Chapter 10. To Production

> Anything that can go wrong will go wrong.
>
> Murphy's law

When you're starting your career as a programmer, production can feel both intimidating and exciting. Production is where your code moves from development into a live environment used by real people. This is the culmination of all the planning, problem-solving, and late nights.

Production represents one of the final stages in the SDLC. It's where your application will face its true test: serving real end users, handling real data, and operating under extensive workloads.

You're leaving the safety net of your local development environment where it's just you and your code and entering a space where actual people are depending on your work. When issues arise, they are visible to everyone. You might think of your codebase as just another application, but in the hands of users, it becomes a tool that drives business value and creates meaningful change in the world.

In this chapter, you'll gain a better understanding and appreciation for production. You will begin to start thinking about building production-ready code earlier on in the SDLC. A key component of this is understanding the fundamental differences between development and production environments and how to bridge that gap effectively.

The Complexities of Production Environments

"Houston, we have a problem." These famous words from the near-disastrous Apollo 13 mission remind us that things can go wrong even in controlled environments. While software development rarely involves life-or-death stakes, unexpected issues can arise when transitioning to different environments, especially production.

Imagine you and your team have been working on a new product for months. You have sufficient test coverage, and your QA teams have put it through the wringer in various environments. You get sign-off from all teams involved, and you're ready to go.

All the right buttons are pushed, and your project takes its maiden voyage to production. At first, everything seems fine as traffic starts flowing in. But suddenly, chaos erupts. Users report bugs, performance issues appear unexpectedly, and your monitoring system along with your inbox becomes flooded with alerts.

What happened? The code worked on your machine. It worked in every environment you tested. If you've experienced this scenario, you're not alone. It underscores why production matters so much. It's not just the final stop for our

code. It’s the real world, the ultimate test, where actual users interact with our applications.

Although your software might work in test environments, its true value is determined in how it performs in production with real users. The differences between development and production often manifest in two ways: users interacting with your application in unpredictable ways, and the infamous “it works on my machine,” where code behaves differently across environments. Let’s explore both challenges and discover some practical strategies for bridging this gap.

Users Are Unpredictable

Developers often fall into the trap of thinking that they can anticipate how their users will interact with their applications. Sometimes you may lack a complete picture of your user base. Sometimes you think you know more than you actually do. Both scenarios can lead to misaligned expectations between developers and users.

While this is absolutely important to the success of a project, there is no number of tests that you can write or scenarios that you can simulate that can replicate the unpredictable nature of real-world usage. This is why users are your ultimate test.

A “Users Are Unpredictable” Story

Dan here. One of my first jobs out of college was as a Level 2 support specialist. A user called me asking why the “drink holder” on his computer was malfunctioning. I replied, “Sir, computers don’t have drink holders.” It turned out he was referring to the CD-ROM drive. This incident perfectly illustrates how no amount of testing could have prepared us for a user mistaking a CD-ROM drive for a drink holder.

The following are factors that contribute to user unpredictability:

Real-world data
: Even the most meticulously designed test data can’t anticipate the creative ways users will interact with your forms. They’ll input data in ways that would seem plausible only in a Hollywood screenplay.

Scale and concurrency
: Real-world traffic can expose scalability issues that weren’t apparent in controlled environments with test data.

Diverse environments
: Real users have all sorts of environments. This includes varying operating systems, system resources, security settings, browser vendor and version, network latency, and more.

Environmental drift
: Different versions of software or libraries in different environments can cause inconsistent behavior that wasn’t present during testing.

Unexpected use cases
: Users might interact with your software in ways developers didn’t anticipate, revealing edge cases or unforeseen scenarios.

To prepare for the unknown of the real world, you can adopt practices like canary releases, A/B testing, and robust observability (comprehensive monitoring through logging, metrics, and distributed tracing). These methods leverage the power of real-world user testing while minimizing risk.

“But It Works on My Machine”

You get a notification for a new issue that was filed in production. You read through the description and realize this is a feature you worked on. You turn to your laptop to run through the scenario and see whether you can reproduce the issue and, of course, you can’t. Works on my machine, must be user error. “Unable to reproduce” is the phrase developers use to close issues in these situations.

In software development, “works on my machine” is a phrase developers use when their code runs correctly on their own computer but fails when deployed to other environments. While “It works on my machine” might help you quickly close a bug report, it does not solve the problem for your end user, and you need to determine what is causing the discrepancies between development and production.

The following are variables that can change between environments:

Environment differences
: Imagine you’ve been developing on your MacBook Pro with 16 GB of RAM, but production runs on Linux servers with different memory constraints. Your local MySQL database might be version 8.0, while production uses PostgreSQL 14. These differences in operating systems, available resources, network configurations, database vendors, and dependency versions can cause code that runs perfectly on your machine to fail spectacularly in production.

Configuration management
: In development, your application might connect to a local database running on your laptop. You might be logging everything to help you debug issues, sending requests to a payment provider’s test API that uses fake credit card numbers. But when you deploy that same code to production, it needs to automatically switch to a secure cloud database, reduce logging to only capture errors, and connect to a real payment processor that charges actual money.

Concurrency and load
: Imagine this scenario: you’ve been testing your shopping cart feature locally by clicking Add to Cart a few times to make sure it works. But on Black Friday, thousands of customers hit that same button simultaneously. Suddenly your database, which handled your solo testing with ease, is overwhelmed by requests. Your server’s memory, which was cruising along at 30% usage, spikes to 95% as it tries to process hundreds of user sessions at once. This is the reality gap between development and production.

Data diversity
: Your local test database probably has a handful of clean, well-formatted records that you created yourself. Maybe “John Smith” has a perfect phone number like “555-123-4567” and a well-formatted email address. But production tells a different story. Real customer data is messy and unpredictable. Legacy systems might have stored data differently that now breaks validation logic. Then there’s the sheer volume problem. Your test queries run instantly against hundreds of records, but production has millions of customers spanning years of business. This is why testing with realistic, diverse datasets matters. Production data will always have a way of surprising you.

External dependencies
: Modern applications are like team sports: no single player is going to win you the game alone. Your application might handle the user interface, but it passes to a payment processing system to handle payments. It might send requests to an external system for email delivery. It might rely on Google

for user authentication. But in production, when one of your “teammates” is having an off day (payment processor is slow, the email service hits rate limits, or the authentication provider goes down), your well-put-together team (application) can still lose the game.

Security constraints
: Production environments are locked down in ways your local machine isn’t. Your laptop might happily accept HTTP requests from anywhere, but production servers sit behind firewalls that block unexpected traffic. Your local API calls work fine over plain HTTP, but production requires HTTPS with valid TLS certificates. That frontend you’re testing locally can freely call your backend, but production enforces CORS policies that might block the same requests. These security layers protect real user data and money, but they can cause your code to fail in production even when it works perfectly on your development machine.

To bridge the gap between “works on my machine” and “works in production,” consider the following:

Containerization
: Use tools like Docker to create consistent environments across development and production, eliminating the “but it works on my machine” scenario.

Observability
: Implement comprehensive logging, metrics, and distributed tracing to understand how your application behaves in production and catch issues before they impact users.

Environment parity
: Maintain staging environments that closely mirror production, allowing you to catch environment-specific issues early.

Continuous integration and continuous deployment
: Practice CI/CD to ensure that your code is regularly tested in production-like conditions. You will learn more about CI/CD later in this chapter.

