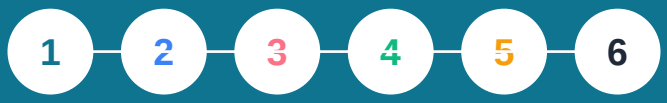
Software Development Life Cycle

A Beginner's Step-by-Step Guide

Project: Build a Travel Website (Wanderly)

Planning - Requirements - Design - Development - Testing - Deployment



What you will learn

How real software is built - from idea to live website.
Writing a project blueprint and an SRS document.
Designing wireframes, mockups, and a database schema.
Using Git and GitHub like a professional developer.
Code reviews, pull requests, and team collaboration.
Testing fundamentals - unit, integration, and manual.
Quality Assurance, bug tracking, and the QA workflow.
Waterfall vs Agile vs Scrum - which fits your project.

egotechworld.com

Free coding tutorials, project source code, and AI tools

Beginner-Friendly - Step by Step - 2026 Edition

Table of Contents

Chapter 1 - What is the SDLC and Why It Matters

Welcome! In this tutorial we are going to learn the **Software Development Life Cycle** (SDLC) - the framework professional teams use to build software that actually works. Instead of reading abstract theory, we will apply every phase to a real project: a travel website for a small Sri Lankan tour company called **Wanderly**.

What is the SDLC?

The Software Development Life Cycle is the journey software takes from "someone has an idea" to "users are happily using it on the internet". Every successful software project follows some version of this cycle. The names and order can vary, but the **six core phases** are remarkably consistent:

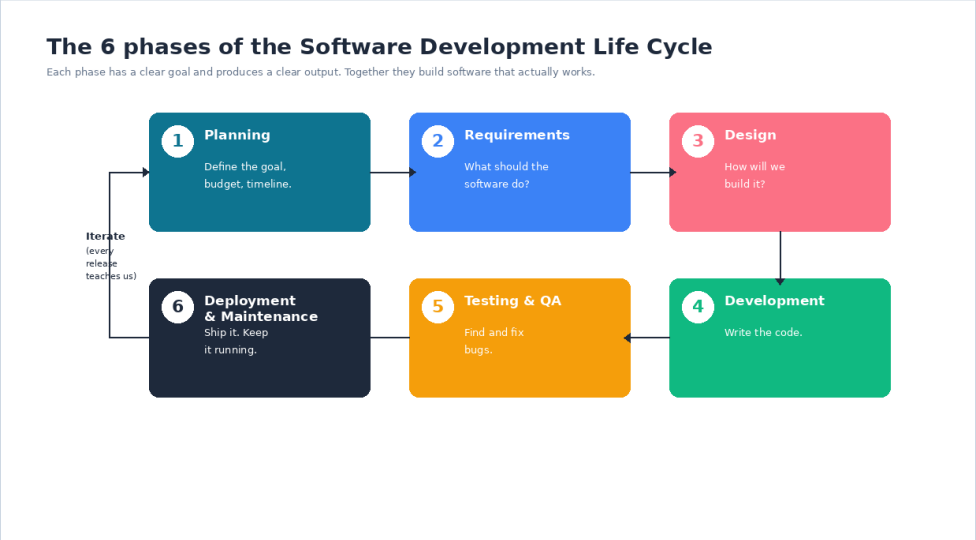


Figure 1 - The six phases of the SDLC. Each one feeds the next.

Why bother with all this process?

When you are coding alone on a tiny project, you can skip most of this. But the moment a team is involved, or the project lasts more than a week, or someone is paying real money for the result - skipping these phases causes pain. Common consequences:

- **Building the wrong thing** - because nobody wrote down what the right thing was.
- **Endless rework** - because the design changed five times after coding started.
- **Bugs in production** - because nobody tested before launch.
- **Code that nobody understands** - because there are no comments, no docs, no version control history.
- **Missed deadlines** - because the team had no plan.
- **Angry clients** - because they were promised features that were impossible to build in the time available.

Following the SDLC does not eliminate these problems. It DOES make them visible early, when fixing them is cheap.

