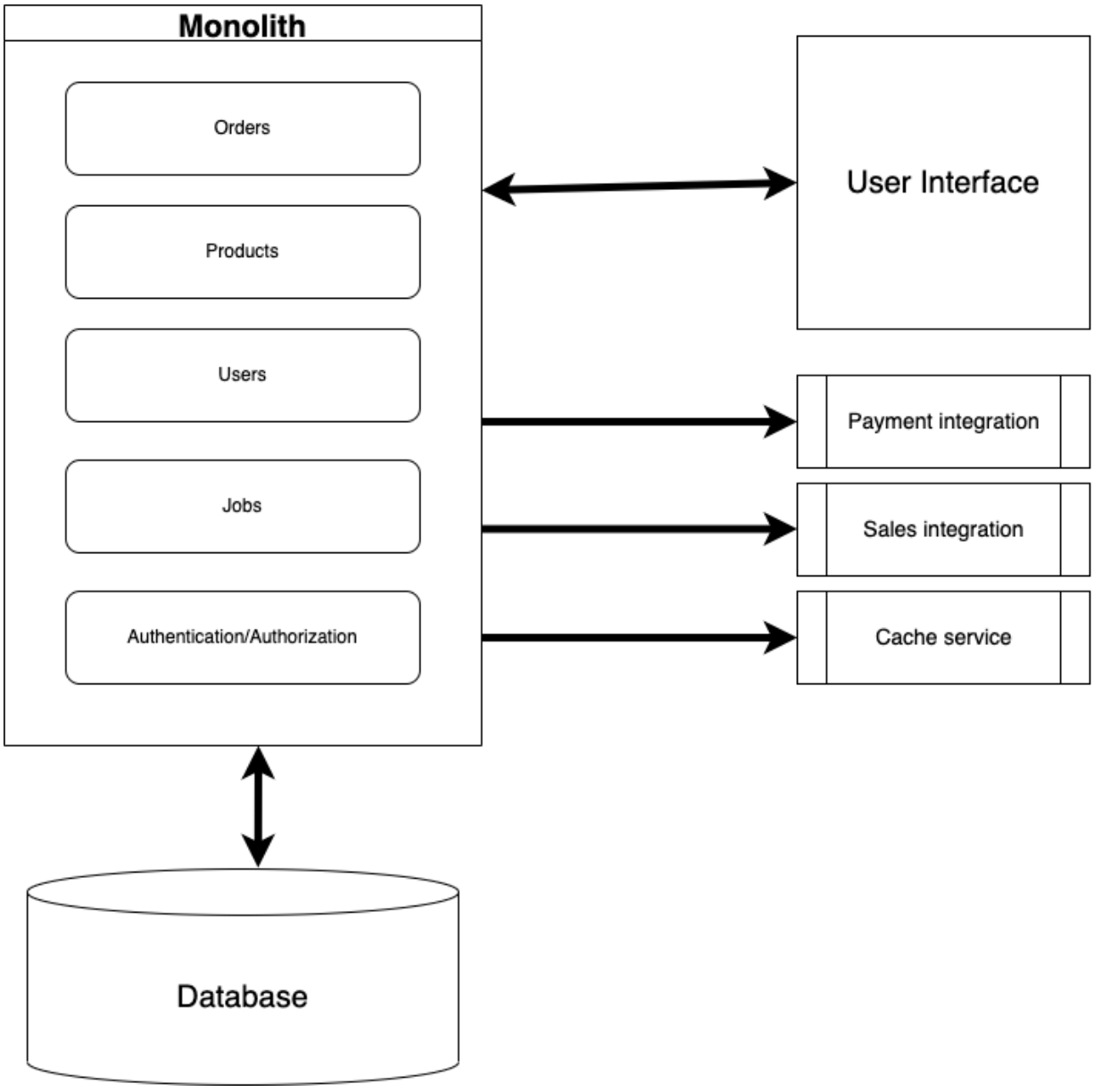


Figure 2-2. : Architecture diagram for simple monolith

On the other hand, a *microservice* is the complete opposite of a monolith. If you have a directory or set of files dedicated to one piece of business logic in your monolith, it becomes its own API in the form of a microservice. With microservices, all of the back-end business logic is split into different

chunks based on functionality. Each microservice has its own codebase, database, and server.

One of the biggest benefits of microservices is scaling. Since all of the business logic is separated into specific entities, you can ramp up resources for one endpoint while keeping the others the same. A really unique thing you can do is use different programming languages for different microservices. You might have a microservice written in Rust for concurrency handling, a microservice written in Python to serve a machine learning model, and a microservice written in TypeScript because that's the dominant language used on the team.

Microservices give you more flexibility throughout development and deployment. There are a few drawbacks though. Debugging microservices can get tricky. It may be hard to figure out the source of an issue when you have to sift through all of the microservices you have running. The issue of data integrity may also come up if you have references to the same data across multiple microservices.

Figure 2-3 shows an example of what a microservices back-end can look like:

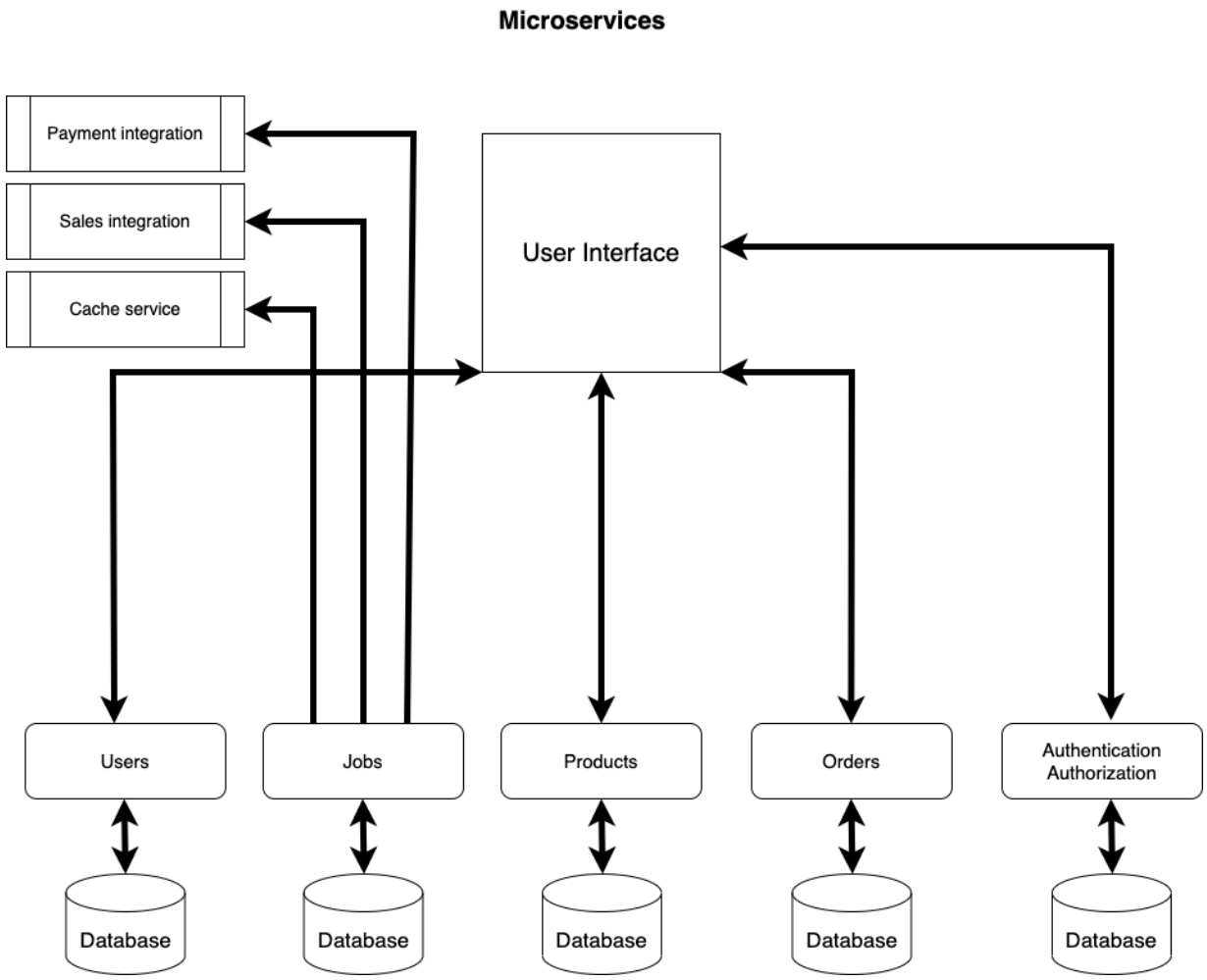


Figure 2-3. : Architecture diagram for simple microservices

Another architecture you'll hear about is serverless. These can be used to handle all of the back-end functionality, you just have to be aware of their differences because it can make a huge difference in your cloud provider bills. But if you need a few lambda functions, for example, to handle very specific behavior, this could be a good addition to your back-end design. You might see serverless and monolithic architectures used together as the product grows.

Before microservices became a thing, there was service-oriented architecture (SOA). The main difference is that with SOA, all of the separate services are sharing the same database. This can still be a good approach if you're app gets complex enough that it still might be a good idea to break out functionality into smaller chunks code-wise to keep things maintainable. I'm not going to dive into the details of SOA, but that you can consider if they should be used as part of your architecture.

The beautiful thing about architecture is that you can take the bits and pieces that work for your use case and put them together. As you work with microservices, you might hear about service meshes too. I encourage you to learn more about service meshes and how they are implemented, but I won't cover them here.

Remember, as you go through the process of designing a back-end, it will take time. This isn't something that should be rushed through and you should seek out as much feedback as possible from the other developers and teams you work with. Selecting any combination of these approaches cements the future development of the project because it's not an easy task to switch the entire architecture to something else.

Selecting an API Design Pattern: REST and GraphQL

There's something else you need to decide on at this phase of setting up the project. Are you going to go with Representational State Transfer (REST) APIs, GraphQL, or something else? This will determine how the front-end and other services are able to interface with your back-end and it will also determine how you extend your code as new roles, permissions, and other features are added over time.

REST is the standard in back-end development at the moment and GraphQL has grown to be a close second. There are other options, like Simple Object Access Protocol (SOAP) APIs, but hopefully you won't have to work with those.

The main difference between REST and GraphQL is that GraphQL uses a single endpoint to handle all requests while REST uses multiple endpoints for requests. You might hear people talk about these paradigms like they're super different, but they really aren't. It's more in the difference of the way you can build the APIs and consume them. One thing to call out is that GraphQL does come with subscriptions built in. The only way to handle that with REST is with WebSockets or a similar technology.

With REST APIs, you typically have one type of resource per endpoint. Take the `orders` endpoint in your project for example. You have `GET`,

`POST`, and `PUT` requests for the different ways you need to interact with your order data. If this were a GraphQL API, the endpoints would become your GraphQL schema like this:

```
[source, javascript]
type Order {
  id: ID
  name: String
  total: Number
  products: Product[]
  createdAt: Date
  updatedAt: Date
}
type CreateOrderInput {
  name: String
  total: Number
  products: Product[]
}
type UpdateOrderInput {
  name: String
  total: Number
  products: Product[]
}
type Query {
  order(id: ID!): Order
  orders: Order[]
}
type Mutation {
```

```
    create(input: CreateOrderInput): Order
    update(input: UpdateOrderInput): Order
}
```

With GraphQL, you can specify the operation you want with the +query+ or +mutation+ keyword instead of an HTTP method. There's no difference in the data that you'll receive compared to your HTTP requests. The GraphQL response will still be in JSON format with the same values as an HTTP response. The difference is in how you manage the code.

When you write HTTP requests in JavaScript, you specify an endpoint path, a method, and any data you need to send in the request. With GraphQL, you have more flexibility with your requests. Using queries and mutations, you can write resolver functions instead of endpoint paths. Here's an example of what resolvers might look like for your orders functionality:

```
[source, javascript]
{
  Query: {
    order: (root, { id }) => find(orders, { id: i
  },
  Order: {
    products: (order) => filter(products, { order
  },
}
So when you're ready to make a request, the query
```

```
[source, javascript]
query {
  order(id: 'fejiw-f4wt301-4tfw2g-g4t24') {
    name
    products {
      name
      price
    }
  }
}
```

This is where some of the power of GraphQL starts coming in. You can query for the exact data you want without returning values you don't need. When you make HTTP requests, you can't ask for one or two values to be returned. You get all of the data from that endpoint every time. GraphQL can save on response times for systems that handle massive amounts of data that are interrelated because you can ask for exactly what you need.

This book isn't going to do a deep dive into GraphQL, but I wanted to make sure that you know this option is available and it's widely used across the tech industry. You can think of GraphQL as the next evolution after REST. It's like when TypeScript came along to become the standard over JavaScript. It's worth taking some time to learn more about it. You can check out some resources directly at https://graphql.org/. You'll find all kinds of communities, classes, tools, and docs based around GraphQL.