Remember, your users don’t care that it works on your machine. They care that it works on theirs. These are the complexities of production and some tips on how to bridge the gap between testing environments and the unpredictability of real users. In the next section, you’ll learn how to avoid some of these issues by thinking about them throughout the development lifecycle.

Building Production-Ready Code

If you’re a parent preparing to send your child off to their first day of school, you have to think about so many details. You wouldn’t just wake them up, hand them their backpack, and push them out the door, would you? Of course not. You’d make sure they’re well-fed, dressed appropriately, have all their supplies, and know how to contact you if something goes wrong. You’d prepare them for the world they’re about to face.

The same applies when preparing your code for production. Your code is your baby, and you need to be certain it’s ready to face the unforeseen challenges of the real world. Just as parents can’t predict every scenario their child might encounter, developers can’t anticipate every possible issue in production. However, with proper preparation, you can give your code the best chance of success by considering performance optimizations, environment-specific configurations, error handling and logging, and security essentials throughout the development process.

Performance Optimization

Performance optimization isn't a dial you simply turn up when you want things to run smoothly. It's a mindset you need to adopt before writing any code, and it should be an area of your application you constantly review throughout the development process.

While performance can indeed be about making your code run faster, ultimately it's about providing the best possible experience for your users. You can employ several key strategies to improve performance. Think of these as building blocks: you don't need to implement all of them at once but can gradually incorporate them into your development process:

Asynchronous programming
: Instead of waiting for a task to complete before moving on to the next one (synchronous execution), asynchronous programming allows tasks to run in the background, notifying the main program upon their completion. This is a common solution to this problem, but use it cautiously as it introduces its own complexities and trade-offs.

Reducing network calls
: Every request your application makes has a cost associated with it, and the currency is time and resources. Instead of making multiple smaller calls, consider batching them together into a single request.

Caching strategies
: If you can identify frequently accessed data that doesn't change, you can store it in memory or some type of fast storage system. This will reduce database load and increase response times. You can learn more about the trade-offs of caching in Chapter 8.

Database query optimization
: Use efficient queries, proper indexing, and avoid N + 1 query problems.

Code minification
: In modern frontend development, code minification removes unnecessary characters from your code without changing its functionality. Think of it like removing all the spaces and line breaks from a book while keeping all the words: the meaning stays the same, but it takes up less space. Minification removes characters like whitespace, newlines, and comments to make files smaller and faster to download.

Code bundling
: Bundling combines multiple separate code files into fewer, larger files. Instead of your browser downloading 20 separate JavaScript files, it downloads 1 or 2 bundled files. Modern build tools like Webpack or Vite can perform bundling alongside other optimizations like tree shaking (removing unused code) and minification.

Lazy loading
: Load resources (images, code chunks, components) only when they're actually needed rather than up front. This reduces initial page load time.

Content delivery networks (CDNs)
: CDNs complement optimizations like minification and bundling by serving your assets from locations physically closer to your users, which can significantly reduce load times.

Performance optimization is not just about making code run faster, but about providing the best possible experience for your users. You can do this by changing your mindset and thinking about some of the optimizations you learned about in this section.

Environment-Specific Configurations

You might hear the phrase, “Configuration is code,” but unlike code, configurations vary across environments. Configuration is similar to adjusting your phone’s camera settings for different lighting conditions. Just as settings optimized for bright sunlight might falter in low light, your application needs tailored configurations to perform reliably in each specific environment.

Let’s look at a common anti-pattern and its solution. In the following example, the developer has hardcoded the database credentials for their local machine. Instead of hardcoding credentials like this, favor a configuration that can be set for each environment:

```
// DON'T DO THIS
public class DbConnection {
   private final String url = "jdbc:postgresql://localhost:5432/fose";
   private final String username = "admin";
   private final String password = "secret";
}

// DO THIS INSTEAD
public class DbConnection {
   private final String url;
   private final String username;
   private final String password;

   public DbConnection(AppConfiguration config) {
       this.url = config.getDatabaseUrl();
       this.username = config.getDatabaseUsername();
       this.password = config.getDatabasePassword();
   }
}
```

You just saw an example of hardcoding a database connection string that is specific to a development environment. This doesn’t scale as you move your code into different environments. Here are some more examples of configurations that might change across environments:

- Logging settings
- External service endpoints
- Database settings
- Cache settings
- Email settings
- Security settings
- Feature flags and app behavior

Now that you know that this is something you need to be aware of, let’s look at some ways to configure these items across environments.

Configuration files

We can use configuration files to configure everything from database connection strings to API keys to application-specific settings. While they might be code, they are often human-readable, which makes them appealing. Configuration files also offer you the option to have a file for each environment you are deploying to.

In the following example, we define two configuration files: one for development and one for production. This lets you define the properties for each environment:

```
# application.yaml (dev)
services:
 service1:
   url: http://localhost:8080
   api-key: 123456
database:
 url: jdbc:postgresql://localhost:5432/fose
 username: postgres
 password: postgres
logging:
 level: INFO

# application-prod.yaml
services:
 service1:
   url: http://real-service1.com
   api-key: 789012
database:
 url: jdbc:postgresql://fosedb.com:5432/fose
 username: postgres
 password: f@3ser#4il_m$%@$LW0
logging:
 level: ERROR
```

Environment variables

If you were paying attention to the configuration files in the previous section, you may have noticed some sensitive information in there. Even though these configuration files are specific, they still pose problems. The first problem is that you are exposing sensitive information that will be checked into version control. The other problem is that sensitive information like API keys is usually rotated at scheduled intervals. This means that to change an API key, you will need to redeploy the entire application, which isn't ideal. In this example, the sensitive information is now defined in an environment variable and is no longer hardcoded:

```
# application-prod.yaml
services:
 service1:
   url: ${SERVICE1_URL}
   api-key: ${SERVICE1_API_KEY}
```

Feature flags

Feature flags (also known as *feature toggles*) are a software development technique that allows developers to enable or disable features at runtime without deploying new code. Consider them like a control panel for your application's features.

Feature flags offer several benefits for both your development teams and your users. You can ship code to production while keeping features hidden from users, allowing for safer deployments and better separation of code releases from feature releases. They enable gradual rollouts to specific user groups, letting you test features with a subset of users before full deployment. Feature flags also make A/B testing straightforward by allowing you to compare different user experiences, and they provide the ability to quickly roll back features without requiring a code deployment.

There are three common types of feature flags to consider. *Release flags* help you hide incomplete features in production, keeping your main branch deployable while work continues. *Experiment flags* are designed for comparing user experiences and gathering data on user behavior. *Permission flags* control feature access based on user roles or subscription levels, making them useful for tiered product offerings.

Tip

Feature flags should be regularly reviewed and cleaned up. Temporary flags that become permanent create unnecessary complexity in your codebase. Establish a cleanup schedule that works well for your team and set expiration data when creating new feature flags. Create documentation that clearly states the purpose and expected lifespan of each flag.

Secrets

Secrets are sensitive configuration values like API keys, passwords, and tokens that require special care. Managing secrets properly is crucial for application security and requires following several important practices.

Never store secrets directly in your code or commit them to version control. This security violation can cost you money and damage user trust. If you do commit these to version control, it will require you to rewrite Git history, which can be a complex process. Additionally, sharing source code with third parties becomes a significant security risk when secrets are embedded in the codebase.

Instead, use dedicated secrets management services like AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault. These services provide secure storage, encryption, and access controls specifically designed for sensitive data. Make credential rotation a regular practice, ideally automated through your secrets management platform. This limits the window of vulnerability if credentials are ever compromised.