Who this tutorial is for

- Computer science students who learnt theory but never saw it applied.
- Self-taught developers who can code but don't know how teams operate.
- Project managers who need to understand what their developers do.
- Anyone preparing for a software engineering interview.
- Anyone about to start their first dev job.

About the Wanderly project

Wanderly is the imaginary travel company we will design the website for. They specialise in Sri Lankan tours - cultural sites, hill country, beaches. The website needs to:

- Showcase featured destinations on a beautiful homepage.
- List all available tours with filters (region, price, duration).
- Let visitors book a trip by filling out a simple form.
- Save bookings so the company can follow up.
- Look good on phones, tablets, and laptops.

Preview of the finished site

1. The homepage

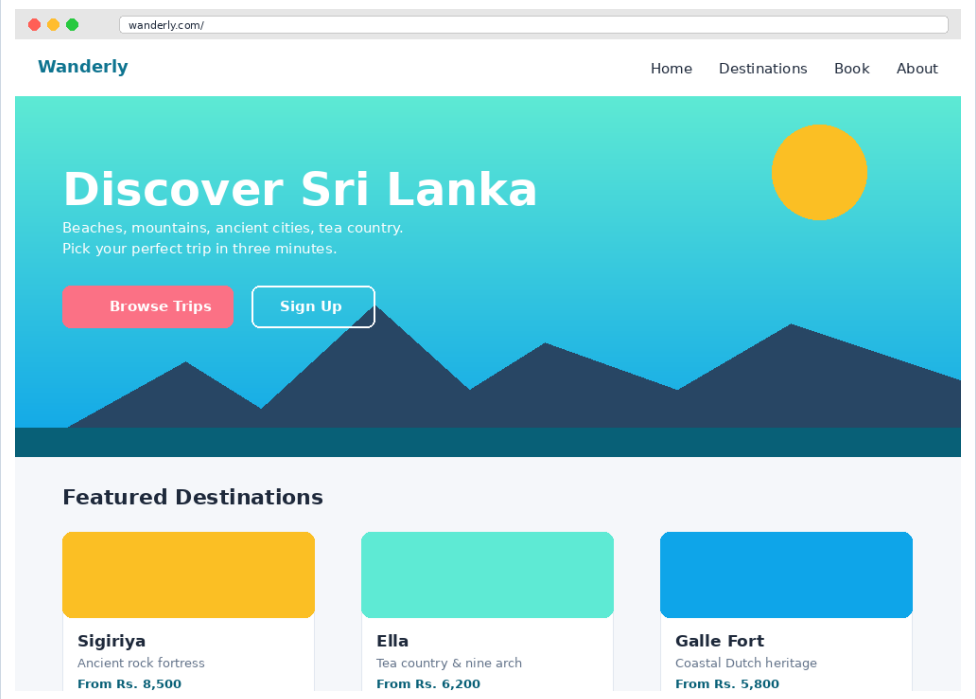


Figure 2 - Homepage with hero banner and featured destinations.