Conclusion

In this chapter, you've learned about the Nest.js app you'll be setting up to handle your back-end. You've learned about a few different architectures and API paradigms and their trade-offs. Now that you've done the initial setup for the Nest.js back-end and you know why each decision was made and some of the trade-offs, it's time to turn your attention to the data schema.

Chapter 3. Building the Data Schema

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

The data layer covers a lot. That's why it's awesome we have so many good tools to work with. Regardless of the tools you choose, you do need to understand a bit about what's going on beneath the surface. When you start building your data schema, take your time and really write it out. The data schema drives everything for the app and any of its dependencies. Over time, it can really hard to update the schema without breaking everything.

Since you've already decided what kind of back-end architecture to go with, you probably already have an idea of how data will be related and

what that data looks like. That's why this chapter is going to cover:

- Initial considerations for setting up the data schema
- Setting up a database
- Using object relational mapping (ORM) tools
- Data migrations in your database

You want to be as detailed as you can and document as much of the business logic as possible. By the time you finish this chapter, you'll have a good foundation of where to start building out your data schema.

Initial Considerations

To build out your data layer, you'll go through these basic steps:

1. Make a diagram for a data schema. This should include the entities, their columns and data types, and their relationships.
2. Set up the database connection in the app. This example will use Prisma to connect to a PostgreSQL database, but there are other popular tools like Knex or Drizzle you can use. Some of the reasons you're using Postgres is because it is open-source and has a long history of reliability. You'll find it behind huge, complex apps that have been in production for decades and newer apps that have just been released.

3. Write the data schema in the app. Take your diagram and translate it into code.
4. Add seed data. This is to ensure that your database has the essential data it needs from the beginning. It's also a great tool for adding data for testing in different environments.
5. Run migrations. After the connection is established and the schema and seed data are ready, you need to run a migration to get these changes to the database.
6. Test the database with simple SQL queries. Check that the tables are creating, updating and storing data as expected. Double-check the relations between tables by looking at foreign keys.

Some apps will have a more complex scenario than this, but you'll see a process similar to this across all projects. You already know what data you're expecting based on your conversations with Product, so now it's time to make a good diagram for the dev team.

NOTE

We're not going to discuss non-relational databases in this book because we're going to work with relational databases. Relational databases enforce strict rules between data whereas non-relational databases don't. Choosing between the two depends on the type of project you're working on. If you want somewhere to store any format of data that comes in, like with events that might have constantly changing information for example, non-relational databases can give you more flexibility. Non-relational databases have specific use cases, but many apps are fine with a normal relational database.

Diagramming the Data Schema

A diagram is a great visual reference for developers and QA to understand what values to expect and why. It’s a great tool to spark discussions between the front-end and back-end, and to document relations between data in a non-code way. Again, they don’t have to be anything fancy. Something developers tend to get hung up on are little details in places that don’t matter. As a senior dev, you have to be self-aware enough to notice when you start diving too deep on a task that doesn’t need that much attention.

These tables, for example, can be just like the ones below. When possible, you want to use a tool that connects directly to the database to create the visualization, like DBeaver. That way you can see the exact relationships you’ve defined. The important part is that it’s in a format that everyone can understand.

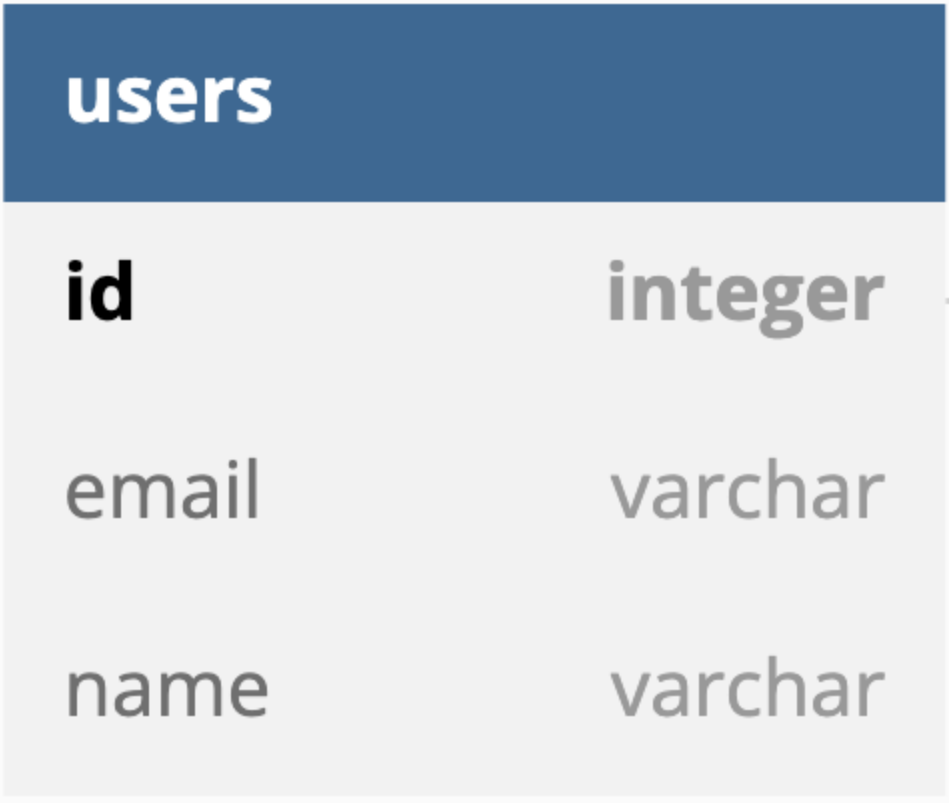


Figure 3-1. : This table shows the columns you can expect in the user table.

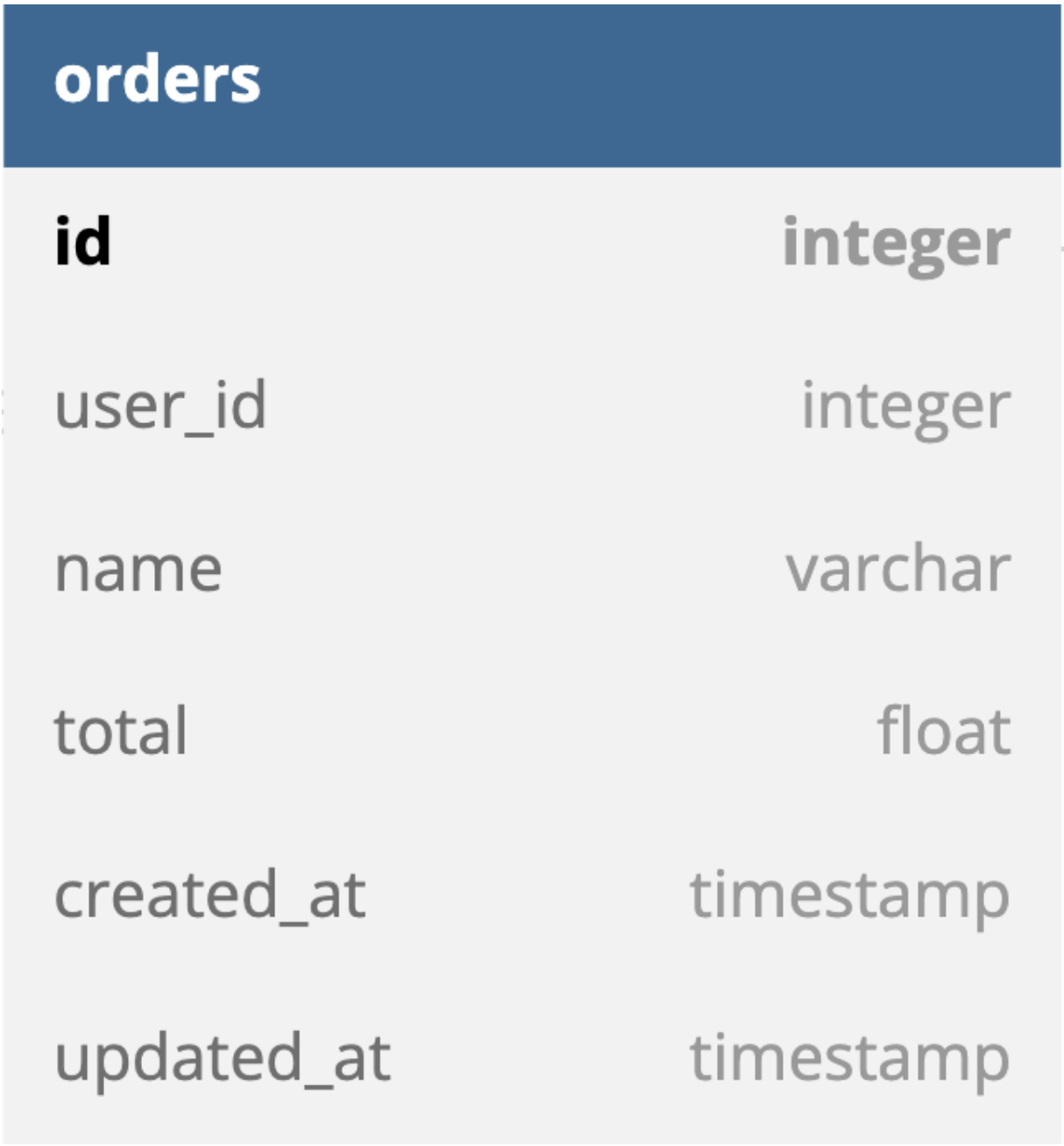


Figure 3-2. : This table has the columns for the orders information.

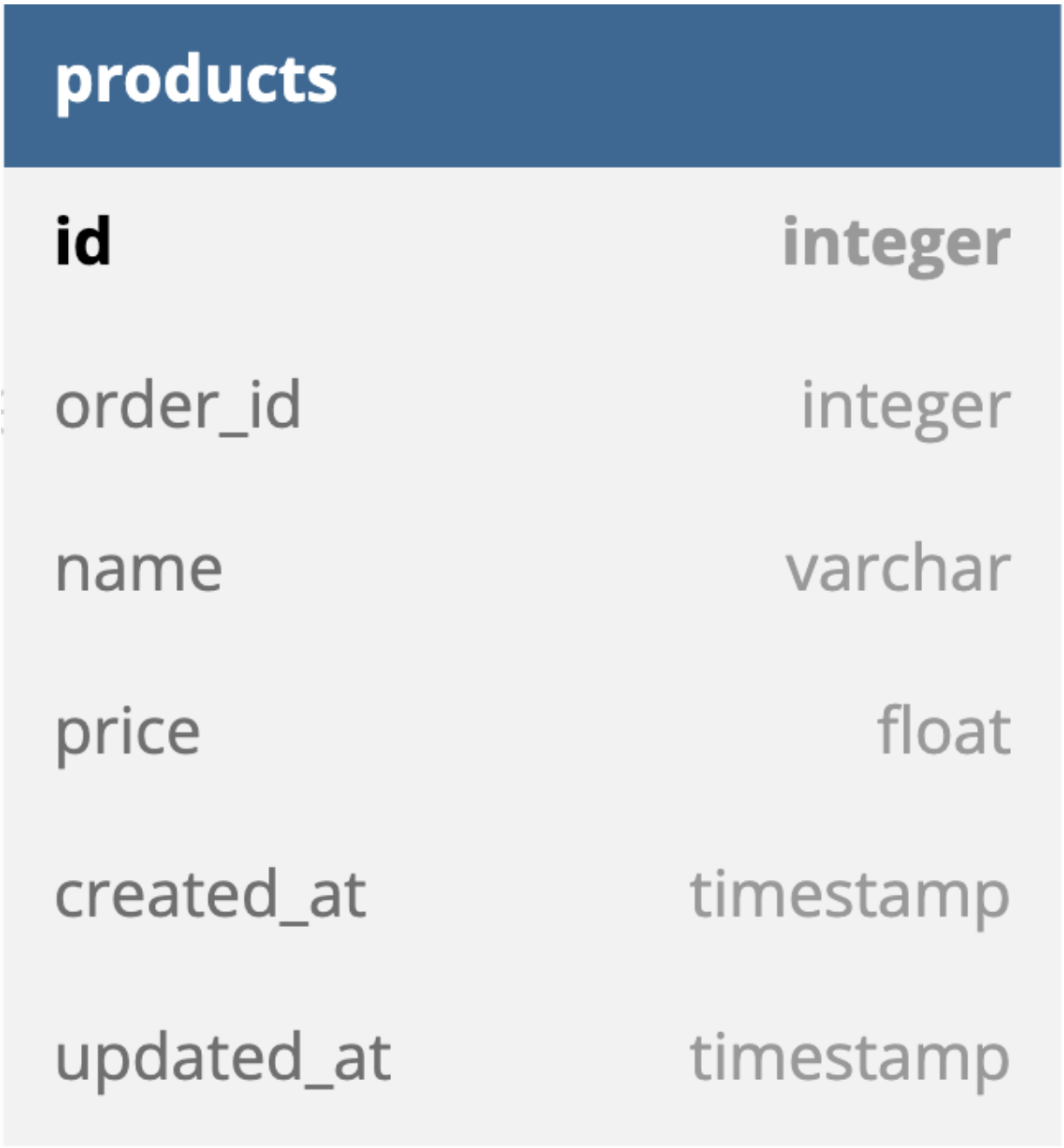


Figure 3-3. : In this table, there are the columns associated with a product.