Access to production secrets should be strictly controlled and limited to team members who absolutely need it. Use the principle of least privilege and implement proper authentication and authorization controls. Finally, maintain separate secrets for each environment the application gets published to. Development, staging, and production should never share the same credentials. This isolation prevents accidentally exposing production environments during development and testing.

As you can see, there are a number of things to remember when configuring your application for multiple environments. The next consideration for building production-ready code is to employ proper error handling and logging to catch issues when they arise.

Error Handling and Logging

As you have already learned, the world of production can be an unpredictable place. Murphy’s law reminds us that if something can go wrong, it will go wrong. In this unforgiving environment, proper error handling and logging are not just good practices; they are essential survival skills for production applications. They enable us to catch issues before they cascade into system-wide failures and provide critical insights when solving the problems that will inevitably arise.

When you begin building out a new feature, it's natural to focus on the "happy path." This is the ideal scenario your users take where everything works perfectly. However, in the real world of software development, this represents only a small portion of what you need to consider. Following the Pareto principle, while the happy path might represent 20% of your code, handling edge cases and errors often accounts for 80% of your time and effort. This is where you'll spend most of your development time and where experienced developers distinguish themselves.

Error handling

So what does effective error handling actually look like in practice? Here are some concrete strategies you'll use regularly:

Graceful degradation
: When a feature fails, provide a simplified version instead of breaking entirely. For example, if your weather API is down, show cached weather data with a note that it might be outdated rather than displaying an error page.

Meaningful error messages
: Replace technical jargon with user-friendly language. Instead of "Database connection failed," tell users "We're having trouble loading your data right now. Please try again in a few minutes."

Logging for debugging
: Record what went wrong behind the scenes so you can fix it later. Log the technical details (like error codes and stack traces) that help developers, while showing users only what they need to know.

Fallback options
: Always have a Plan B. If your primary payment processor fails during checkout, automatically switch to a backup processor so customers can still complete their purchases.

Logging

Effective logging provides insight into production issues. The challenge lies in striking the right balance. Insufficient logging can leave you in the dark, while excessive logging can create noise that obscures crucial information.

While the implementations might vary across programming languages the concept of logging levels is common. Here are the logging levels in Java that help developers categorize and filter log messages based on their severity and importance:

ERROR
: Use for serious issues that need immediate attention

WARN
: For potentially harmful situations

INFO
: Normal business process events

DEBUG
: Detailed information for debugging

When logging, try not to be too verbose. Focus on the critical items. You should not log the following:

- Sensitive information, such as passwords, API keys, authentication tokens, or secrets.

- Personally identifiable information (PII). You will learn more about this later in this chapter.
- Large objects or entire response payloads. Instead, log meaningful information that will help identify the issue at hand.
- Duplicate logging of the same information across multiple layers of your application.

As you just learned, error handling and logging are vital to getting visibility when things go wrong in production. Another important aspect of building production-ready code is making sure that it is secure.

Security Essentials

Security isn't something you think about once and move on with your life. It is an ongoing process that requires vigilance and regular attention. As you deploy applications to production, remember that security should be built into your development process from the start, not bolted on at the end. While you don't need to be a security expert, understanding these fundamentals will help you build more secure applications and make you a more valuable team member. One really good resource for learning about web application security is the Open Web Application Security Project (OWASP). OWASP provides free, practical security guidance that's widely trusted by developers worldwide. In this section, you will learn some ways you can make your applications more secure.

Securing communication with HTTPS

While you're developing locally you might be inclined to use Hypertext Transfer Protocol (HTTP) because it's frankly just more convenient. However, it is important to test with HTTP Secure (HTTPS) before deploying to production. There are tools that allow you to create trusted certifications for local development.

There are certain messages that can be seen by anyone, and there are messages that you want only the recipient to see. Imagine a postcard you send to someone while you're on vacation. The postcard has a picture of the beach on the front and on the back a handwritten message for anyone to see. This is what it is like to send data over HTTP. HTTP transfers plain text data between your browser and the server. Any sensitive information like passwords, credit card numbers, secrets, or personal data is visible to anyone who intercepts the traffic.

If you want to keep your vacation message private, you would put it inside a sealed envelope; this is what HTTPS does for you. HTTPS adds a layer of encryption using Transport Layer Security (TLS) protocols.

TLS certificates are like digital ID cards for websites. When you visit an HTTPS site, the server presents its certificate to prove its identity. These certificates are issued by trusted certificate authorities (CAs) who verify the website owner's identity.

Once your browser verifies the certificate, TLS uses this trusted connection to set up strong encryption for all data transfer. This encryption ensures that your information can't be read by prying eyes and also means that the information can't be modified while it's being transmitted between your browser and the server.

Note

While Secure Sockets Layer (SSL) is an older protocol that TLS has replaced, many people still use the term “SSL” out of habit. SSL has known security vulnerabilities and should not be used in production applications. When someone refers to SSL today, they usually mean TLS, but it’s important to verify that systems are actually using TLS protocols.

HTTPS isn’t going to fix all of your security problems. It is one layer of security that you should be thinking of as you take your applications to production. Here are some best practices you can follow:

- Use HTTPS everywhere, not just for login pages.
- Redirect HTTP to HTTPS automatically.
- Keep certificates up-to-date (consider using automated certificate management tools).
- Use secure TLS versions (1.2 or higher).

Authentication best practices

Security breaches can cause a lot of problems within your organization, like exposing sensitive data, damaging customer trust, and potentially resulting in significant financial and reputational losses. Proper authentication is a good defense against these threats.

Authentication is the process of verifying that users are who they claim to be, while *authorization* determines what verified users are allowed to do. The first and most important rule is: never, under any circumstances, create your own security measures. While crafting your own authentication systems might seem straightforward, security experts have spent decades identifying and mitigating complex vulnerabilities that aren’t always obvious.

Spring Security in the Java world represents over 20 years of security work. It offers proven solutions for common security problems like password storage. It includes password encoders that prevent storing passwords as plain text, which is very dangerous. In the following example, we create a new `BCryptPasswordEncoder`, which will be used to encode passwords:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

And then use that password encoder to encode the password before saving it off to a database:

```
// Create new user entity
User user = new User();
user.setUsername(registrationDto.getUsername());
user.setEmail(registrationDto.getEmail());

// Encode the password before saving
user.setPassword(passwordEncoder.encode(registrationDto.getPassword()));
// Save and return the new user
return userRepository.save(user);
```

Authentication is a vital component to application security, serving as the first line of defense against unauthorized access. Let’s explore the building blocks of a robust authentication system.

Strong password management is essential for protecting your systems against unauthorized access. For example, enforce passwords with a minimum of 12 characters, combining uppercase, lowercase, numbers, and special characters. Check against databases of common or compromised passwords and prevent users from reusing previous passwords. Use modern hashing algorithms with unique salts for each password. A *salt* is random data added before hashing to prevent rainbow table attacks. Encourage and support the use of password managers with auto-fill capability.

Even with all these technical protections in place, remember that passwords ultimately depend on human behavior. The strongest password policy won't help if users write passwords on sticky notes or share them with colleagues. This is why implementing multiple layers of security (like multifactor authentication, covered later) is so important

Securing user accounts requires multiple layers of defense against unauthorized access attempts. For example, implement exponential backoff for failed login attempts, starting with short delays like 5 seconds and gradually increasing to 30 minutes or more. Track failed attempts across multiple accounts from the same IP address and monitor when single accounts are being targeted from multiple locations. Add CAPTCHA after several failed attempts to prevent automated attacks, and alert users when login attempts occur from new devices or unfamiliar locations. Finally, implement secure account recovery flows with appropriate verification steps to help legitimate users regain access while keeping attackers out.