2. The destinations page

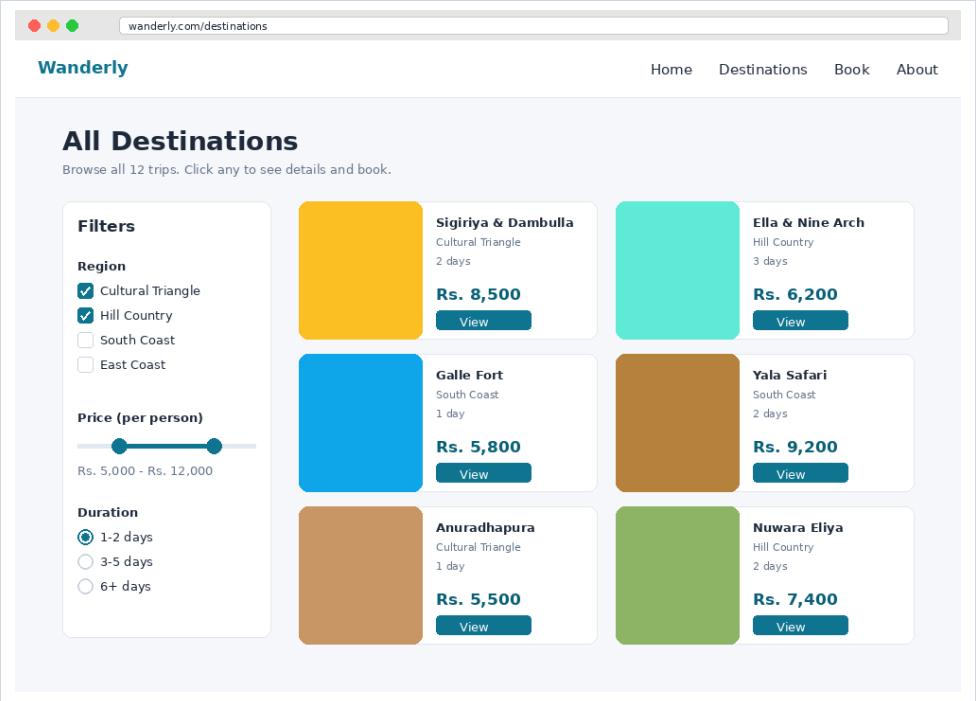


Figure 3 - All trips with filter sidebar (region, price, duration).

3. The booking form

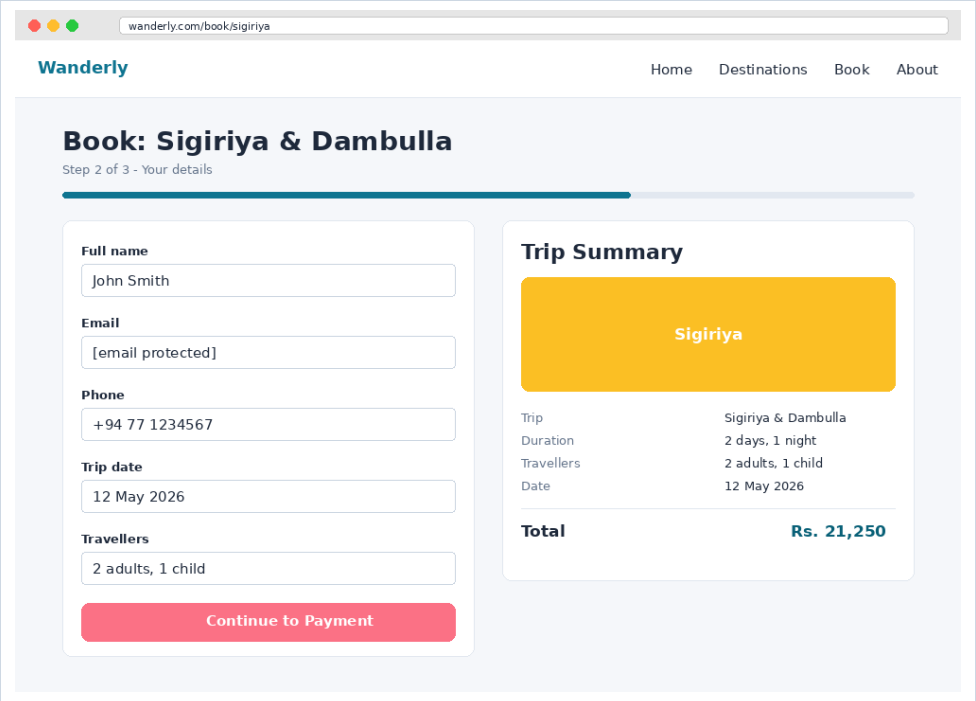


Figure 4 - Two-column booking page: form on the left, summary on the right.

> **TIP**
>
> Wanderly does not actually exist. It is a teaching project. But every step we take is exactly what a real agency would do for a real client. The patterns transfer to any project you ever build.

What you will NOT learn here

This tutorial focuses on the **process and lifecycle**. We will write some HTML, CSS, and JavaScript along the way, but the goal is not to teach you to be a senior frontend developer. For that, see our other tutorials at **egotechworld.com** on React, Laravel, Node, and Python.