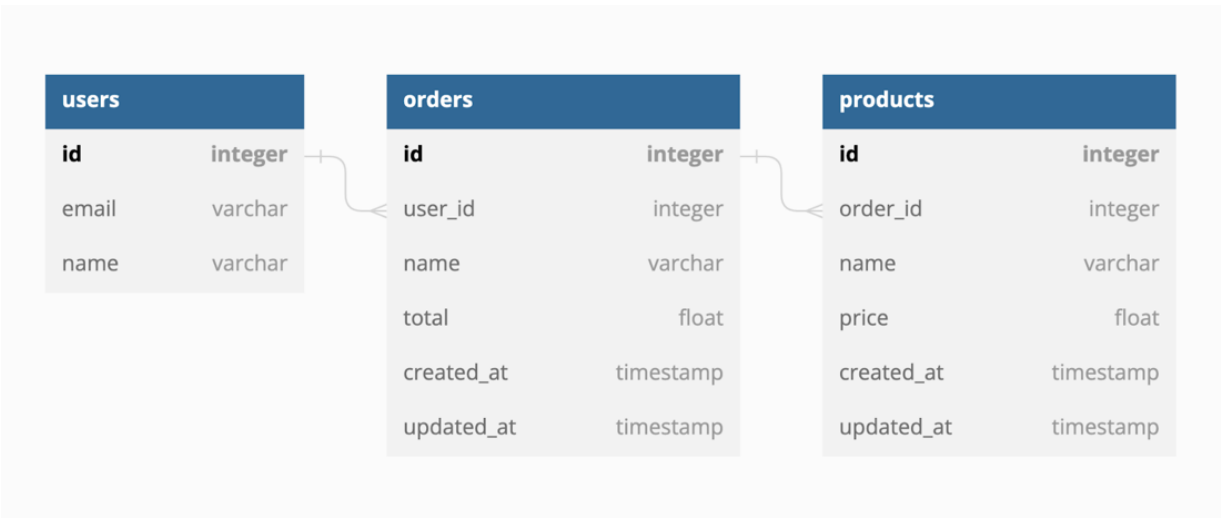
Figure 3-4: A visual of how Figures 3-1, 3-2, and 3-3 are related.

Finally, in Figure 3-4, you can see how all of the tables are related to each other.

Now that you have the data schema documented. You'll take this to the team and walk through what your thoughts are and ask for feedback. The front-end devs might have specific data format requests based on how they have to render elements. When it comes to the front-end, you typically shouldn't plan the schema around it. It's important to make sure that the queries they typically make, like searches and sorts, are thought of in the schema design though. Other devs might bring up security considerations.

By opening this up for everyone to think about, you end up with a stronger schema than if one person handled it alone. As a senior dev, you'll learn how to be more confident with sharing your ideas and getting others to open up with theirs. Don't be afraid to be wrong during these discussions because it might spark an idea for someone else.

It can be an uncomfortable place to be for a while because it feels like everyone is heavily scrutinizing your code and your technical prowess. They aren't. You have to get used to receiving constructive feedback and that's what will help you and the team build better code. The more you present your code and thoughts to the team, the quicker you'll be able to improve things for everyone.



With the data schema diagramed and agreed upon by the front-end and back-end devs, you can start work on connecting the back-end to the database to create these tables. You'll be working with a Postgres instance locally, but this can also be hosted on a server in the cloud.

Setting Up Postgres

You need to set up Postgres so you can get the connection information for your app. This will normally be handled by the team that manages the infrastructure for your production and non-production environments. You'll still need your own local instance to make sure the changes you and the team are making work as expected. This is something you'd include in a Docker container if that's how you wanted to keep the local environments consistent. As a senior dev, something else you're responsible for thinking of is how to make onboarding smooth for new devs. Getting the local environment setup is one of the biggest hurdles for anyone joining a new team, so doing this will really pay off as the team and the product grows.

You can also download Postgres for free here. Follow their documentation to get everything up and running. You should set a master password or a password on the database you create. Doing some basic database security locally can help reveal potential issues early on.

If you do choose to create a database password, make sure you remember it. You'll need it to connect your app to the database. You'll also need the database name, port number, and database username. This project will name the database, `dashboard`. Unless you chose something other than the default values, the other credentials will be the following:

- Database username: `postgres`

- Database port number: `5432`

The password depends on what you decided to do when you initially set up Postgres. If you installed via the linked download, then you can create your database in pgAdmin. It will look similar to this:

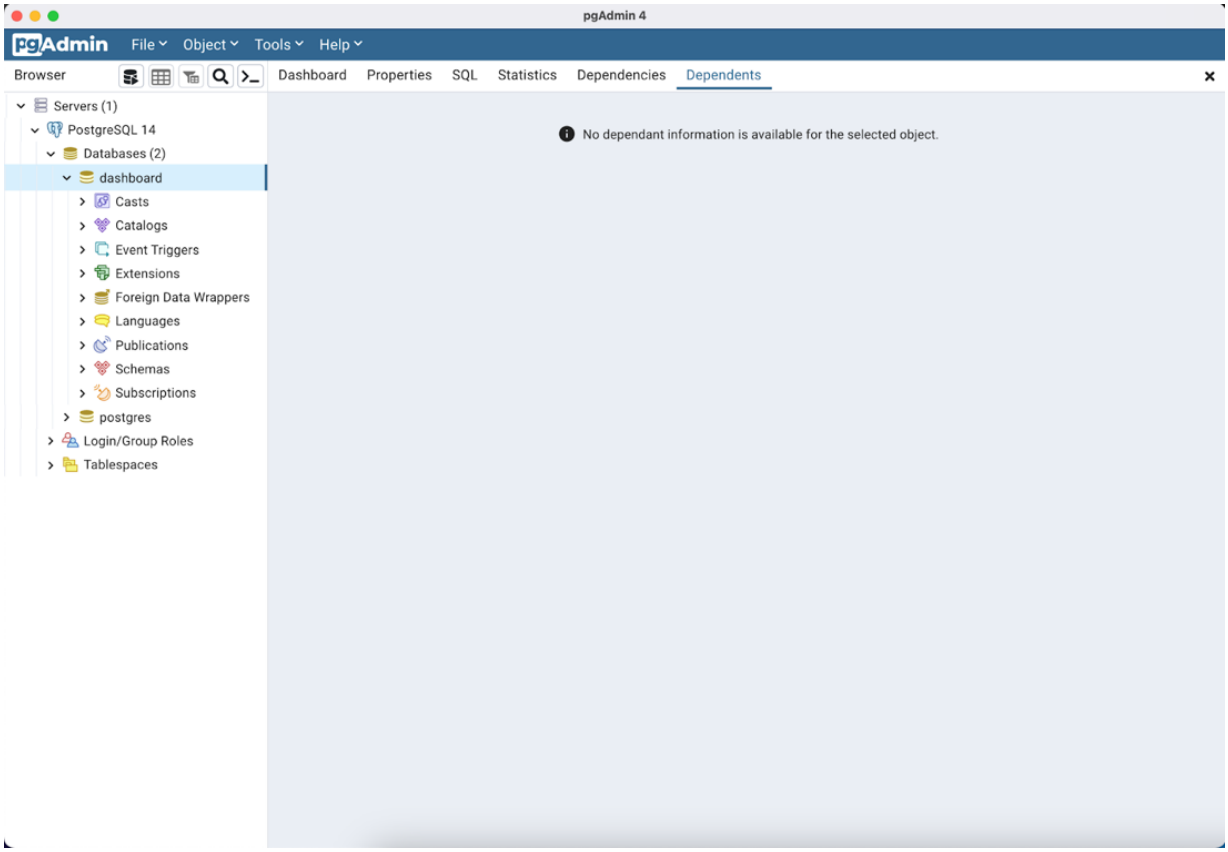


Figure 3-4. : Dashboard database in local Postgres

When you're at this point, you're ready to start creating the tables you diagramed earlier. We're going to look at a couple of different approaches to create these tables: SQL statements and using an object-relational mapping (ORM) tool.

Basic SQL Commands to Know

Let’s start with using SQL statements. As a senior full-stack developer, it’s good to know some basic SQL commands. You don’t have to get super in-depth with things like views and indexing, but knowing enough to do create-read-update-delete (CRUD) operations goes a long way. One way to get started is by writing a statement to insert a new row into a table. You can do that with the following SQL:

```
[source, sql]
INSERT INTO table_name (column1, column2, column3
```

You can change `table_name` to the table you want to insert the data into. The `column1, column2, column3` fields represent the column name you want to put the data into. Finally, `value1, value2, value3` are the values you want to add to the respective columns. For testing purposes, you can add new row entries like this:

```
[source, sql]
INSERT INTO Orders (id,name,total)
VALUES (4, 'Mark', 25.99)
RETURNING id;
```

To check that your data is stored like you expect it to be, you can query the data like this:

```
[source, sql]
SELECT * FROM Orders;
```

And to round things out, you may need to delete some data to clean up an example you've been working on. You can do that with something like this:

```
[source, sql]
DELETE FROM Orders
WHERE id = 4
RETURNING *;
```

If you can confidently use commands like these, you have enough SQL knowledge to double-check values directly in the database. Of course you can dive deeper into SQL and database management, but you don't need to in order to be a good senior full-stack dev. With these little SQL commands in your background knowledge, let's move on to the real tool you'll likely be using for database operations from your app.

Deciding What ORM to Use to Work

with the Data

Aside from choosing the framework for your back-end, the ORM tool is one of the big choices you'll make for the lifetime of your project. It's not an easy task to switch to a different ORM tool once you've started building. Nest.js comes with TypeORM, Sequelize, and Mongoose built in if you don't have a preference. There are other commonly used ORM tools like Knex and Prisma, which you'll be using.

All of these tools essentially do the same thing, but with a different flavor. The one you use will come down to the comfort level of the team and any limitations they may have. For this project, you've decided to go with Prisma because it's the tool everyone on the team has used before, the documentation is well maintained, and it's a tool used by a lot of tech teams across different projects in the industry. There's a built-in ORM, but Prisma has more support, a bigger community, and it works really well with Postgres. Reasons like these make it a strong candidate and that's why you and the team have selected it.

Now you'll need to install it as a dev dependency in your project with the following command:

```
[source, bash]
npm install prisma @prisma/client --save-dev
```

NOTE

As a senior dev, there are some finer details you need to be aware of, especially concerning dependencies. When you are adding a dependency to your project, make sure you understand what type of dependency it needs to be. There are 3 types of dependencies: dependencies (regular), development (dev) dependencies, and peer dependencies.

Dependencies are the packages your app needs to run correctly. Dev dependencies are the packages that you need to do development work, like interfacing with the database. You'll learn more about peer dependencies if you work on a project that needs to be published as its own package, but peer dependencies are the packages that your package expects to be installed in the container app.

With Prisma installed in your project, you need to set up some configs to connect the app to your Postgres database. Like any ORM, you have to initialize it in your project. With Prisma, you can run this command to do that:

```
[source, bash]
npx prisma init --datasource-provider postgresql
✔ Your Prisma schema was created at prisma/schem
  You can now open it in your favorite editor.
warn You already have a .gitignore file. Don't fo
Next steps:
1. Set the DATABASE_URL in the .env file to point
2. Run prisma db pull to turn your database schem
3. Run prisma generate to generate the Prisma Cli
```

```
  More information in our documentation:
  https://pris.ly/d/getting-started
```

NOTE

Always read the console output after you run commands. They usually give you useful advice!

This will create the `prisma` directory and a `.env` file in your project. Make sure the `.env` file is in your `.gitignore` first. Then inside the `prisma` directory, you'll find the `schema.prisma` and this is where you'll set up your database connection and the models for the app. The `.env` file has a URL to your database and this is the connection string that will contain the database credentials. Update the value for `DATABASE_URL` in the `.env` file with your local credentials. It might look something like this:

```
  [source, bash]
  DATABASE_URL="postgresql://username:password@loca
```

Any values in your `.env` should be handled in your CI/CD pipeline. Work with the DevOps team to get this in place depending on the infrastructure setup. Now you can move over to the `schema.prisma` file and start writing your model. You've already done the hard part of thinking out how

everything relates, so now you can confidently start coding. You'll notice this is where you can see one of the biggest differences between the ORM tools. Prisma has its own flavor that you'll have to get used to and it's ok to refer to the docs often.

In your `schema.prisma`, you can start adding pieces of your model. Add the following code to the end of the file:

```
[source, javascript]
// schema.prisma
...
model User {
  id       Int       @id @default(autoincrement()
  email    String    @unique
  name     String
  products Product[]
}
model Product {
  id       Int    @id @default(autoincrement())
  name     String
  imageUrl String
  price    Float
  quantity Int
  User     User?  @relation(fields: [userId], ref
  userId   Int?
}
```

Building models with Prisma is similar to writing objects or type definitions if you're familiar with TypeScript. These models use special Prisma syntax for the types, but they match closely to the common types you work with. The most important thing to note is how relationships are defined between tables. On the Product table, you see we have an associated user id. On the User table, we have an array of products.