Effective session management protects users while they interact with your system and prevents unauthorized access to active sessions. Use cryptographically secure tokens with sufficient entropy to ensure that sessions cannot be easily guessed or hijacked. Implement both idle and absolute session timeouts. For instance, automatically log users out after 30 minutes of inactivity or 8 hours total, with shorter timeouts for sensitive operations like banking. Maintain a blocklist of revoked tokens until their natural expiration to prevent reuse of compromised sessions.

Adding multiple layers of verification strengthens your authentication security beyond just passwords. Support multiple authentication methods like authenticator apps, security keys, and biometrics to give users options that work for their situation. While SMS and email are convenient and widely used, be aware they are more vulnerable to attacks such as SIM swapping.

You can trigger additional verification based on unusual behavior. For example, require extra authentication when users log in from new locations or devices. Require MFA during account recovery processes to prevent attackers from bypassing your security. Provide secure backup codes so users are not locked out when their primary MFA method is unavailable. Consider allowing trusted device caching with appropriate security controls so users are not constantly challenged on their regular devices.

Safeguarding user data

Your users are trusting you with their data, and you have both a professional and ethical responsibility to protect that trust. Beyond legal requirements, respecting user privacy and data security is simply the right thing to do. Your decisions as a developer can genuinely impact people's lives, financial security, and personal safety.

When you start building applications that handle user data, you will most likely come across the term *personally identifiable information* (*PII*). This type of data can include obvious details like a name and social security number but also less obvious ones like IP addresses or device identifiers. Some data is considered PII when it is combined with other data.

One way to start thinking about this in development is by marking data as sensitive. In the following example, written in Java, a custom annotation marks data as PII. Within that custom annotation, you can use various masking methods:

```
public class UserProfile {
   private String userId;
   private String hashedPassword;
   @Sensitive  // Custom annotation to mark PII
   private String email;
   @Sensitive
   private String phoneNumber;
   @Sensitive
   private LocalDate dob;
}
```

Encryption: Your last line of defense

What happens when you have sensitive data like financial information (credit cards, bank account numbers, tax returns, etc.) and you need to store it in a database? This is information you absolutely should not store in plain text. While in development, you should start thinking of encryption and consider it your safety net. Think of it as a secure vault: even if someone breaks in, they can't use what they find without the key.

The following class provides a secure method for encrypting sensitive data in Java. It uses the Advanced Encryption Standard (AES), a highly secure encryption algorithm that provides both confidentiality and data integrity protection. When implementing AES, you'll need to choose a key length—currently 128 bits provides strong security for most applications, though many developers are moving to 256-bit keys for futureproofing against advancing computing power:

```
class DataEncryptionService {
  private final SecretKey key;
  DataEncryptionService(SecretKey key) { this.key = key; }

  String encrypt(String s) throws GeneralSecurityException {
    byte[] iv = new byte[12]; new SecureRandom().nextBytes(iv);
    Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
    c.init(Cipher.ENCRYPT_MODE, key, new GCMParameterSpec(128, iv));
    byte[] ct = c.doFinal(s.getBytes(StandardCharsets.UTF_8));
    byte[] out = ByteBuffer.allocate(iv.length + ct.length)
                 .put(iv).put(ct).array();
    return Base64.getEncoder().encodeToString(out); // encodes IV||ciphertext
  }
}
```

Compliance requirements

Software compliance means following specific rules, standards, and regulations. This includes both internal company policies and external legal requirements. As you write code, keep these compliance practices in mind:

- Start with privacy by design: build data protection in your systems from the beginning.
- Apply secure coding standards in your daily work.

- Implement proper data encryption for sensitive data both in transit and at rest:
    - In transit: Use HTTPS for all data transmission (this protects data while traveling between user and server).
    - At rest: Encrypt sensitive data stored in your database (this protects data even if someone gains database access).
    - Be extra cautious when decrypting data in server memory, as skilled attackers might find ways to access it there.
- Create clear mechanisms for data deletion and export requests.
- Document all interactions with sensitive information.

You don’t need to memorize every regulation but recognize when your code touches regulated data and ask for help from someone on your team. Security is a team effort. While you’ll implement secure best practices, your organization’s security professionals should be involved in design decisions from the start of each project. As a developer, you won’t always recognize when you’re approaching something that could create security risks, and that’s normal.

The following outlines some key privacy regulations developers should be aware of:

General Data Protection Regulation (European Union) and California Consumer Privacy Act (California)

Major privacy regulations that affect how we handle user data. These regulations essentially require the following:

- Clear user consent for data collection
- The ability to delete user data (“right to be forgotten”)
- Data minimization (collect only what you need)
- Proper data handling and security measures

Health Insurance Portability and Accountability Act (HIPAA)

If your application handles any health information (medical records, insurance data, even fitness tracking), HIPAA applies. Key developer considerations include the following:

- Encrypt all health data both in storage and transmission.
- Implement strict access controls: users should see only their own health information.
- Maintain detailed audit logs of who accessed what health data and when.
- Ensure that any third-party services you integrate with are also HIPAA compliant.
- Never store health information longer than necessary for your application’s purpose.

Payment Card Industry (PCI)

If your application handles credit card data, you'll need to follow Payment Card Industry Data Security Standard (PCI DSS) requirements. While the full standard is complex, here are key points to consider as a developer:

- Never store sensitive authentication data (like CVV codes).
- Encrypt cardholder data during transmission and storage.
- Implement strong access controls.
- Maintain secure systems and applications.

Software bill of materials (SBOM)

An SBOM functions as an ingredient list for your application. It documents every library, framework, and dependency your code uses along with their version numbers. Think of it as an inventory of all building blocks in your software. For compliance purposes, SBOMs matter because security vulnerabilities frequently appear in popular libraries. With an SBOM, you can quickly identify if your application uses affected components and update them promptly. Most development tools generate SBOMs automatically, eliminating the need for manual maintenance.

Warning

Always log security-related actions, but never log sensitive data! When in doubt, consult your security team for guidance. Create a maintenance schedule for updating your dependencies. Set a regular cadence (monthly or quarterly) to review your SBOM for known vulnerabilities and update to the latest secure versions of your libraries.

Remember that security is about layers. No single security measure is perfect, but implementing multiple layers of security makes it significantly harder for attackers to compromise your system. This is something you need to be thinking about during the development process, so try to layer on security to get into a good practice of being security conscious.

Now that you have some security essentials to keep in mind about building production-ready code while still in development, it's time to discuss the deployment process itself.

Deployment Pipeline

A *deployment pipeline* is your roadmap for safely and reliably moving your code from one environment to production. Think of it as a well-oiled assembly line where each component plays a specific role in transforming your code into a running application that real users can access.

Your deployment pipeline consists of four interconnected components that work together like gears in a machine:

Deployment environments

These are the stages where your code will live and run, from your local development setup to the production servers where customers interact with

your application. Each environment serves a specific purpose in validating your code before it reaches users.

Version control strategies

: Coordinates how multiple developers contribute changes to the same codebase without stepping on each other's toes. These strategies become your safety net for managing releases and handling emergencies.