How the rest of this guide is structured

Each upcoming chapter takes one phase of the SDLC, explains it, and shows the output it produces for our Wanderly project. By the end you will have:

- A real project blueprint document.
- A real Software Requirements Specification.
- Real wireframes and a database design.
- A working website on GitHub.
- A test plan and a bug-tracking workflow.
- A deployed live site.
- A clear understanding of how Waterfall, Agile, and Scrum differ.

Let us begin with Phase 1 - Planning.

Chapter 2 - Phase 1 - Planning and the Project Blueprint

Planning is the first SDLC phase. It happens before any code is written, before any wireframes are drawn, sometimes even before the team is hired. Its goal: answer the most important question - **should we even build this?**

What planning produces

The output of the planning phase is a **project blueprint** (also called a project charter or scope document). It is usually a 1-3 page document that contains:

- **The problem** we are solving.
- **The goal** the software is supposed to achieve.
- **The scope** - what is in, what is out.
- **Stakeholders** - who cares about this project.
- **Budget and timeline** - rough estimates.
- **Risks** - what could go wrong.
- **Success criteria** - how we will know we succeeded.

> **TIP**
>
> The blueprint is short on purpose. If you cannot fit it on three pages, you don't understand the project well enough yet. Long documents in this phase usually mean someone is hiding uncertainty behind verbosity.

Wanderly project blueprint

Here is what our Wanderly blueprint looks like. Read it carefully - it sets the direction for everything that follows.

1. The problem

Wanderly has 12 tour packages. Right now they sell them via Facebook posts and WhatsApp messages. They want a real website so they look professional, customers can browse 24/7, and bookings come in even when staff are asleep.

2. The goal

Launch a public website that displays all 12 tours and lets visitors submit booking inquiries. Have the first 10 bookings come through the new site within one month of launch.

3. Scope - what's in

- Public homepage with hero image and 3 featured tours.
- All-tours listing page with filters (region, price, duration).
- Individual tour detail page with photos and description.

- Booking form (name, email, phone, date, travellers).
- Bookings stored in a database; staff can view them.
- Mobile-friendly responsive design.
- Basic SEO (page titles, meta descriptions).

4. Scope - what's out

- Online payment processing (bookings are inquiries, paid offline).
- User accounts and login.
- Multi-language support (English only for v1).
- Real-time chat or chatbot.
- Mobile app.
- Customer reviews / ratings system.

> **WARNING**
>
> **What's out** is just as important as what's in. Without this list, the client will keep asking "can we just add..." and the project will never finish. This is called *scope creep*.

5. Stakeholders

Role	Person	What they care about
Client	Wanderly's owner	Looks professional, brings in bookings.
End user	Tourists planning trips	Can find tours, get clear info, book easily.
Developer	You (or your team)	Clear requirements, reasonable timeline.
Designer	Whoever does the visuals	Brand consistency, modern look.
QA	Whoever tests it	Working features, no broken links, consistent UX.

6. Budget and timeline

- **Total budget**: Rs. 200,000 (or equivalent freelance rate).
- **Timeline**: 4 weeks from kick-off to launch.
- **Team**: 1 developer (you), part-time designer, owner reviewing weekly.
- **Hosting**: Rs. 1,500/year for shared hosting + domain.

7. Risks

Risk	Likelihood	Mitigation
Client keeps asking for new features	High	Strict scope document; new features = phase 2 / extra cost.
Photos arrive late / poor quality	Medium	Set deadline for photo delivery in week 1.
Mobile design takes longer than expected	Medium	Reserve a buffer week; test on real phones early.
Hosting issues at launch	Low	Test the production environment a week before launch.

8. Success criteria

We will know the project succeeded if, one month after launch:

- The site is live with all 12 tours visible.
- At least 10 booking inquiries have come through the form.
- The site loads in under 3 seconds on a 4G connection.
- Google Lighthouse score of 80+ for performance and accessibility.
- Wanderly's owner reports they would hire us again.