This is how Prisma defines relationships between tables and I highly encourage you to look through their documentation to learn about building more complex relationships. You might also want to look into the Prisma VS Code plugin to help make development smoother. For this app though, these few models will get you moving.

This completes the data schema for your app so far. Now it's time to get this schema onto the database with a migration.

Writing Migrations

Migrations are the SQL queries statements by the ORM based on your schema definition. When you connect to the database to run a migration, you're essentially executing SQL statements. That's what makes ORM tools so useful. Instead of having to manually write the SQL for multiple tables, like you did earlier, you can write the query in JavaScript and the ORM will

translate it to SQL. That’s why JavaScript developers use these tools so they don’t have to learn all of the details of SQL and database quirks.

To run a migration with Prisma, you’ll open your console and navigate to the root of the project and run this command:

```
[source, bash]
npx prisma migrate dev --name initialize_dashboar
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": PostgreSQL database "dashboard",
Applying migration `20230318132006_initialize_das
The following migration(s) have been created and
migrations/
  └─ 20230318132006_initialize_dashboard_db/
    └─ migration.sql
Your database is now in sync with your schema.
```

Anytime you update your schema you’ll need to make a new migration to update the database. Make sure to give it a descriptive name so you can quickly understand what happened in the database history. This is super helpful if you run into data issues on the back-end because it’s easier to see what changes have been made and when. Every migration will automatically generate the timestamp at the beginning of the folder name in

Prisma. The timestamp is important so the database knows what order to run the migrations.

Other tools will handle migrations a little differently. Knex, for example, lets you write migrations in pure SQL if you want and it generates migration files instead of folders. If you look in your project's `prisma` directory, you should see a `migrations` folder with an subfolder. This is where the generated SQL for your migration is. That's the beauty of an ORM. You can write in syntax you're familiar with and it will generate the SQL for you. Take a look at the `migration.sql` file in the migration folder. You'll see something like this:

```
[source, sql]
-- CreateTable
CREATE TABLE "User" (
    "id" SERIAL NOT NULL,
    "email" TEXT NOT NULL,
    "name" TEXT NOT NULL,
    CONSTRAINT "User_pkey" PRIMARY KEY ("id")
);
…
```

This is just the initial set up migration for the database and you'll be adding more tables and columns and changing names as the app grows. When you run into issues with a migration, you can roll it back and you'll learn how to do that in the debugging section of this chapter. For now, you have what

you need to add some seed data to the database so you can start working on the rest of the app.

Seeding the Database

Since this is your local database, you'll eventually want some test data to play with. You can add that as part of the dev env setup or have it as part of the setup for the real database. To start, create a new `seed.ts` file in the `prisma` directory. This is where you'll use the Prisma Client to create some records. In your `seed.ts` file, add the following code:

```
[source, javascript]
// seed.ts
const { PrismaClient } = require('@prisma/client
const db = new PrismaClient();
async function main() {
  const orderData = [
    {
      name: 'inv_924',
      quantity: 2,
      total: 92.42,
      createdAt: new Date('2023/02/21'),
      updatedAt: new Date('2023/02/24'),
    },
    {
      name: 'inv_925',
```

```
        quantity: 7,
        total: 24.56,
        createdAt: new Date('2023/02/21'),
        updatedAt: new Date('2023/02/24'),
      },
  ...
```

You can find the complete `seed.ts` file here:

https://github.com/flippedcoder/dashboard-server/blob/main/prisma/seed.ts

This code allows you to connect to the database and insert these new rows into their respective tables. Then it disconnects from the database. In order to run this with Prisma, you need to add the `seed` config to your `package.json` like this:

```
[source, json]
...
"collectCoverageFrom": [
      "**/*.(t|j)s"
    ],
    "coverageDirectory": "../coverage",
    "testEnvironment": "node"
  },
  "prisma": {
    "seed": "ts-node prisma/seed.ts"
```

```
  }
}
```

This tells Prisma where to look for your seed file and how to execute it. With the configs and some data ready, you can run this command to actually insert the data into your Postrgres instance:

```
[source, bash]
npx prisma db seed
Environment variables loaded from .env
Running seed command `ts-node prisma/seed.ts` ..
Y The seed command has been executed.
```

Now if you look in your Postgres tables, you should see the values you wrote in `seed.ts`. This should be everything you need for your data concerns right now. Just to double-check, here's a quick checklist you can go through when you think you're finished setting everything up:

- Does the schema match the designs and functionality explained in the behavioral doc?
- Have you had at least one other dev look at it?
- Have you checked to make sure the schema works for all of the apps consuming data from this database?

There are more advanced things to consider as well that are out of scope for this book. Here's a list of some of those things and links to more resources:

- Does your schema give room for the app to grow? https://www.oreilly.com/library/view/software-architecture-the/9781492086888/
- Is there a way to audit actions and the users who triggered them? https://www.auditboard.com/blog/what-is-an-audit-trail/
- Have you considered different user role levels for table operation access? https://www.imperva.com/learn/data-security/role-based-access-control-rbac/

These advanced topics could have their own chapters or even books. There are a wide range of questions you can ask here, but to keep things moving forward at this stage, don't get too deep into the details. If you can answer these questions, explain the schema decisions to another dev, and you can demo it to the Product team, you're good to go for now.

Take notes and make Jira tickets for any optimizations you see along the way. That way you can come back and actually work on the details because it's documented in a way that Product considers. You've decided that the schema is in a good place and the tickets regarding different endpoints can be unblocked. This is when you'll start working on the API that different apps will interact with to get data from the database you just set up.

Conclusion

In this chapter, you went through the process of drawing diagrams for your data, setting up a Postgres instance, and handling some initial setup with your ORM. You should feel pretty good about expanding your data schema from here because you've already asked a lot of questions about the future of the app.

Chapter 4. REST APIs

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

With the data schema defined, it's time to get into the details of exposing that data. You'll build an application program interface (API) with multiple endpoints that perform various operations, like getting all of the orders or submitting new ones. This is one area of the backend that will get very into the details.

In order to create an API that can be maintained by different developers as the team changes, you need standard conventions for the team. Standard conventions are usually outlined in a document that defines the way the

team agrees to approach code implementation from everything to naming conventions to the error codes and messages used in responses.

This set of conventions will make it easier to check for deviations in pull request (PR) reviews for any code changes. Your PRs are changes you submit for other developers to review before they get merged with the central code and deployed. Since there might be multiple PRs for a piece of functionality, this rule enforcement is important to maintain code consistency across code bases.

This chapter will go over how to address this convention while actually building an API and cover these areas:

- Working through data formatting with the frontend and other consuming services
- Writing an example of code conventions
- Writing the code for the interface, service, and controller for the API
- How to track errors with logs and custom error handlers
- Ensuring validation is in place

These are some of the concerns that will come up as you build out your API and the different endpoints. You'll run into a lot of different approaches to API development and architecture decisions and all of them are valid. Like with everything else, it depends on the needs of your project and team preferences.

For this API, we’re going to follow these conventions:

- It will send and receive data in JSON format
- Endpoint logic shouldn’t reference other endpoints to avoid race conditions
- Use endpoint naming to reflect relationships of data and functionality (ex. /orders/{orderId}/products)
- Return standard error codes and custom messages
- Version the endpoints to gracefully handle deprecation
- Handle pagination, filtering, and sorting on the backend
- All endpoints receiving data should have validation
- Endpoint documentation should be updated with all changes

Making sure the frontend and backend agree

There is always a partnership between the frontend displaying data and the backend processing it. When the frontend has to make multiple calls to fetch data, it can cause the view to render slower for a user and it might prevent them from doing much on the page. When the backend sends more data in responses, it can lead to unnecessary information being gathered and sent which makes the response take longer. This is a trade-off you have to balance.

Typically any type of pagination, filtering, sorting, and calculations should happen on the backend. This is because you can handle the data more efficiently on the backend compared to loading all of the data on the frontend and making them do these operations. It’s very rare you’ll ever want to load all of the data for an app in the frontend.

If these requests come from the frontend team, it’s common for the backend to honor it. This might also be a case for introducing a microservice into the architecture. If there’s a specific endpoint that is called way more than the others, it’s worth researching if it makes sense to separate it out and what approach you would take. That can possibly help both the frontend and backend performance.

One area that you might have to push back on is how endpoints send data to the frontend. They could request that data gets sent back on the same endpoint to help with performance and reduce the number of calls made. If it crosses data boundaries, then you need to double-check if the data should be combined. Data boundaries are how we keep a separation of concerns between different functionality. Here’s an example of keeping a boundary between any products and orders calls.

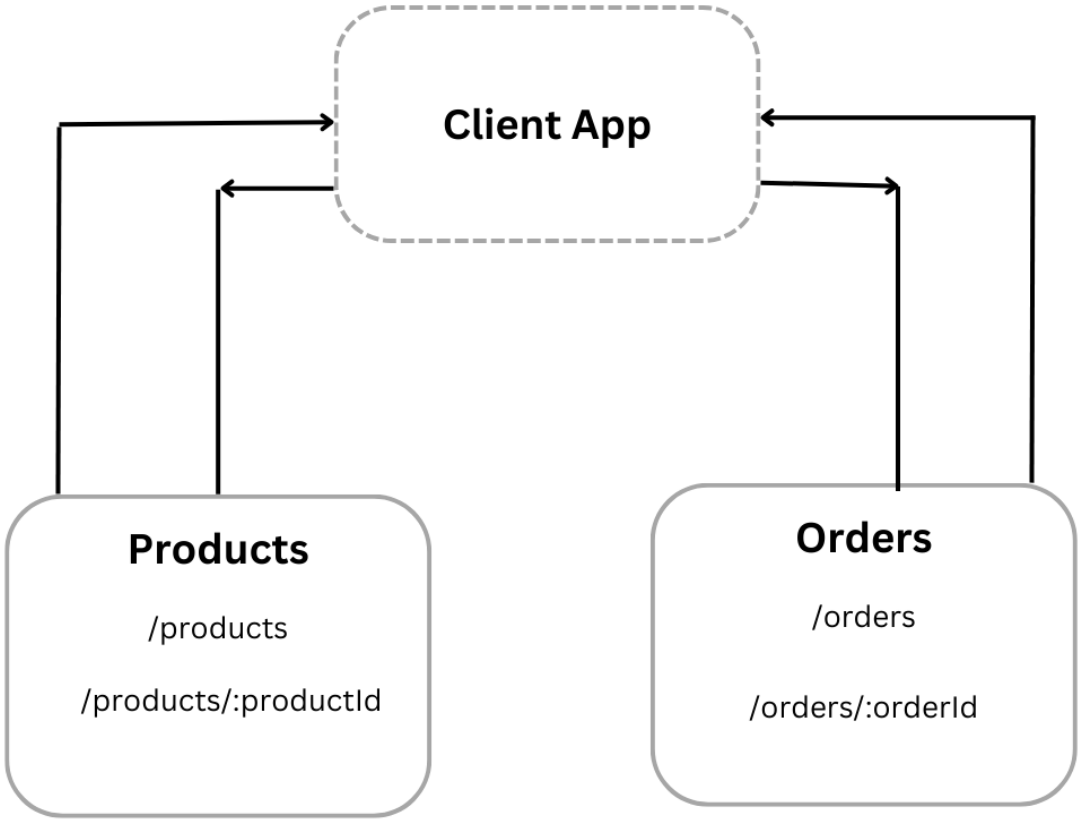


Figure 4-1. Example of data boundaries and how they don't cross

Remember, the frontend can filter the data out in the view, but anyone can check the developer tools in the browser to check the network response. It's always the responsibility of the backend to enforce security when it comes to data handling. The frontend can make some security decisions, but users can bypass the UI and use the endpoints directly. We'll address some

security concerns in chapter 8, but this is one of the reasons you want to enforce data boundaries.

Now that the frontend and backend understand the expectations of each other, let's work on the conventions doc for the backend.

Creating a document for conventions

You can tighten up your code conventions even more with a doc that you share with the team and that you use to help new devs onboard with the team's code style. These will evolve over time as you encounter new scenarios. It serves as a source of truth for the team when a new endpoint or even a new API needs to be created. That way everyone knows how to build the code so that consistency is maintained everywhere.

There are also tools available like Husky, ESLint and Prettier that can help you automatically enforce the conventions with every PR. Sometimes nitpicks like spacing, tabs, quotation marks, and tests can get tedious to manually check for. Using these types of tools makes it so devs have to meet the conventions before a commit can even be made. Now every PR will have these minute checks which makes reviews faster and code more consistent.

You can find an example of of some conventions here: https://github.com/Wolox/tech-guides/blob/master/backend/backend-

standard-and-best-practices.md Docs like this can be starting points for your in-hourse conventions that you extend to match team preferences. For example, your conventions doc might contain an additional section to the one linked above.