Deployment automation

: Eliminates the manual, error-prone steps of moving code between environments. Instead of following a checklist and hoping you don't miss anything, automation ensures that the same reliable process runs every time.

CI/CD (continuous integration and continuous deployment)

: Ties everything together by automatically moving your code through the pipeline—from the moment you commit changes to when they're running in production.

By the end of this chapter, you'll understand how these pieces fit together to create a deployment process that's both reliable and stress-free.

Deployment Environments

Before you can grasp the deployment process within your company, it's crucial to understand the environments that you will be deploying code to. These will vary across organizations and even among teams within the same organization. Here is a list of common environments you might find and their purposes:

Local development

: This is your local playground where your coding journey begins. While it's your personal workspace for experimentation, remember that your fellow developers share a similar setup.

Testing/QA

: Although you hopefully have a comprehensive test suite to run locally, the testing environment is your first shared checkpoint. This is where quality assurance takes place, and your test suite might behave differently than it does locally.

Staging

: Think of staging as the dress rehearsal for the big show. It's a production-like environment that serves as the final testing ground before going live. This is ideal for performance testing and user acceptance testing.

Production

: This is the main stage and your ultimate goal. Here, real users will interact with the application you've invested so much time and energy in creating.

Earlier in this chapter, you learned about configuration management. A crucial part of moving an application between environments is ensuring correct configuration for each stage. Configuration management should be an integral part of your deployment process, not an afterthought. To understand how to move code from one environment to the next, you must understand what will change between those environments.

Version Control Strategies

Version control strategies become especially important in production environments where code stability, reliability, and deployment efficiency are critical. In production, these strategies enable teams to manage feature releases without disrupting live services, quickly address problematic deployments (either by rolling back to a previous version or rolling forward with a fix), and maintain detailed audit trails for compliance purposes.

Through education or work, you likely have some experience with version control. You've probably used it in personal projects and have a good understanding of basic Git concepts like cloning repositories, adding and removing files, and pushing changes. This foundation is great for getting started, but as you move into an organizational setting, the introduction of teams brings new complexities to version control.

The following are popular workflows for handling these issues:

Git Flow
: Uses multiple long-lived branches to separate different types of work (features, releases, hotfixes). This provides strong isolation between development and production code. We'll take a closer look at Git Flow in this section.

GitHub Flow
: A simpler approach that uses just a main branch and short-lived feature branches. Changes go directly from feature branches to production after review.

Trunk-based development
: Emphasizes frequent integration where all developers work on a single main branch with very short-lived feature branches (often less than a day).

Release train
: Coordinates regular, scheduled releases (like weekly or monthly) where features are bundled together for deployment.

Let's explore Git Flow because its clear structure makes it excellent for understanding how teams collaborate with version control. This branching strategy organizes work by using different types of branches, each with a specific purpose. Some of the branches will remain throughout the lifecycle of the project, while others will exist only long enough to complete a particular task.

We'll explore each type of branch and learn how teams work together without stepping on one another's toes.

Core branches

Git Flow is built around two core, long-running branches that form the foundation of your project.

The *main* branch represents your production-ready code. It is your source of truth. This branch should always be stable and deployable. When working in a team:

Keep the main branch protected from direct commits
: In Git Flow, the main branch receives code that has already been reviewed and tested through the develop branch, so protection here focuses on preventing accidental direct commits rather than code review.

Ensure that the main branch passes all automated tests
: Automated testing is your safety net for detecting problems before they reach production.

Note

The main branch was traditionally called *master*, but many organizations now use *main* to adopt more inclusive terminology. Both terms refer to the same concept.

The *develop* branch is where the day-to-day action happens in Git Flow. Think of it as your team's shared workspace, where individual features come together before they're ready for production.

When working with the develop branch:

Protect the develop branch with code review requirements
: This is where code review actually happens in Git Flow. When feature branches merge into develop, team members review changes, catch issues, and ensure coding standards.

Keep the develop branch always buildable
: While it may contain incomplete features, the code should always compile and pass basic tests.

Test integration regularly
: Since develop contains multiple features being worked on simultaneously, run your full test suite frequently to catch integration issues early.

The develop branch acts as a staging area where features are tested together before being packaged into a release and ultimately merged to main.

Supporting branches

Besides the core branches, Git Flow uses temporary, short-lived branches for specific tasks. These supporting branches help keep work organized and prevent conflicts among team members.

Feature branches are where you'll spend most of your development time. When you start working on a new feature or bugfix, you create a dedicated feature branch from the develop branch. This isolates your work so you can experiment and make changes without affecting other developers or the stable code.

Giving these branches clear names tells other developers (and future you) what work is happening on that branch. Table 10-1 shows examples of good versus poor branch names.

Table 10-1. Branch name examples

Good branch names	Poor branch names
• feature/user-authentication • bugfix/login-timeout • enhancement/performance-optimization • feature/JIRA-123-user-authentication (when using ticket tracking) • bugfix/GH-456-login-timeout (GitHub issue number)	• my-branch • stuff • feature/123 (without context about what 123 refers to)

Beyond feature branches, you may be wondering: "Why not just merge develop directly to main when you're ready to release?" That is a really good question to

ask. Release branches solve several issues that might come up when working in a team environment:

Stabilization period
: While your release branch undergoes final testing and bug fixes, your team can continue adding new features to develop for the next release.

Release preparation
: Version number updates, final documentation, and deployment configuration changes need a dedicated space that won't interfere with ongoing development.

Quality gates
: Many organizations require a "release candidate" phase where stakeholders can approve the exact code that will go to production.

Scheduled releases
: If your team releases every two weeks, you need a way to "freeze" a set of features while continuing development for the next cycle.

Not every team needs release branches; it really depends on your organization's practices. When your team has a formal release process or scheduled deployment windows, release branches provide good separation.

You can also have hotfix branches. You also might be wondering: "Why do I need a separate branch for a hotfix when I can just use develop?" What happens when production has a serious issue, but your develop branch contains half-finished features that aren't ready for production?

Hotfix branches solve this emergency scenario by doing the following:

Bypassing unstable code
: You can fix the production issue without deploying incomplete features from develop.

Maintaining clean history
: The fix gets applied to both main (for immediate deployment) and develop (so it's not lost in future releases).

Speed and focus
: Your team can work on the critical fix without worrying about other ongoing development work.

Minimal risk
: Since hotfixes branch from the stable main branch, you're changing only what's absolutely necessary

In this section, you learned about the branch types in Git Flow. As you can see, each branch type serves a specific purpose in keeping your team's work organized and properly versioned. The core branches (main and develop) provide stability and coordination, while the supporting branches (feature, release, and hotfix) handle specific tasks without disrupting the main workflow. This might seem overwhelming at first, but the more you work with a strategy like this one, the more you will see how it creates clear boundaries that help teams work together safely.

The version control lifecycle

So, how do all of these branches work together to form a version control strategy? In Git Flow and many other strategies, development occurs in feature branches, gets merged into a develop branch, and finally placed into main after release, with hotfixes applied directly to both master and develop when necessary.

The Git Flow workflow is illustrated in Figure 10-1.