> **TIP**
>
> Notice how all success criteria are **measurable**. "Looks professional" is not a success criterion - it is an opinion. "Lighthouse score 80+" is one. Always measure what you can.

What happens with this document?

Both the client and the developer sign off on this blueprint before any other phase begins. It becomes the reference document the rest of the project is measured against. If the client later asks for something not in scope, you point at this document. If the developer cuts a feature that was in scope, the client points at this document.

Chapter 3 - Phase 2 - Requirements (Writing the SRS)

Once the blueprint is signed, we move to **Requirements**. The blueprint says *what* we are building. Requirements say exactly *what it should do*, in detail, so a developer could build it without asking questions.

What is an SRS?

SRS stands for **Software Requirements Specification**. It is a document that lists every feature, every screen, every rule, every constraint of the software. Larger companies write formal SRS documents that can be 50+ pages. Smaller teams may use a simple list in a Google Doc. Either way, the goal is the same:

Make the requirements unambiguous, testable, and signed off.

Functional vs non-functional requirements

Requirements split into two kinds:

Type	Definition	Example
Functional	What the system DOES.	"User can submit a booking inquiry."
Non-functional	Constraints on HOW the system behaves.	"Pages must load in under 3 seconds."

Both are equally important. A site that does what it should but takes 10 seconds to load is a failure. A blazing-fast site that doesn't have a contact form is also a failure.

Wanderly functional requirements

We use a numbered format like **FR-1**, **FR-2** so we can refer to specific requirements during testing and bug reports.

- **FR-1**: The homepage shall display a hero banner with a heading, sub-heading, and two call-to-action buttons.
- **FR-2**: The homepage shall display 3 featured destinations in a grid below the hero.
- **FR-3**: Each destination card shall show name, region, starting price, and an image.
- **FR-4**: The destinations page shall list all 12 tours.
- **FR-5**: The destinations page shall provide filters by region (Cultural Triangle, Hill Country, South Coast, East Coast).
- **FR-6**: The destinations page shall provide a price range slider with min/max values.
- **FR-7**: The destinations page shall provide duration filters (1-2 days, 3-5 days, 6+ days).
- **FR-8**: Clicking a destination card shall navigate to that tour's detail page.

• **FR-9**: A tour detail page shall display name, description, photo gallery, price, duration, and a Book button.

• **FR-10**: The booking form shall require fields: full name, email, phone, trip date, number of travellers.

• **FR-11**: The booking form shall validate that email contains an @ symbol and trip date is in the future.

• **FR-12**: On successful submission, the form shall display a confirmation message and clear the fields.

• **FR-13**: All submitted bookings shall be saved to a database.

• **FR-14**: Staff shall be able to view a simple list of all bookings (admin page).

• **FR-15**: The site shall display a 404 page for any URL that does not exist.

Wanderly non-functional requirements

• **NFR-1 (Performance)**: Each page shall load in under 3 seconds on a 4G mobile connection.

• **NFR-2 (Accessibility)**: All images shall have alt text. All form fields shall have associated <label> tags.

• **NFR-3 (Responsive)**: The site shall be usable on screens from 320px to 1920px wide.

• **NFR-4 (Browser support)**: The site shall work in Chrome, Firefox, Safari, and Edge - last 2 versions of each.

• **NFR-5 (Security)**: All form submissions shall use HTTPS.

• **NFR-6 (SEO)**: Each page shall have a unique <title> tag and meta description.

• **NFR-7 (Maintainability)**: All code shall be in a public GitHub repository with at least one commit per feature.

• **NFR-8 (Localization)**: All copy shall be in English. Multi-language is out of scope for v1.

User stories - a friendlier alternative

Agile teams often skip formal SRS and write **user stories** instead. They are shorter and focus on user value:

Format: "As a <type of user>, I want to <do something>, so that <benefit>."

```
As a tourist, I want to filter tours by region,
   so that I find trips near where I plan to be.

As a tourist, I want to see prices clearly on each card,
   so that I can pick something within my budget.

As a tourist, I want to book a tour with just my name,
email, and date, so that I don't have to create an account.

As Wanderly's owner, I want to see new bookings in one place,
   so that I can call customers to confirm quickly.
```

User stories and SRS are not exclusive - many teams use both, or translate user stories into formal SRS items during sprint planning.