Example 4-1. Excerpt from SampleCorp's API team's code conventions doc

```
—-
For pagination, responses should return this stru
{
 "page": [
    {
      "id": 4,
      "first_name": "My first name",
      "last_name": "My last name",
      "email": "myemail@server.com"
    },
    {
      "id": 5,
      "first_name": "My first name",
      "last_name": "My last name",
      "email": "myemail@server.com"
    },
    {
      "id": 6,
      "first_name": "My first name",
      "last_name": "My last name",
```

```
        "email": "myemail@server.com"
      }
  ],
  "count": 3,
  "limit": 3,
  "offset": 0,
  "total_pages": 4,
  "total_count": 12,
  "previous_page": 1,
  "current_page": 2,
  "next_page": 3,
}
For error handling, responses should return this
{
  "errors": [
    { "statusCode": "111", "message": "age must b
    { "statusCode": "112", "message": "email is r
  ]
}
—-
```

This is another way you can get feedback from others and make the convention doc the best it can be for everyone.

NOTE

There are some things you'll set in your conventions that will be annoying to stick to sometimes, especially in a time crunch. That's why it's important to implement a few features and see how things go in practice before you really enforce them through linting and other scripts.

Once they're in a good place though, don't deviate from them unless there's a significant change to the direction of the project. The consistency throughout your code bases is what will make this project easier to maintain over the long-term.

Making the API and first endpoint

As discussed in chapter 2, Nest.js is the framework you'll build with. You won't go through writing all of the code tutorial-style here. I'll address underlying reasons for why functionality is implemented a certain way. This is the type of thinking that spreads to any framework. I will leave it up to you to look through the Nest.js docs to understand the syntax of the code.

There are so many things to consider when you start coding on the backend. The secret is to just pick an area and focus on it first. You'll come back through and address security, performance, and testing concerns. For now though, you need to get some endpoints working so the frontend can start work connecting the UI to the API. You'll start by writing the basic CRUD operations you know the app will need. In this project, you'll need CRUD endpoints for:

- Managing products

- Managing orders
- Administrative functions

The first two sets of endpoints are based on what you already know about the app. The last set of endpoints will come from other discussions with Product. There will be actions the Support team will need access to that no user should ever be able to touch. You'll learn how to handle these different user permission levels and access control in chapter 8. Remember, these endpoints will likely change. The main thing is that you have to start building somewhere.

You can delete some of the boilerplate files, specifically `app.controller.spec.ts`, `app.controller.ts`, and `app.service.ts`. Also go ahead and update `app.module.ts` to remove the references to those files. This is to keep things as clean as possible as you start to make changes and add new code.

Working on the orders endpoints

You can start by working on the functionality around orders. In the `src` directory, make a new subfolder called `orders` and add the files to handle the types for this endpoint, the tests, the service, and the controller.

The `orders.controller.ts` file is where you define all of the endpoints for this specific feature. So anytime you need to fetch orders or make changes to them, the frontend will only reference the endpoints here.

This is a great place to do your initial validation on data received in requests. Here’s an example of an endpoint:

```
// orders.controller.ts
…
@Get()
  public async orders(): Promise<Array<Order>> {
    try {
      const orders = await this.ordersService.orc
      return orders;
    } catch (err) {
      if (err) {
        throw new HttpException('Not found', Http
      }
      throw new HttpException('Generic', HttpStat
    }
  }
```

Here you can find some custom error handling and it sends a message and status code like you defined in the conventions. The controller shouldn’t contain any business logic because that will be handled in your service. Controllers are just there to handle requests and responses. This makes the code more testable and it keeps the code separated based on what it should do.

Controllers will also do some of that validation I've mentioned. Let's take a look at an update endpoint:

```
// orders.controller.ts
…
@Patch(':id')
public async update(
  @Param('id', ParseIntPipe) id: number,
  @Body() order: UpdateOrderDto,
): Promise<Order> {
  try {
    return await this.ordersService.updateOrder(
      where: { id },
      data: order,
    });
  } catch (err) {
    if (err) {
      throw new HttpException('Not found', HttpSt
    }
    throw new HttpException('Generic', HttpStatus
  }
}
```

Note how the `try-catch` statement is used for both of the endpoints. This is a clean way of making sure your errors are handled. The code in the `try` block is always run first. If any errors happen in this block, the

`catch` block will be triggered. Then you can focus on how to handle the errors that are caught. This is something that can get overlooked when devs are in a hurry and it's a prime candidate to include in your code conventions.

The validation here is happening through the `UpdateOrderDto`. Here's what it looks like in `orders.interface.ts`.

```
// orders.interface.ts
export class UpdateOrderDto {
  @IsNotEmpty()
  name: string;

  @IsNumber()
  total: number;
}
```

The validation is handled with the class-validator package, so if the name is empty or the total isn't a number, an error will be thrown to the frontend telling it the body data was in the wrong format. Sending proper validation messages to the frontend will help the devs know what to do and give users helpful information.

Working on the orders service

The last file is `orders.service.ts`. This is where the business logic for the orders functionality is handled. Any calculations, sorting, filtering, or other data manipulation is likely happening in this file. Here's an example of a method to update an order:

```
// orders.service.ts
…
public async updateOrder(params: {
    where: Prisma.OrderWhereUniqueInput;
    data: Prisma.OrderUpdateInput;
  }): Promise<Order> {
    const { data, where } = params;
    this.logger.log(`Updated existing order ${dat

    try {
      const updatedOrder = await this.prisma.orde
        data: {
          ...data,
          updatedAt: new Date(),
        },
        where,
      });

      this.logger.log(`Updated for existing order
```

```
        return updatedOrder;
      } catch (err) {
        this.logger.log(`Updated for existing order

        throw new HttpException(err.message, HttpSt
      }
    }
```

Now you're adding even more backend best practices and following the conventions because you have error handling and logging happening here. There will also be errors that come from the service level, which is why you have the `try-catch` statement to bubble those errors back up to the controller.

You should also add logging like this in your controllers. One thing you'll find is that logs are invaluable when you're trying to debug the backend. Make your logs as descriptive as you need to in order to track values across your database and other endpoints or third-party services.

Regardless of the framework you decide to use on the backend, you've seen the core things you need to implement: validation on inputs, logging for the crucial parts of the flow, and error handling for issues that may arise. As long as you remember these things and you keep the code conventions in mind, your team is on the way to a strongly built codebase.

It’s time to look at some of the other parts of the backend that will ensure the endpoints and services work as expected.

Establishing a working database connection

Many projects have a folder called `utils` or `helpers` or something like this. You’ll need to create one of those to hold the service you’re going to use to instantiate `PrismaClient` and connect to the database. In the `src` folder, make a new folder called `utils`. In this folder, make a file called `prisma.service.ts` and put the following code from the Nest.js documentation in there: https://docs.nestjs.com/recipes/prisma#use-prisma-client-in-your-nest js-services. You don’t have to worry about writing everything from scratch most of the time if you spend a few minutes reading and looking through docs. That’s a thing you’ll find senior devs doing all the time.

Don’t be afraid to add more things to this `utils` folder! When you see some small function that is repeated in numerous parts of the app, like data formatters, move them here so they are easy for other devs to find and use.

If you haven’t stopped to make a Git commit, this is a good time to do so. Now you have the backend in a state where other backend devs can come in and add more functionality or configurations. One of the hardest tasks is to set something up that others can improve. That’s what you’re doing right now.

As you build on this application through out the book, you'll start to add calls to third-party services, handle data from different sources, and work on security concerns. All of these will involve endpoints and other service methods that you'll add on as you move through the tasks on your sprint.

It's important to get some practice in, so try to add error handling, logging, and validation to the remaining endpoints in the orders controller. Of course, you can always check out the GitHub repo too.

Conclusion

We covered a lot in this chapter and we will dive even deeper! The main takeaways from this chapter are how to make an agreement with the frontend, setting up strict conventions for your API as soon as possible, making some initial endpoints to get the frontend moving, error handling, validation, and logging. Now that you have a few endpoints up, you can start building on top of them.

Chapter 5. Third-party services

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

There will come a point in developing a product that you'll need to use a third-party service. Third-party services offer complex or proprietary functionality that exists outside of your code base and outside of your company. It's usually managed by another company and you pay a fee to use it and have access to support when things go wrong.

Third-party services come up when you need functionality that would take over a year to implement, would be difficult to maintain, and would likely

need its own team, like payment systems, authentication/authorization, and monitoring and logging for your app.

In this chapter, you'll go over:

- How to choose third-party services
- Implementing a third-party service in your system
- How to manage errors, outages, and upgrades

Working with third-party services is going to come up at some point, so it's best to go in with an idea of what you're getting into. Usually the third-party service provides an API or SDK as an npm package you normally install in your app, but you'll need to provide some type of credentials to access the full functionality. There are also some SDKs that exist without a service just to give you some specific functionality.

Keep in mind that there's a chance something on the service's side could change and break without letting you know. It's all interface code that you'll be working with, so it won't be any different in that regard.

We'll be adding Stripe as the third-party service to handle the payment part of the app. First, you'll go over some of the things you should consider for third-party services. Then you'll implement the controller, service, and other code to get Stripe working in the demo project.

Choosing a third-party service

There are numerous services to handle payments, authentication, data processing, tax handling, logging, working with other company's data, and any other functionality you can think of. That can make the task of choosing a service seem daunting because we have to decide what to build and balance that with costs. In reality, it's pretty similar to the process you use to select a package you'll use in your code.

The most important difference between selecting a third-party service and a package is that the services are going to cost money. There's nothing wrong with either of these things because it's going to happen. Getting locked into a great product with a good pricing structure is something to hope for.

Here are some of the things you might look into as you research services you need:

- Does it have good documentation that is regularly updated?
- How do they handle new version releases?
- Is there responsive support available?
- Is the pricing structure clearly defined, even if you have to get on a call with a sales rep?
- Are there other options for the functionality you need?
- Is the company mature and stable?
- How easy would it be to add to your existing architecture?

- How does this solution compare to other popular solutions for this service?
- Is there a good sandbox environment for testing?
- How much effort would it take to switch to a different tool from this one?

These are things to consider and there will be some interesting trade-offs. You might even get into more industry-specific questions around legal compliance and regulations. Once you've done a thorough business analysis of a service, take it for a spin in your code base. See how long it takes you to add the smallest working integration to your code. That'll be a good early test run to see what it would be like working with it.

When you're doing this testing, don't forget to share your findings with the team as well as management. Do some quick demos to show how things are going in the code. This is one way you as a senior dev can help others on the team level up. It'll also help you understand your service at a deeper level because you'll be answering questions they have.

While you're checking out how the integration works, also look into the reputation of the product and company. It's important to know the experiences other developers have had with any service you're considering paying for. If they seem awesome and have great support, see if anyone else had that experience after they signed a contract. It may seem cynical, but

it’s good to check out these things before you are deeply integrated with a product.

Another thing you need to consider is the country of origin of the service. If you develop software for any government, there will likely be countries that you have to avoid selecting software from. This is something that you might not initially consider depending on what industry you’re in, but geopolitics affects the tools we use to develop software. On the other side, if your company wants to show its support for a country, you can also evaluate services for good geopolitical reasons.

As you continue to look through third-party services, also consider looking at the GitHub repo for the service if there is one. You’ll be able to see other developers’ experiences with it and how the third-party service company handled any bugs or questions. You’ll probably find the most common issues everyone runs into before you’re too deep into the implementation.

One more thing to check for is how third-party services interact with each other. You might need to use multiple services through another service like Zapier or your cloud provider’s services and it can make things complicated. See if any of the services complement the other tools and services you’re using.

List of potential services

It can be hard to know what to even look at when you figure out you need a third-party service. These are some of the services you may run into and some options to start with:

- Payment handlers: Stripe, Square, Clover, PayPal, Paddle
- Logging/monitoring: DataDog, New Relic, Splunk, Sentry
- Third-party apps: Meta, Instagram, YouTube, GSuite
- Ecommerce: Shopify, Amazon, Etsy, BigCommerce
- Authentication: Auth0, FusionAuth, Amazon Cognito, SuperTokens, Clerk
- Email services: SendGrid, Amazon SES, Mailgun, Postmark, Brevo

You can also run into some very niche services out there depending on what your product needs. These can be challenging to integrate, so make sure you are clearly communicating what you're finding and ask others to join in the search too.

Since the app you're building will require the ability for users to make payments, you'll need to implement a third-party service to handle this. A service that has Payment Card Industry Data Security Standard (PCI DSS) compliance built in and has loads of security in place to keep users' financial and personal information secure. For this part of the project, you'll get to work with Stripe.

Integrating Stripe

The reason you'll use Stripe in this project is because they have great documentation (which you'll refer to often), there's a large community of developers that use it, and there's a pretty good testing environment you can use. I do encourage you to take a look at some of the other options and compare the trade-offs you see.

NOTE

I won't go through setting up a Stripe account because it gets pretty in depth and you will need to add your personal information. If you don't feel comfortable doing that, it's ok. This information will be provided by your company anyways. I'll go over the programmatic implementation so you can follow along with the code to see what you would do after the account is set up.