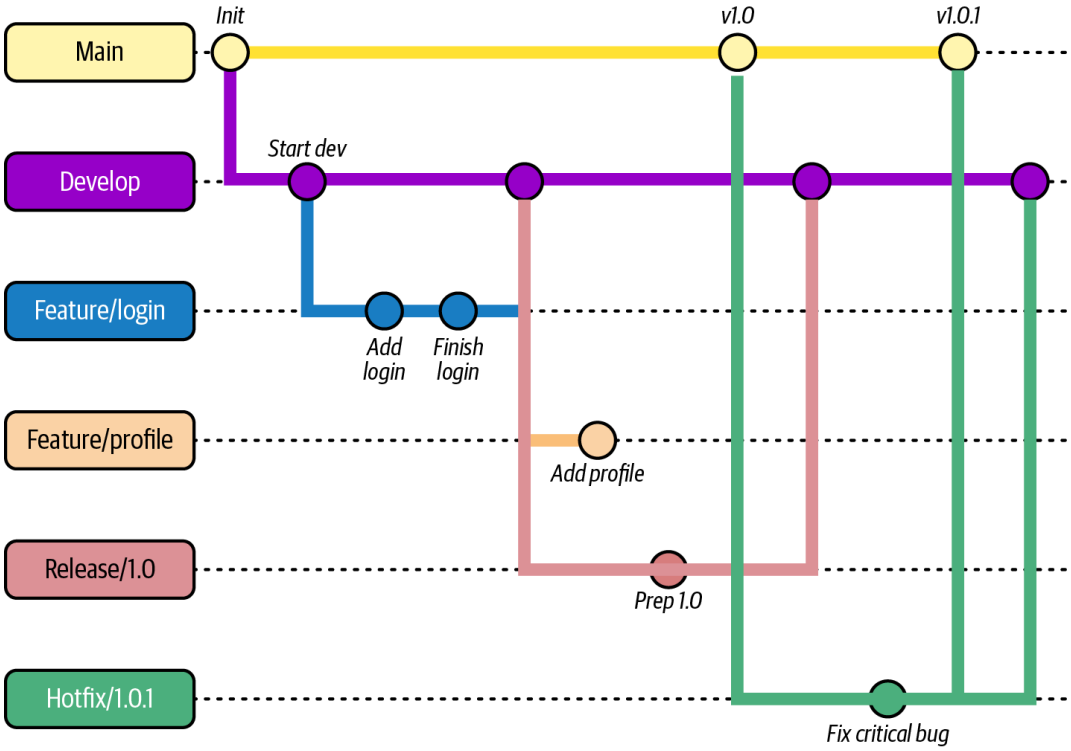

Figure 10-1. Git Flow workflow

By understanding Git Flow, you have a solid foundation for team-based development, but remember that it's just one approach among many. As you work with different teams and projects, you'll encounter variations of these strategies, but the goals will remain the same. Provide a way to protect your code, isolate your work, and create a clear process for integrating changes. In the next section, we'll explore how to automate deployments to ensure that your carefully managed code reaches production reliably and consistently.

Deployment Automation

Manual deployments to production are often time-consuming, error-prone, and stressful. This is because you need to reproduce a deployment step-by-step with human intervention that involves code preparation and packaging, testing and verification, server preparation, deployment execution, post-deployment verification, and more. The good news is that we can address this problem by automating our deployments. In this section, you'll learn about several key elements that will help streamline your deployment process.

Scripting deployments

Manual deployments are like handcrafting a custom bookshelf every single time you need one. Even if you've built the same bookshelf as before, you're likely to miss a step, make small variations, or take much longer than necessary. Deployment automation is like having a factory that can build identical bookshelves to your exact specifications—precise, repeatable, and without human error. A deployment script serves as your factory instructions, documenting every step needed to consistently move your application from development to production.

Here's a simple example of a deployment script that will run steps like creating directories, backing up existing directories, deploying the application, and starting it:

```
#!/bin/bash

# Configuration
APP_NAME="myapp"
JAR_FILE="target/${APP_NAME}.jar"
DEPLOY_DIR="/opt/applications/${APP_NAME}"
BACKUP_DIR="${DEPLOY_DIR}/backups"
LOG_FILE="${DEPLOY_DIR}/app.log"
PID_FILE="${DEPLOY_DIR}/${APP_NAME}.pid"
JVM_OPTS="-Xmx512m -Xms256m"

# Create directories
mkdir -p "${DEPLOY_DIR}" "${BACKUP_DIR}"

# Backup existing deployment
if [ -f "${DEPLOY_DIR}/${APP_NAME}.jar" ]; then
    mv "${DEPLOY_DIR}/${APP_NAME}.jar" \
    "${BACKUP_DIR}/${APP_NAME}-$(date +'%Y%m%d_%H%M%S').jar"
fi

# Stop running application
if [ -f "${PID_FILE}" ]; then
    kill $(cat "${PID_FILE}") 2>/dev/null || true
    rm -f "${PID_FILE}"
    sleep 2
fi

# Deploy new version
if [ -f "${JAR_FILE}" ]; then
    cp "${JAR_FILE}" "${DEPLOY_DIR}/${APP_NAME}.jar"
else
    echo "Error: JAR file not found: ${JAR_FILE}"
    exit 1
fi

# Start application
nohup java ${JVM_OPTS} -jar "${DEPLOY_DIR}/${APP_NAME}.jar" \
    > "${LOG_FILE}" 2>&1 &
echo $! > "${PID_FILE}"

# Quick health check
sleep 5
if kill -0 $(cat "${PID_FILE}") 2>/dev/null; then
    echo "Deployment successful. PID: $(cat ${PID_FILE})"
else
    echo "Deployment failed. Check ${LOG_FILE} for details."
    exit 1
fi
```

While this example is basic, it illustrates key components of deployment automation:

- Clear, documented steps
- Error handling
- Backup procedures
- Systematic approach

Even a simple script is better than no script. Start small and gradually improve your deployment automation over time. Your future self will thank you for putting in the time to automate this process now.

Rollback procedures

We've established that automating your deployment will save time and reduce stress. However, even with a well-tested automated deployment, things can occasionally go wrong.

Nothing is perfect, so you need to be prepared for unexpected issues. Having a plan to quickly revert changes to a working state ensures that your customers can continue using your application without interruption.

Here's an example of a rollback script that will look for the most recent backup, stop the application, restore, start, and then verify that the application is up and running correctly:

```
#!/bin/bash

# Configuration (should match deploy.sh)
APP_NAME="myapp"
DEPLOY_DIR="/opt/applications/${APP_NAME}"
BACKUP_DIR="${DEPLOY_DIR}/backups"
LOG_FILE="${DEPLOY_DIR}/app.log"
PID_FILE="${DEPLOY_DIR}/${APP_NAME}.pid"
JVM_OPTS="-Xmx512m -Xms256m"

# Find most recent backup
LATEST_BACKUP=$(ls -t ${BACKUP_DIR}/${APP_NAME}-*.jar 2>/dev/null | head -1)

if [ -z "${LATEST_BACKUP}" ]; then
    echo "Error: No backup found in ${BACKUP_DIR}"
    exit 1
fi

# Stop current application
if [ -f "${PID_FILE}" ]; then
    kill $(cat "${PID_FILE}") 2>/dev/null || true
    rm -f "${PID_FILE}"
    sleep 2
fi

# Restore backup
cp "${LATEST_BACKUP}" "${DEPLOY_DIR}/${APP_NAME}.jar"

# Start application
nohup java ${JVM_OPTS} \
    -jar "${DEPLOY_DIR}/${APP_NAME}.jar" \
    > "${LOG_FILE}" 2>&1 &
echo $! >  "${PID_FILE}"

# Quick health check
sleep 5
if kill -0 $(cat "${PID_FILE}") 2>/dev/null; then
    echo "Rollback successful. Restored: $(basename ${LATEST_BACKUP})"
    echo "Application running with PID: $(cat ${PID_FILE})"
else
    echo "Rollback failed. Check ${LOG_FILE} for details."
    exit 1
fi
```

Before implementing a rollback script, you should have these prerequisites in place:

- Verify that backup versions exist before deployment.
- Test rollback procedures regularly.