How requirements are tested

Every requirement should be **verifiable** - we should be able to test it and say "yes, this requirement is met" or "no, this requirement is broken". When we get to chapter 11 (testing), each test case will trace back to one or more requirements like *"This test verifies FR-11"*.

> **TIP**
>
> If you cannot write a test for a requirement, the requirement is too vague. "The site should be fast" is not testable. "The homepage should load in under 3 seconds" is.

Chapter 4 - Phase 3 - System Design and Wireframes

We have signed-off requirements. Time to figure out HOW we will build them. This is the **Design** phase. It splits into two halves: the visible design (what users will see) and the invisible design (architecture, database, APIs).

Wireframes - the structural sketch

A **wireframe** is a low-fidelity sketch of a page. No colours, no real images, no fancy fonts - just gray boxes showing where each element goes. Think of it as the floor plan of a house: you decide where the kitchen is BEFORE buying appliances.

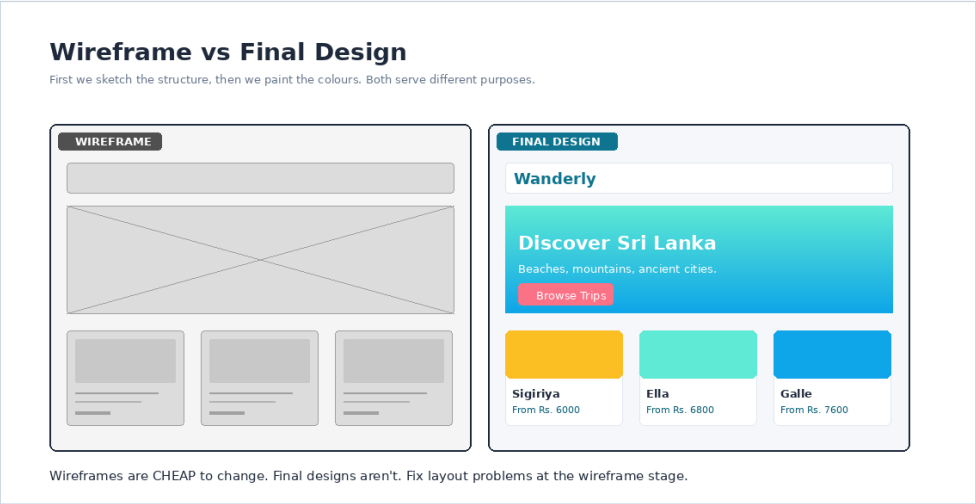


Figure 5 - Same homepage as wireframe and as final design.

Why wireframe before designing?

- **Cheap to change** - moving a gray box takes 30 seconds. Reworking a finished design in Photoshop takes hours.
- **Focuses on structure, not style** - the client cannot get distracted by colours and start arguing about the shade of teal.
- **Reveals missing pieces** - empty wireframe areas show you what content has not been written yet.
- **Aligns the team** - everyone agrees on layout before anyone starts the expensive work.

Tools for wireframing

Tool	Best for
Pen and paper	First drafts. Genuinely the fastest.
Figma (free tier)	Most teams. Browser-based, real-time collaboration.
Balsamiq	Sketchy hand-drawn look. Stops style debates.
Excalidraw	Free, open source, very fast for diagrams.
Whimsical	Wireframes plus user-flow arrows.

From wireframe to final design

Once the wireframes are signed off, the visual designer (or you) creates the **final design** - colours, typography, real images, micro-interactions. This is what the client signs off as the visual target.

Tools for high-fidelity design: **Figma** (industry standard in 2026), **Sketch** (Mac only), or **Adobe XD**.

System architecture

The other half of design is the technical architecture - the invisible structure of the system. For Wanderly we are building a simple website, so the architecture is straightforward:

- **Frontend**: HTML, CSS, vanilla JavaScript (no React for v1).
- **Backend**: PHP + MySQL on shared hosting (cheap, easy).
- **Database**: MySQL with three tables (we'll design these in chapter 5).
- **Hosting**: Shared web host with cPanel for simplicity.
- **Domain**: wanderly.lk (or .com).