The first thing you need to do is decide how you want to handle third-party services in your app. When you're thinking about this, keep in mind that you will likely have other integrations as the product grows and needs more capabilities. The approach you'll take in this project is adding a new folder called `integrations` in `src`. Inside the `integrations` folder, you'll add a subfolder called `stripe` with a few files in it. This will make your folder tree look like this:

```
|__ prisma
```

```
  |__ src
  |____ integrations
  |______ stripe
  |________ stripe.controller.spec.ts
  |________ stripe.controller.ts
  |________ stripe.interface.ts
  |________ stripe.module.ts
  |________ stripe.service.spec.ts
  |________ stripe.service.ts
  |____ orders
  |____ app.module.ts
  |____ main.ts
  |__ test
```

This is also a great time to update your architecture diagram to show how this new service integrates with your overall app. Docs are never really finished, so remember to update relevant docs as you add new functionality. The way third-party services will work with this app includes some event handling with your cloud service with tools like AWS CloudWatch and EventBridge as well as directly interfacing with your own APIs.

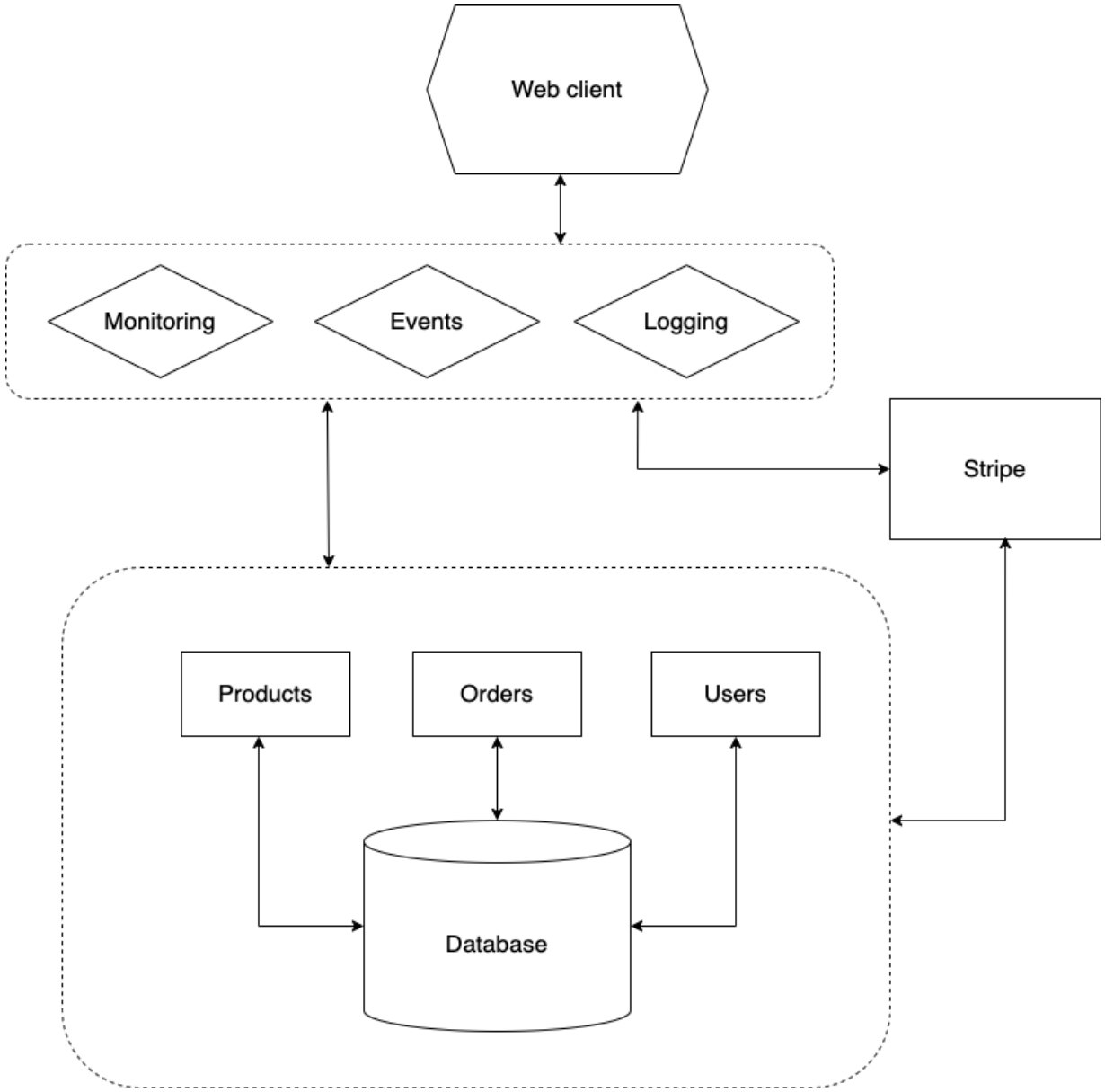


Figure 5-1. Updated architecture diagram

You'll follow the same programming pattern here as you did with the orders functionality. The endpoints your front-end will call are defined in the controller. The interactions directly with the Stripe API will be in the service. Keeping this separation of concerns is what helps makes your code

modular and easier to test and you'll want to keep that up, even as you add other third-party code.

You can start by updating the new `stripe.module.ts` file to import and use the new service and controller because this is a quick win.

NOTE

Starting with a quick win is a way to build up momentum when you're facing something big. It can make the problem feel more approachable. As you write out functionality see if you have an order that you like to write code in. It can help you get to the more complex details faster.

Throughout this book, I'll typically start with the initial module imports, move on to the controller, and finish with the service. I'll update the interface as we go and save the tests for last. But I tend to jump around files *a lot* during real development.

```
// stripe.module.ts
import { Module } from '@nestjs/common';
import { StripeService } from './stripe.service';
import { StripeController } from './stripe.contro

@Module({
  exports: [StripeService],
  providers: [StripeService],
  controllers: [StripeController],
})
```

```
export class StripeModule {}
```

There’s a new environment variable you need to add to `.env` in order to use the Stripe API. You’ll get this value from your account, but there’s a test one you can use that’s found in the docs. So in your `.env`, add the following line.

```
STRIPE_SECRET_KEY="sk_your_stripe_account_secret_
```

Writing the controller

From here, you can jump into your Stripe controller and start writing out endpoints you’ll use. You’ll need an endpoint to take user payments and an endpoint to update your products in Stripe’s system.

So open `stripe.controller.ts` and add this code to start making the endpoints.

```
// stripe.controller.ts
import { Body, Controller, Headers, HttpExceptio
import { StripeService } from './stripe.service'
import { CreateStripePaymentDto } from './stripe
```

```
@Controller('/v1/stripe')
export class StripeController {
  constructor(private readonly stripeService: Str

  @Post('/payments')
  public async createPayment(
    @Body() payment: CreateStripePaymentDto,
    @Headers() headers,
    @Res() res,
  ) {
    try {
      const paymentInfo = await this.stripeServic
        payment,
        res,
        origin: headers.origin,
      });

      return paymentInfo;
    } catch (err) {
      throw new HttpException('Something happened
    }
  }
}
```

In this controller, you've defined one endpoint. The endpoint is how you can handle payments through Stripe. This one is a little different because you'll need to do a redirect in your service method because the user will be sent to Stripe's checkout page through the payment request and then you need a way to send them back to your app after the payment has been made.. You check out why you do the redirect by looking at the diagram on how Stripe handles checkouts.

Keep in mind you have the `CreateStripePaymentDto` in this controller so you need to double check that the interface meets the exact data requirements. As the app grows, you will also have some nested validation so you can ensure you're sending the correct values to Stripe's API.

```
// stripe.interface.ts
import { IsNotEmpty, MinLength } from 'class-vali

export class CreateStripePaymentDto {
  @IsNotEmpty()
  priceId: string;

  @MinLength(1, {
    message: 'quantity has to be at least 1',
  })
```

```
    quantity: number;
  }
```

This validation functionality can also be done with other packages like Joi or Zod in your middleware if you aren't using Nest. `StripePriceData` is something you might use multiple times throughout your service implementation as it grows. A good practice is to mirror the types and requirements from the service docs so you know you're sending exactly what they expect. Sometimes you can simply use the types that come directly from the SDK.

There are also times when you know that you'll only be using certain fields from the types and you'll be adding other fields that don't come from the SDK. You'll have to determine if it's worth the time to fit the SDK types into what you need or if you should write your own. When you know that you'll use the response exactly how it comes to you, go ahead and use the SDK types. If you know that you will have some new combination of fields or the SDK types need to be in a different format, consider mirroring the types to build your own interface.

NOTE

Not every service is as developer-friendly as Stripe. If you aren't sure what types the request expects or what the response will be, just test out the service and see what you get back. Sometimes third-party services require a discovery phase so that you know what you're working with and how you can add the correct types to your app. You might use tools like mitmproxy or even Postman to see what you'll be working with.

Writing the service

Now that you've written out a bulk of the code for the controller, turn your attention to the service where you'll work directly with Stripe. They have a nice npm package that will let you use the API.

Let's implement the ability to make payments in Stripe. Here are the docs for the checkout API to handle payments so you can quickly reference them as you write the code. In `stripe.service.ts`, you'll do some file setup like import packages and add your constructor and logger. Then you can add the method to handle the request to the Stripe checkout API for your payments.

```
// stripe.service.ts
…

@Injectable()
export class StripeService {
```

```
  private readonly logger = new Logger(StripeServ
  // Keep the version string like this because St
  private stripe = new Stripe(process.env.STRIPE_

  public async createPayment({
    payment,
    origin,
    res,
  }: {
    payment: CreateStripePaymentDto;
    origin: string;
    res: any;
  }) {
    this.logger.log('Started payment in Stripe'),

    try {
      // Create Checkout Sessions from body param
      const session = await this.stripe.checkout
        line_items: [
          {
            // Provide the Price ID (for example,
            price: payment.priceId,
            quantity: payment.quantity,
          },
        ],
        mode: 'payment',
        success_url: `${origin}/?success=true`,
        cancel_url: `${origin}/?canceled=true`,
```

```
        });

        res.status(303).redirect(session.url);
      } catch (err) {
        throw Error('Something happened with Stripe
      }
    }
  }
```

One of the biggest things to note here is the logger that's been initialized in the service. The logs you write will be invaluable when you're debugging issues because you can create a record of every action that happens. When you combine this with another third-party service like Datadog, the insights you get will help you fix issues with precision.

Between the logs and the error handling, this is the heart of third-party service integration: making sure that your code can handle any issues that come up from the third-party code. One thing to note is that third-party code might break some of your conventions, but that's one of the tradeoffs of using the service.

Something else to keep in the back of your mind are any edge cases that you can think of. What will your app do if the service you need is down? It's important to have this conversation with the Product team as you find out the quirks of the service through all of your testing, which you'll get into in chapter 7.

You'll notice there's a new field that needs to be added to the database. It's the `stripeProductId`. You want to keep track of some third-party data in your system so that you can perform actions through their API and validate any changes. That means you'll need to update the `schema.prisma` and run a migration. I'll leave that to you to do, but you can check your `schema.prisma` against the version in the GitHub repo.

Conclusion

The focus of this chapter was getting comfortable researching and using third-party services in your code. The biggest thing you should remember when working with these services is that sometimes they just don't behave the way you expect. Weird race conditions can happen, data can come back in inconsistent formats, and request parameters can change. This is where error handling and logging will be your best friend.

Issues will pop up with third-party services from time to time and it's ok. It's nothing that you can't handle with some communication among the dev team and product team and a willingness to try different code strategies.

Chapter 6. Background Jobs

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

Once you have some third-party integrations, it's likely that you'll end up needing automated ways to keep the company's data in sync with the third-party's data and to trigger other events like sending emails. That's where background jobs and cron jobs come in.

A *background job* executes code or actions that need to run outside of the flow of an API in a systematic way. This will include tasks like sending emails or doing complex calculations after a user has made an update. Background jobs are going to work alongside your server-side application

code and are typically triggered directly from an endpoint call. These jobs are usually defined at the service level because they usually have business logic around them.

A *cron job* is a task that is executed on a schedule. They'll run at some time interval you set based on the business needs. Syncing data from or to a third-party service on a schedule is a prime example of using cron jobs. These can be in your application, but it's more common and better practice to have them handled somewhere else like an AWS EC2 instance or just another server. That way you won't use up resources that your application is running with.

One of the hardest things about working with background jobs or cron jobs is that since they run on their own schedules, they can affect the state of the whole system in unexpected ways. That's where your senior dev skills come in. In this chapter, you'll dive into:

- Incorporating background and cron jobs in your app architecture
- Alerts, monitoring, and logging for background and cron jobs
- Handling data sync issues with third-party services and cron jobs
- Handling task execution issues with background jobs
- Future-proofing your background and cron jobs

The first thing to note is that your jobs can be written in any programming language. So you aren't doing anything drastically different, we're still

going to use TypeScript. This is just code that gets run somewhere aside from your endpoints or service logic. This is why understanding the overall architecture of the app is important because jobs can get lost over time as developers come and go and the product changes.

As a senior dev, you'll be expected to be able to make and document those architecture changes and explain how they will affect resource allocation. You might even be responsible for the initial research around the tools that work best with your infrastructure. This is where it's great to talk to the DevOps team or whoever manages the infrastructure.