- Document database migration rollback steps.
- Keep multiple backup versions.
- Monitor the system during rollback.

The goal of deployment automation isn't just to save time. It's to create a reliable, repeatable process that gives you confidence in your deployments. Start small, perhaps with a simple script that automates a few steps, and gradually build up to more comprehensive automation. Each improvement reduces risk and brings you closer to the ideal of predictable, stress-free deployments.

Deployment Strategies

There is no one-size-fits-all strategy when it comes to deploying your code into production. The strategy you choose for your application can be the difference between a smooth rollout and being on pager duty all weekend. Let's take a look at some of the most popular approaches and when to choose them.

All-at-once deployment (big bang)

This strategy replaces the entire application in one go. The old version is completely swapped out for the new version at once. It's simple to implement but carries a high risk, as any issues will immediately affect all users. There is no gradual transition period, making rollbacks complex if problems come up. This approach is best suited for smaller applications with thorough testing or in environments where downtime is acceptable.

Big bang deployment is best for the following:

- Small applications with limited users
- Development and testing environments
- Systems that can tolerate some downtime

Gradual deployment (phased approach)

A gradual deployment is like your favorite restaurant introducing a new recipe to a small number of customers at a time. This allows the chef to gather feedback, iterate, and gradually expand the recipe's availability. Gradual deployment follows this approach by rolling out changes to a subset of users or servers before wider release. This controlled approach allows for early issue detection and course correction.

Common examples include *canary deployments* (which deploy to a small percentage of users first) and *rolling deployments* (which gradually update servers one by one). Both can often be done with zero downtime using techniques like blue-green infrastructure or load balancer traffic shifting.

Gradual deployment is best for the following:

- Large-scale applications
- Features that benefit from user feedback
- Systems requiring careful monitoring

Zero downtime considerations

Whether you choose all-at-once or gradual deployment, you can minimize or eliminate downtime by using techniques like blue-green deployment (maintaining two identical environments and switching between them) or rolling updates. While more complex to set up, these provide the smoothest user experience for mission-critical applications and high-traffic services.

Choosing your strategy

When should you choose each approach? By now, you probably have already guessed that the answer is usually "it depends." Consider factors like system complexity, tolerance for downtime, and monitoring capabilities when selecting the right strategy for your application.

Remember: There's no shame in starting simple and evolving your deployment strategy as your application and team mature.

Continuous Integration and Continuous Deployment

In Chapter 5, you learned that automated testing provides confidence during the time of writing and committing code while helping identify issues early in the development lifecycle. But how can you ensure that these tests are run each time a release is promoted to a new environment like production? How do you guarantee that your code is built, tested, and deployed without the errors that manual human intervention can introduce?

This is where *continuous integration and continuous deployment*, or CI/CD (or delivery) comes in. It's a set of automated processes that takes your code from development all the way to production, where your users live. *Continuous integration* automatically combines and tests changes from your entire development team. *Continuous delivery* prepares your tested code for release, while *continuous deployment* can automatically release it to production.

For example, say it's your second week on the job and the team has asked you to help with the deployment of a smaller microservice. You open the lengthy deployment document that is filled with vague steps like "build the application," "run the tests," "update the database," and "deploy to the server." Your panic builds as you check off each item, knowing that if you miss even one little step, you could bring down production.

Sounds pretty stressful, right? This is exactly why you need automation. Automation can eliminate a lot of issues by creating a repeatable, reliable process. The benefits of automation include the following:

Consistency
: Remember the "works on my machine" scenario where the application works locally but has problems when promoted to another environment? When you automate your build process, you minimize that problem.

Automated tests
: Instead of relying on developers to run a suite of tests before committing their code, your CI/CD process can run tests automatically.

Reduced stress
: Instead of worrying about manually checking off each step in a deployment process, you can be sure your automation is run the same way every single time. A script never skips a step or mistypes a command.

Early detection of integration issues
: A well-defined CI/CD process can prevent issues from sneaking into production, saving you time and money.

Reduced time between code and production
: Go to production frequently by leveraging a well-defined automated process that minimizes deployment time. Remember, code has no real value until it reaches end users.

Building a basic CI/CD workflow

A CI/CD workflow is like a recipe: it defines the steps needed to take your code from development to production. Let's look at a basic workflow:

Code
: A developer pushes code to the repository.

Build
: The application is compiled and built.

Test
: Automated tests are run.

Package
: The application is packaged for deployment.

Deploy
: The application is deployed to the target environment.

To put this into practice, let's create a basic GitHub Actions workflow. A *GitHub workflow* is an automated process that you can set up in your GitHub repository to build, test, package, release, or deploy your code. GitHub Actions is a popular choice for implementing CI/CD pipelines because it's easy to get started with and integrates naturally with your GitHub repository.

Here's an example workflow file that implements these steps for a Java application:

```
name: Java CI/CD Pipeline

on:
 push:
   branches: [ main ]
 pull_request:
   branches: [ main ]

jobs:
 build-and-test:
   runs-on: ubuntu-latest

   steps:
   # Check out your repository code
   - uses: actions/checkout@v4

   # Set up Java development environment
   - name: Set up JDK 25
     uses: actions/setup-java@v4
     with:
```

```
        java-version: '25'
        distribution: 'temurin'

    # Build the application
    - name: Build with Maven
      run: ./mvnw -B package --file pom.xml

    # Run automated tests
    - name: Run tests
      run: ./mvnw test

    # Package application (creates a JAR file)
    - name: Package application
      run: ./mvnw package -DskipTests

    # Store the package as an artifact
    - name: Upload artifact
      uses: actions/upload-artifact@v4
      with:
        name: app-package
        path: target/*.jar
```

This workflow file demonstrates several key concepts:

- The `on` section defines when the workflow runs (on push to main or PRs).
- Each job runs in a fresh virtual environment (`ubuntu-latest`).
- Steps are executed sequentially, with each step depending on the success of previous steps.
- The workflow handles building, testing, and packaging our application.

By implementing this workflow, you've automated several critical steps in your development process. When you push code or create a PR, GitHub automatically does the following:

- Builds your application to catch compilation errors early
- Runs your test suite to catch potential bugs
- Creates a deployable package
- Stores the package for later use

A CI/CD workflow automates the journey of code from development to production through a series of predefined steps. This automation eliminates manual intervention, catches errors early, enforces quality standards, and ultimately delivers software more reliably and effectively in a repeatable pattern.

Advanced CI/CD patterns

Don't stress out about having to learn everything at once. Start simple and build on the foundations that you have learned in this chapter. Begin by automating your build and test process. As it begins to make more and more sense, you will feel more confident, and then you can gradually add more advanced deployment strategies.

As you move forward in your career, you'll come across more sophisticated CI/CD patterns like these:

Canary releases

Deploy new code to a small subset of users or servers first to test in production with minimal risk, expanding gradually if no issues are found.

Blue-green deployments
: Maintain two identical production environments (blue and green), with only one active at a time. New versions deploy to the inactive environment and, once verified, traffic switches over completely.