> **WARNING**
>
> Architecture decisions cascade. If we picked React + Node + Postgres + AWS for this small site, hosting would cost 10x more and the timeline would double. The right architecture matches the project's actual needs - not the developer's hottest new interest.

Documenting the design

The output of the design phase is a **Design Document** containing:

- Wireframes for every page.
- High-fidelity mockups (Figma file).
- Style guide (colours, fonts, button styles).
- Information architecture (sitemap of pages and links).
- Database schema (next chapter!).
- Tech stack choices and reasons.
- Folder structure for the codebase.

When the developer starts building, this document tells them everything they need to know.

Chapter 5 - Database Design and the ER Diagram

Almost every web app needs a database. Designing it well at the start saves enormous pain later. The traditional tool is the **Entity-Relationship Diagram** (ERD).

What is an ER diagram?

An ERD is a picture of the tables in your database and how they connect to each other. Each box is a table. Each line between boxes shows a relationship.

Wanderly database design

Our blueprint and SRS told us we need to store: tours, customer bookings, and (eventually) staff users to view those bookings. Three tables.

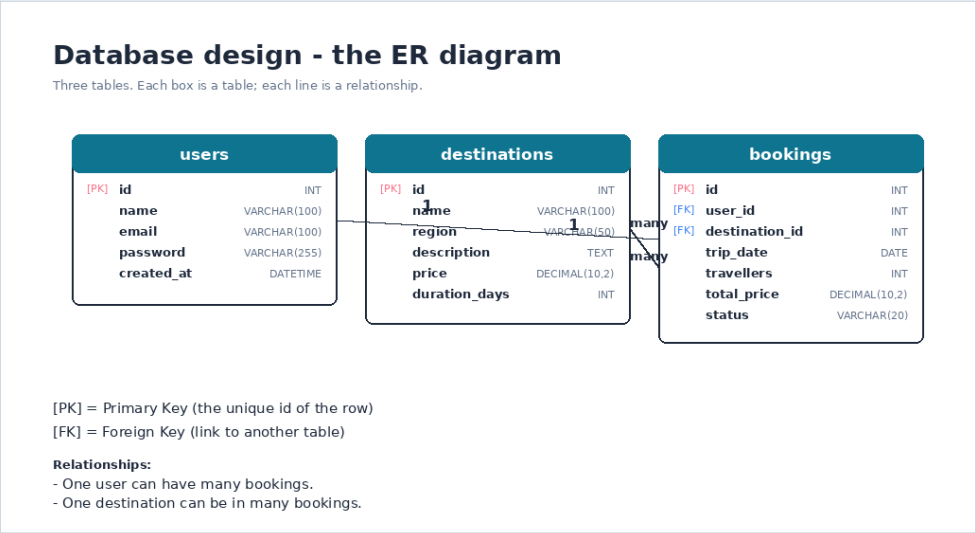


Figure 6 - Three tables: users, destinations, bookings.

Reading the ER diagram

users table

Holds staff accounts (the people who view bookings). Each row is one user with id, name, email, hashed password, and the timestamp of when they were created.

destinations table

Holds the 12 tours. Each row has id, name, region, description, price, and duration in days. We could later add columns for main_image, gallery, etc.

bookings table

Holds every booking inquiry. Each row links to a user (the staff member viewing) and a destination (which tour was booked). It also stores trip_date, number of travellers, total price calculated at booking time, and a status (pending, confirmed, cancelled).

Primary keys and foreign keys

Every table has an **id** column - the **primary key**. It uniquely identifies a row. The database fills it in automatically.

When one table needs to link to another, we use a **foreign key** column. In our diagram, *bookings.user_id* is a foreign key pointing to *users.id*. It says "this booking belongs to that user".

Same for *bookings.destination_id* pointing to *destinations.id*.