Find out what cloud providers are used to handle all of the apps for the company. This will give you an idea of the tools you have available and how you should approach writing the code for your jobs. The tool you use will have a direct impact on your implementation because you have to use their SDK, the interface from their resources to your code. You'll learn more about this as we go through this chapter, but let's start with the architecture before going too deep.

Updating the Backend Architecture

For this project, you'll implement a background job to send emails and a cron job to sync data from Stripe to your database. To make sure you and everyone else understands where these fit into the backend, you need to update the architecture diagram. Whenever there is a question about how

things go together and why resources are used a certain way, the architecture diagram should be the source of truth.

This is another document that it'll be essential for you as a senior dev to make sure is maintained. This is also a good place to see if there are any initial concerns with how a new feature will get integrated into the backend. Here's what the diagram will look like now.

NOTE

Updating docs is an incredible opportunity for you to really see if you know what you're talking about. As a senior dev, mentoring more junior devs is so important that it can't be overstated. Take architecture updates as a way to introduce new concepts to the team and to make sure everyone, including you, remembers how this whole product works. You might even run into some areas you need a refresher on!

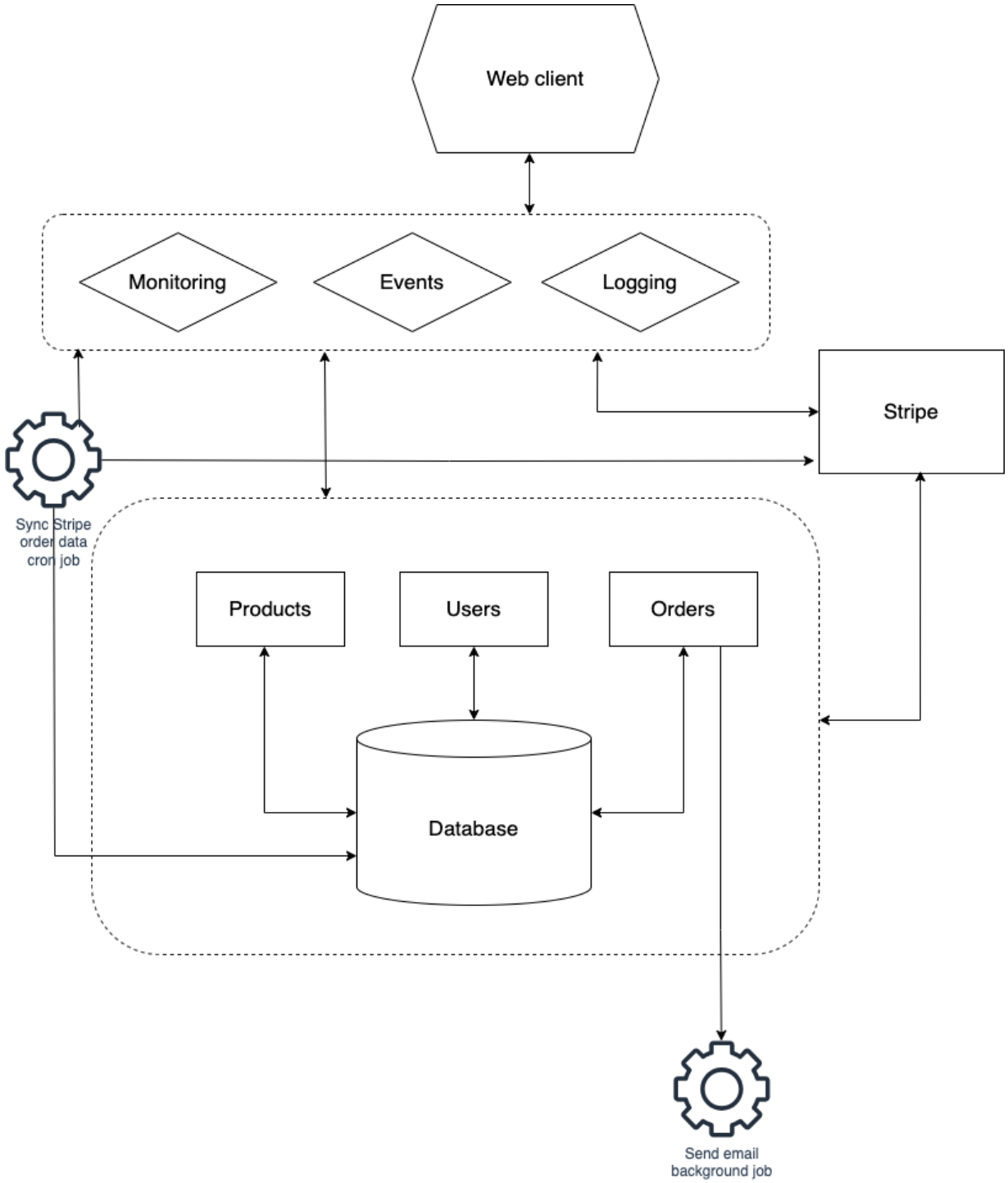


Figure 6-1. Updated architecture diagram with background and cron jobs

Now you have a visual representation of the flow of the jobs. The background job to send emails will only get triggered when something

happens in the orders API. The cron job to sync Stripe data, like products, with your database will happen on a schedule.

Since you have an idea of how these jobs will work within the existing architecture, it's time to take a glance at the infrastructure.

Using Cron Jobs

This is where things get really specific so make sure you have input from the DevOps team. For this project, we'll assume that your infrastructure is built on AWS. You won't get a deep dive into the services in this book other than some quick implementation. If you want to learn more about a specific cloud provider, I suggest starting with their docs, blog posts, and free training materials.

Often, cron job scheduling is handled through the cloud configurations that you usually won't touch. You'll be responsible for the code that runs though. The example cron job you'll implement to sync the Stripe data will be in the repo with the rest of the project for now. This is a good way to test that it works as expected before you start updating the infrastructure.

Just to see what a cron job can look like, here's a code snippet from the project you're working on. You can find the complete file here: https://github.com/flippedcoder/dashboard-server/blob/main/src/jobs/data-sync/data-sync.cron.ts

```
@Cron('5 00 * * *')
  async cronSyncStripeOrders() {
    this.logger.debug(`Stripe orders sync started
    const today = new Date();
    const yesterday = new Date(today).setDate(toda
    // Set params to get the invoices that have b
    const params = {
      created: {
        gte: yesterday,
      },
    };
    // Get invoices from Stripe
    const { data: latestInvoices } = await this.st
    // Check that there are new invoices to proce
    if (latestInvoices.length === 0) {
      this.logger.debug(`No new Stripe invoices s
      return;
    }
    // Loop through invoices and create new DB re
    for (const invoice of latestInvoices) {
      const orderRecord = {
        name: invoice.account_name,
        stripeInvoiceId: invoice.id,
        total: invoice.total,
      };
      try {
        await this.prisma.order.create({ data: or
      } catch (error) {
```

```
          this.logger.debug(`Something happened wit
        }
      }
      this.logger.debug(`Stripe orders sync finishe
    }
```

Let’s break down some of the key things. The first is the cron expression ’5 00 * * *' that defines the schedule for your sync job. You can play with different values on this cron expression site. The expression is set to run the code at five minutes after midnight, every night. That way your data syncs when you can get the most info with the least impact on your users. Keep timezones in mind here because that can have a surprising impact on your system. If you have the code running on a server in UTC-5 and the user lives in UTC-8, that can cause some issues with data syncing on time.

Alerts and Monitoring

There’s another almost hidden part happening with this cron job. As those errors are being logged, certain ones may be more urgent than others. That’s where your monitoring tools will come in. This is something that will likely be handled by the DevOps team because they are responsible for things like resource management and infrastructure-level security. Although when programmatic issues come up, it’s on your dev team to solve them and figure out the root cause.

You’re watching for things like a number of errors being thrown in a specified time range, a specific type of error that occurs, or something else you and the DevOps team have agreed on. Monitoring can be used to trigger alerts which helps the team address issues as soon as they come up. Monitoring usually involves agreed-upon metrics and it can also help establish baselines for how the app should perform.

Most of the time, those alerts will be configured in the infrastructure. If you have a job that starts to trigger a lot unexpectedly, the DevOps team may have an alert that lets them know when a certain resource utilization level is reached. Or if you start seeing a lot of 403 errors in your monitoring tool, it could indicate an attack on the system. There are also alerts that the dev team will need, like code continuously throwing errors.

Between logging, monitoring, and alerts, you should be able to start triaging an issue as soon as it comes up and have the info you need to find a root cause.

Logging

The next important thing to note is the logging that’s in place. When you have jobs running automatically, they will have errors from time to time. Debugging scheduled jobs can be hard for a number of reasons. Things like the service throwing errors or being down, records getting skipped, or database issues happen.

Sometimes you can’t replicate the errors because they happen under specific conditions. That’s where going through logs comes in and we’ll get to that in chapter 9. There are a number of logging lines in here because these are the places in the job where errors may happen and cause other issues. As you think about what to log, think about the info you might need to debug errors. Things like ids, statuses, names, etc are all things you can look up across different parts of the system.

Another thing that can help is to put specific searchable terms in your logs. As the product grows, there will be a ton of logs generated every time some endpoint is called. So being able to quickly filter the logs down to a specific event is essential for concise debugging.

Data Sync and Task Execution Issues

Once you have a root cause, or the source of the error in the code or environment configs, and you’ve figured out the fix, you need a way to retrigger the job that failed. It’s a great backend practice to have a retry mechanism for any cron jobs or background jobs. When you run into data sync issues, usually retrying to fetch the records will fix that issue. In your retry function, you might do a quick count to see how many records were updated so you know the impact of the issue before fixing it.

I’m not going to walk through the code for the background job that will send emails, but you can find it here:

https://github.com/flippedcoder/dashboard-server/blob/main/src/jobs/background/emails.service.ts

For jobs that are time-sensitive and communicate important info to the customers, your alerts will be helpful here for the dev team. This can involve an automated response to the customers or the dev team letting the Customer Support team know something has happened. You can even get to the point that your monitoring metrics can be used to trigger retry events automatically. Most of the time you'll work closely with the DevOps team to debug issues with jobs because of the way they integrate with the infrastructure.

Future Considerations

There will be more errors that happen as the amount of jobs and the amount of data they handle increase with users. You're at a great point in the project to build a stop mechanism for your jobs. Sometimes jobs just execute unexpectedly and you need a way to stop them quickly. You can have stop mechanisms for each job or specific jobs that are more impactful on the overall system. If you start by building this functionality into your first job, when it inevitably gets copy-pasted to create a new job, it'll already have this handling baked in.

When you create jobs, there's a chance that the code won't get updated as often as it should as long as the jobs run successfully. To prevent the code

for your jobs from becoming legacy code, always have a ticket somewhere in your backlog to update the packages and do refactors for the jobs. Doing this at least once a quarter should be frequent enough to keep the code maintainable.

Conclusion

In this chapter, you were able to write a cron job to handle syncing your Stripe data to the company database and a background job to handle sending emails to users for orders. This is where having a solid understanding of the architecture becomes important so that functionality doesn't get lost or forgotten about. You also can tell how much the dev team and the DevOps team work together.

You don't have to be a DevOps engineer to get a basic understanding of how the infrastructure works and how to use the tools that have been set up. As a senior dev, you might feel like you should know how to do everything. You don't. Learn how to lean on other teams for their expertise to help your team get things done more efficiently.

Chapter 7. Backend Testing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo is available at *https://github.com/flippedcoder/dashboard-server*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *vwilson@oreilly.com*.

At this point in the project, you've got quite a bit of code. There are services, controllers, and integrations. While you and the team will do your best to not introduce code regressions (where previously working functionality is broken from an unrelated change), it will happen at some point and it's normal as the code grows. That's why you write tests for the major functionality of the app.

On the backend you'll test for things like errors being called in the correct scenarios, data being returned in the correct format, and that the correct

methods are being called with the correct parameters. Writing unit tests like these will help keep you from making regressions, they'll help you understand the way the code should work, and they will help make the code maintainable because they help you write more concise code.

In this chapter, we'll cover:

- Tradeoffs between having tests and not
- Using Jest to write tests
- The importance of mock data

Regardless of whether you are starting a project from scratch or inheriting one, tests can be a great help. This is also a great time to collaborate with the Product team on the current and future state of the product. Many times writing tests will even bring up more detailed questions about exactly what is expected from some functionality.

Why spend the time on tests

You'll find that developers have differing views on how tests should be implemented. Some strong opinions can come up here because just like with everything else, there are tradeoffs. A big consideration is the level of test coverage for a project. I've seen on many projects that getting close to 100% test coverage on the backend is something that isn't far fetched, but aiming for 90% or higher is more reasonable. There comes a point where

you and the team have to decide if more tedious tests are worth writing by looking at the value they add compared to the amount of time they take to write.

When you take the time to write tests, it does slow down release cycles initially. Tests can take a while to run in your CI and they do take time away from feature development or refactoring. I think that the benefits outweigh the costs, but you do have to maintain a balance between the costs and benefits. So while you might trade off some speed in the beginning, I've seen how you can gain it back tenfold as the app grows. Having tests improves the developer experience for the team because you all can make changes without as much worry about regressions and things breaking in unexpected places.