Here are some ways to introduce advanced patterns and gradually improve your CI/CD process:

1. Start with a basic pipeline that builds and tests your code.
2. Gradually incorporate more automated checks and validations.
3. Maintain a fast pipeline to ensure quick feedback loops.
4. Regularly monitor and optimize your pipeline based on team needs.
5. Document your pipeline to facilitate team collaboration.

CI/CD is not just about tools and automation. It is about fostering a culture of continuous improvement and reliable software delivery. By embracing these practices early in your career, you'll build a strong foundation for professional growth and become an invaluable team member.

Production System Monitoring and Maintenance

You have written your code, tested it thoroughly, and now it's ready for production. Your application went through the proper CI/CD pipeline and is now live in production. It's time to kick up your feet and relax, right? Not quite. The journey of a working application on your machine to a reliable system in production is an ongoing one.

Monitoring

Proper monitoring and logging can tell you a lot about the state of your application. Without these in place, you won't know how your system is performing currently or over time. Too many developers want to throw their app into production and hope for the best. They think that no news is good news. The truth is, users rarely tell you when something is wrong. They just leave.

Your application relies on two types of monitoring systems:

Real-time monitoring
: Tracks what's happening right now. Is your application's response time too slow? Is the server load or throughput too high? Individual metrics never tell the complete story. A 300 ms response time might seem fast, but if your application normally responds in 50 ms, this could indicate a problem.

Logs
: Serve as your historical record. When issues arise, good logs help you reconstruct what went wrong and when. They provide the context for troubleshooting.

After experiencing several critical incidents with our application, we established these vital monitoring rules:

1. Log information you will need when problems occur.
2. Always include timestamps and user identifiers.
3. Mark errors as errors. Do not bury important alerts in info logs.
4. Keep private data out of logs (no passwords or personal details).
5. Delete old logs before they consume your storage.

Tip

Keep your monitoring system simple at first. Watch your response times, error counts, and server resources. That's enough to start. The fancy tools can come later.

Good monitoring provides peace of mind. Problems will eventually occur; that is the nature of software. When they do, you'll be thankful you have metrics and logs to help you diagnose and fix issues quickly.

AI Note

Once you have logs and metrics in place, AI tools can significantly enhance your monitoring capabilities. AI can search through large volumes of logs to identify anomalies or patterns that might indicate emerging issues before they become critical. Additionally, AI can help generate monitoring dashboards from your metrics data, which traditionally requires tedious manual configuration, giving you a solid starting point and accelerating the dashboard creation process.

System Maintenance

When a system is deployed to production, it is not the end of the journey; it's just the beginning. System maintenance is a critical aspect of keeping software running efficiently, securely, and reliably. This section covers how to maintain the production system to prevent costly issues and ensure optimal performance.

Keeping systems up-to-date

Software that has been deployed to production needs regular maintenance. While everything is performing well in production, it might be tempting to postpone maintenance, but this would be a mistake. What might start off as minor, inexpensive issues can quickly transform into major, costly problems.

When you are dealing with systems that users depend on, the costs are high. Critical infrastructure requires consistent upkeep to ensure safety and reliability. But somehow in software development, teams will frequently delay necessary updates despite the risks.

System updates

Operating system updates and security patches form your first line of defense for running into major, costly problems. These updates often address the following:

- Critical security vulnerabilities

- Performance improvements
- Bug fixes
- New features and capabilities

Never ignore security updates. While feature updates can sometimes wait, security patches should be applied promptly to protect your systems from vulnerabilities.

Dependency management

Dependency management is arguably the most overlooked aspect of system maintenance. Your application may rely on dozens or even hundreds of external libraries, each potentially harboring security vulnerabilities. Remember the SBOMs we discussed earlier in this chapter? This is exactly why maintaining an accurate SBOM is important for tracking and updating your dependencies.

The following code shows a dependency in a Java app that declares an outdated version with known vulnerabilities:

```
// Example dependency in pom.xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.5.5</version>  // Outdated version with known vulnerabilities
</dependency>
```

When you generate an SBOM for this application, security tools can scan it against vulnerability databases to identify that version 2.5.5 has known security issues and recommend updating to a newer, secure version.

Modern build tools like Maven and Gradle offer dependency analyzers that can alert you to known vulnerabilities. To catch these issues early, set up automated scans. In this example, a GitHub Action runs dependency checks on your code:

```
// Example GitHub Actions workflow snippet
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Dependency Check
        uses: dependency-check/Dependency-Check@main
```

Taking action

You can start to improve system maintenance by establishing an update policy for your team or project:

1. Set regular update intervals.
2. Define emergency patch procedures.
3. Implement automated scanning.
4. Document update processes.
5. Maintain an update log.

Remember, prevention costs less than recovery. Make system updates a priority in your development workflow. Set calendar reminders for regular update reviews.

Despite automation, human oversight remains crucial for maintaining system health.

Wrapping Up

Moving your code from the safety of your local machine to the unpredictable world of production can be intimidating at first. Just know that you are not alone and most developers have felt that same stress at one point in their career. Preparation is the key to ensuring you have a good deployment process in place. Taking code to production is a skill that will improve over time with experience.

Remember these key takeaways:

- Think production ready from day one. Consider performance, security, and error handling while you code.
- Configure your applications properly for different environments by using environment variables and feature flags.
- Monitor your application's health and set up proper logging, but never log sensitive data.
- Use version control strategies like Git Flow to collaborate safely with your team.
- Automate your deployments with CI/CD to make them reliable and repeatable.
- Remember, your users don't care if the application works on your machine. They care if it works on theirs.

As you begin deploying your own applications to production, you will face challenges and sometimes might even fail: that is OK. Each failure you encounter is a chance to learn something new and will make you a better software engineer.

Focus on building good habits now, and production deployments will become a natural part of your development workflow. Before long, you will be an experienced developer helping others ship code to the promised land of production.

Putting It into Practice

It's hard to envision how an entire application will perform in production, so incrementally adopt some of the best practices you learned in this chapter. The following are good habits that can be applied at various points before deploying to production. The next time you're assigned a feature, pick one or two items from each of these categories to apply to your work. Don't try to do everything at once; instead, focus on building these habits gradually.

Before you code:

- Create a simple list of what could go wrong with your feature.
- Plan what information you might need to log to debug these issues.

- Identify what configuration might change from your local machine to other environments in your workflow.

While coding:

- Create a feature branch with a clear, descriptive name before starting work.
- Add basic error handling for the top three things that could fail.
- Include logging statements for key actions (but avoid logging sensitive data!).
- Write a test that mimics real user behavior.
- Use configuration values for anything that might change between environments.

Before deployment:

- Write down your deployment steps as you do them.
- Create a simple rollback plan. How would you turn off this feature if things go wrong?
- Have a teammate review your changes with production in mind.

After deployment:

- Begin monitoring your logs after deployment for an hour or two. Does anything stand out?
- Put together a retrospective about this deployment. This is a meeting or discussion held after completing a project or deployment, where the team can reflect on what went well, what challenges they faced, and what lessons can be learned.

Additional Resources

- *Continuous Delivery* by Jez Humble and David Farley (Addison-Wesley Professional, 2010)
- *The Phoenix Project* by Gene Kim et al. (IT Revolution Press, 2013)
- *Head First Git* by Raju Gandhi (O'Reilly Media, 2022)
- *Learning GitHub Actions* by Brent Laster (O'Reilly Media, 2023)
- *Feature Flags* by Ben Nadel (self-published, Lulu, 2024)