Tests give you a chance to think through and document the way the code should work. It can even bring up more product related questions. You might even run into the cases where tests can lead to questions that shake up your architecture. Having automated regression tests in place also helps you keep preventable bugs from getting to production.

This is a good place to pair with the QA team, if you have one. If you don't, then you're building in some level of QA for your team. Most of the time, the dev team will be focused on writing unit tests. Unit tests are what you use to make sure the code is implemented properly and running as expected for different use cases. The QA team can come in and write some automated

end-to-end (E2E) tests to run against your development environment. An E2E test is more focused on testing the user flow instead of the code. Some good tools for this are Postman and Thunder Client. You can write collections of tests to hit your development environment to make sure endpoints are really working as expected.

Having a way to validate that your endpoints are manipulating data correctly becomes a necessity as the product grows and new business rules come up. Sometimes things can slip through manual testing as people come and go. Having test cases set up for both unit tests and E2E tests is a form of documentation that is kept up to date consistently.

Whenever a new change breaks a test, you can see what the initial assumptions were and then go double check with the dev team or the Product team to see if those assumptions are valid. Again, writing tests with this level of thoroughness can slow down releases because you have to account for the time it takes to write them *and* get them to pass.

Implementing code and deploying it without tests can happen really fast when you need to get features and bug fixes out. The trade off is that you don’t have the assurance that you didn’t break something seemingly unrelated. For example, you may update the data schema for orders and rename a field. This can break things you wouldn’t expect, like functionality for retrieving users.

You might think you can come back and add tests later as long as you get this next release out on time. This is a risky assumption to make. It’s best to write the tests along with the implementation so that you don’t miss anything before releasing. Plus, things like tests tend to get pushed to the backlog as bugs and new features come along.

Work with the Product team to help them understand why writing tests is helpful for getting a better product to the end user. Show them how it will help prevent the need to do hotfixes for unexpected regressions. Once they see the value tests add, they will start asking about different scenarios that will help you write even better test cases.

How to approach test writing

Although they can be helpful, you don’t need to follow a fancy approach like test-driven development (TDD) or behavior-driven development (BDD). The important thing is to make sure you know what to write tests for. That’s a big part that tends to get left out in most test courses or blog posts. When you write unit tests on the backend, you’re checking for specific functionality.

When you make a request to an endpoint, what exactly is supposed to happen? That means you can need tests for the potential errors that could be returned in the response. It also helps to make sure the functions you expect

to get called actually do, they get called with the correct parameters, and they return the expected data.

Go through your code line by line. Is there an error being thrown? Write a test for it. Is there a function being called? Mock the function and write a test to see if it gets called. There's a balance of knowing when to mock functions or let the actual function be tested. If you find that you're trying to test a package you've installed, then that's a good case for mocking the functionality. Usually if it's internal code that you and the team have written, it's ok to let the tests call the actual code. By having these tests to start with, you can save yourself a lot of debugging time when you and other devs start adding new functionality and code gets refactored.

Any time a change is made that makes the tests fail, it will help you get to the root cause faster. This also helps you think through the way the code should work as it grows over time. You might even find areas where the logic can be improved as you handle failed tests and think about how the code should really work. Your tests should focus on where the logic is, so things like conditions in your code, when HTTP status codes will be returned, how user permissions affect the results of API requests, and anything else that will change the outcome a user or service receives.

Let's take a look at the `create` method in `orders.controller.ts` and see how you would write unit tests for this.

```
@Post()
public async create(@Body() user: User, order:
  if (!user) {
    throw new HttpException('Unauthorized', Htt
  }
  if (!user.permissions.includes('get:orders')
    throw new HttpException('Forbidden', HttpSt
  }
  if (!order) {
    throw new HttpException('No order data', Ht
  }
  if (!order.name) {
    throw new HttpException('No order name', Ht
  }
  if (!order.total) {
    throw new HttpException('No order total', H
  }

  try {
    const newOrder = await this.ordersService.c
    return newOrder;
  } catch (err) {
    throw new HttpException('Something happened
  }
}
```

You can see the errors that may get thrown before making the call to the service method. If any of those conditions are met, then an error will be sent to the frontend in the response. You can be explicit in the error message or leave it more generic. As long as you aren't revealing any personal identifiable information (PII) data or anything specific enough to give attackers useful info, you can say just about anything.

It's a good practice to keep the message short and send the associated status code. Now you can look at this from a testing perspective and go through the method line by line. By reading the code, you can identify 5 test cases from the errors being thrown, a test case for a successful service call, and a test case for an error being thrown from the service. Let's look at how you would write these 7 tests with Jest.

```
import { Test, TestingModule } from '@nestjs/test
import { OrdersV1Controller } from './orders.cont
import { OrdersService } from './orders.service';

const user = {
  id: 1001,
  email: 'tester@rest.com',
  name: 'Tester Rest',
  permissions: ['get:orders', 'create:orders'],
};
```

```
const order = {
  name: 'Biggest order',
  total: 125.99,
  stripeInvoiceId: 'stripeInvoiceId',
};

describe('OrdersController', () => {
  let controller: OrdersV1Controller;
  let ordersService: OrdersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.crea
      controllers: [OrdersV1Controller],
      providers: [OrdersService],
    }).compile();

    controller = module.get<OrdersV1Controller>(C
    ordersService = module.get<OrdersService>(Orc
  });

  it('throws unauthorized error if the user is un
    await controller.create(undefined, order);

    expect(controller.create).toThrowError('Unaut
  });

  it('throws forbidden error if the user does not
    const badPermissionsUser = {
```

```
      id: 1001,
      email: 'tester@rest.com',
      name: 'Tester Rest',
      permissions: ['get:orders'],
    };

    await controller.create(badPermissionsUser, c

    expect(controller.create).toThrowError('Forbi
  });

  it('throws bad request error if the order is un
    await controller.create(user, undefined);

    expect(controller.create).toThrowError('No or
  });

  it('throws conflict error if the order name is
    const orderWithoutName = {
      total: 125.99,
      stripeInvoiceId: 'stripeInvoiceId',
    };

    await controller.create(user, orderWithoutNam

    expect(controller.create).toThrowError('No or
  });
```

```
    it('throws conflict error if the order total is
      const orderWithoutTotal = {
        name: 'Biggest order',
        stripeInvoiceId: 'stripeInvoiceId',
      };

      await controller.create(user, orderWithoutTot

      expect(controller.create).toThrowError('No or
    });

    it('successfully creates a new order', async ()
      controller.create(user, order);
      const newOrder = await ordersService.createOr

      expect(ordersService.createOrder).toBeCalledW
      expect(ordersService.createOrder).toReturnWit
    });

    it('throws not found error if something happens
      try {
        controller.create(user, order);
        await ordersService.createOrder(undefined),
        expect(ordersService.createOrder).toThrowEr
      } catch (e) {
        expect(e.message).toBe('Something happened
      }
```

```
    });
  });
```

All of these tests came from scenarios described in the controller code. When you're writing tests in general, you typically don't need to worry about testing the implementation for things like helpers and service functions because they will have their own unit tests. That's one of the benefits of a modular approach like this. It makes writing tests clearer because you have a separation of responsibilities.

For each scenario, you have test data and you have the expected response. This is a good way to ensure that you get the main cases. While you should try your best to anticipate edge cases when you talk with Product and QA, you can also add edge cases to these tests as you find them.

NOTE

One thing that helps me a lot when writing tests is to have the test file and the code file open side by side. That way I can see if I've tested each line of code as I move through the file.

There are a few ways to approach E2E API testing. You can use a tool like Postman because it's friendlier to work with for non-devs. Or you can use a tool like Cypress or Playwright. Here's what the tests would look like in Postman. You can find the JSON file for these tests here. You can import

this into Postman and check out the tests yourself. Something I'll go over in a later chapter is running E2E tests in your CI/CD pipeline.

NOTE

E2E testing is an ambiguous term because it depends on who you talk to about it. Some would argue that E2E tests are only when you test both the frontend and backend together. Others will say that E2E tests can be divided between the frontend or backend if they focus on functionality instead of code implementation, like with unit tests. Just keep in mind that there isn't a specific definition for E2E tests that's more correct than another. As a senior dev, you have to understand that the distinction may depend on the organization.

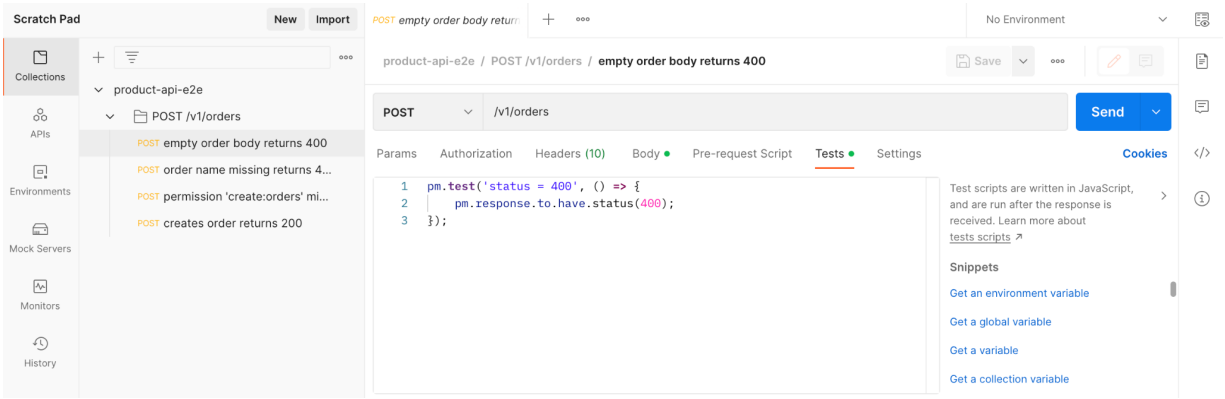


Figure 7-1. E2E tests in Postman

If you decide to go the programmatic way with something like Cypress, the tests will look similar to this.

```
it('/v1/orders (POST)', () => {
```

```
      return request(app.getHttpServer()).post('/v
    });
```

So the syntax is very similar to what you'll be writing your unit tests with. With the programmatic approach, make sure to factor in the time it will take for the team to get ramped up on how to use a particular library. But the scenarios will be very similar across your unit and E2E tests. The unit tests will check code implementation and data manipulation and the E2E tests will check for the flow of the data at a higher level similar to what the frontend or another service would do.

Mocking data

You may have noticed that there was some mock data in a few of those test cases. Your mock data should account for different scenarios that can happen with the project. For the highest level of documentation, it helps having the mocks in separate files with very specific names. Although sometimes the difference between one test case and another is a single field. This is where some of those code conventions might need some updates. As long as the team agrees on the best approach for the scenario, then that's what works.

You can get the mock data directly from your database or by using a tool like Faker or Falso. This is also a great time to double check your types as

well. As you make this mock data, work with the Product team to decide what should happen in different scenarios. It's ok to make assumptions, but verify that they are valid for the product.

Something else you should consider with your mock data is creating test data in your database. This is how you can have mock data available for your E2E tests. This is great for demos to the Product team and any other stakeholders because you can go through the entire flow without having to do code hacks. It also gives them a chance to test things out for themselves and see if anything needs to be changed.

A good way to start mocking data is to make a complete object for each model. For example, you made the user mock data here.

```
const user = {
  id: 1001,
  email: 'tester@rest.com',
  name: 'Tester Rest',
  permissions: ['get:orders', 'create:orders'],
};
```

This has everything you need based on the user schema. Now if you need to modify fields, you can just update this object in the test to account for that

specific case. You have an example of how this works in the user permissions test case where the `permissions` field is modified.

NOTE

Another good way to start mocking data will actually be from your database seed data! This is also a good place to call out that your database seed data should be updated along with any schema changes. That way when you need to set up the project on a new computer you won't run into missing data or incompatible types.

Conclusion

In this chapter we covered testing on the backend and why it's important. You'll run into some developers that haven't written tests with this much detail before and they might push back. Just keep in mind that every time you deploy a bug and it gets found, you cause a mini panic within the company and make the need for hotfixes. Tests help you avoid that even if they do take extra developer time.

We also covered several different ways to test your code. Unit tests are there for the details in the code implementation and E2E tests are there to make sure the flow works as expected from a user standpoint. You can even say that any testing done by the Product team counts as user acceptance testing because they are the ones checking functionality from a user perspective. Testing is a group effort so there might be a lot of discussion

with the Product team and any other stakeholders to make sure everything makes sense.

About the Author

Melecia McGregor is a senior software engineer that's worked with JavaScript, Angular, React, Node, PHP, Python, .NET, SQL, AWS, Heroku, Azure, and many other tools to build web apps. She also has a master's degree in mechanical and aerospace engineering and has published research in machine learning and robotics. She started Flipped Coding in 2017 to help people learn web development with real-world projects and she publishes articles covering all aspects of software on several publications, including freeCodeCamp. In her free time, she spends time with her husband and dogs while learning to play the harmonica and trying to create her own mad scientist lab.