The CREATE TABLE statements

Once the ERD is approved, the developer turns it into actual SQL. Here is what those three tables look like in MySQL:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE destinations (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    region VARCHAR(50) NOT NULL,
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
    duration_days INT NOT NULL
);

CREATE TABLE bookings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    destination_id INT NOT NULL,
    trip_date DATE NOT NULL,
    travellers INT NOT NULL,
    total_price DECIMAL(10,2) NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL,
    FOREIGN KEY (destination_id) REFERENCES destinations(id)
);
```

Why ON DELETE SET NULL?

If a staff user is deleted (e.g. they leave the company), we still want to keep their old booking records - just without the link. **ON DELETE SET NULL** tells MySQL: "if the user is deleted, set this booking's user_id to NULL".

> **TIP**
>
> Database design done at this stage saves weeks of pain later. Adding a column to an empty table is trivial. Adding it to a table with 10,000 rows is a careful operation that needs downtime planning.

Normalisation - one piece of data, one place

Normalisation is the database designer's mantra. It means: every fact lives in exactly one place. If we want a destination's name, we look it up in *destinations* by id. We don't repeat the name in *bookings* - that would let it fall out of sync.

Notice that bookings stores **destination_id** (a number) and **total_price** (the price at booking time, since prices can change later). The booking does NOT store the destination's name - that comes from the destinations table when needed.

Common database design mistakes

• **Storing comma-separated lists** in one column. "jane,john,alex" in a *members* column should be a separate *memberships* table.

• **Repeating data**. If the city is the same for many users, make a *cities* table and link by id.

• **Forgetting created_at / updated_at**. You will want these timestamps for debugging and analytics.

• **VARCHAR(255) for everything**. Pick reasonable sizes. A phone number does not need 255 characters.

• **Skipping foreign keys**. Without them, the database lets you create orphan rows. Foreign keys are guardrails.

> **WARNING**
>
> Spend twice as long on database design as you think you should. Schema mistakes are the most expensive bugs to fix - they often involve data migrations, downtime, and hours of stress.

Chapter 6 - Phase 4 - Development Setup and Folder Structure

Planning, requirements, design - all done. Now we actually build it. The first task in the development phase is setting up your tools and deciding where every file will live before you write a single line of code.

Tools you need

Tool	Purpose
VS Code	Code editor.
XAMPP	Local web server (Apache + MySQL + PHP).
Git	Version control.
Browser	Test the site (Chrome with DevTools).
GitHub account	Online code repository.

Detailed install instructions for each tool are on **egotechworld.com** in the setup guides. We are skipping them here to focus on process.

The Wanderly folder structure

Before creating any files, let us decide where each one will live. A clear structure prevents mess as the project grows.

```
wanderly/
+-- public/                 # what the browser sees
|   +-- index.html          # homepage
|   +-- destinations.html   # all tours page
|   +-- booking.html        # booking form
|   +-- 404.html            # error page
|   +-- about.html          # static about page
|   +-- css/
|   |   +-- main.css        # main styles
|   |   `-- responsive.css  # mobile-specific styles
|   +-- js/
|   |   +-- main.js         # shared JS
|   |   +-- destinations.js # filter logic
|   |   `-- booking.js      # form validation
|   +-- images/             # all photos and logos
|   `-- assets/             # icons, fonts
+-- backend/                # PHP scripts
|   +-- save_booking.php    # handles form POST
|   +-- get_destinations.php # returns JSON
|   +-- db.php              # database connection
|   `-- admin/
|       +-- index.php       # admin login
|       `-- bookings.php    # view bookings
+-- database/
|   `-- schema.sql          # the CREATE TABLE statements
+-- docs/
|   +-- blueprint.md        # project blueprint
|   +-- srs.md              # requirements doc
|   `-- design/
|       `-- wireframes.fig  # Figma file
+-- tests/
|   `-- manual_test_plan.md # manual test cases
+-- .gitignore              # files Git should ignore
`-- README.md               # project overview
```

Why three top-level folders?

- **public/** - everything the browser is allowed to load. This is what gets uploaded to the web host.
- **backend/** - PHP that handles forms and admin. Should NOT be in public if possible (we'll address this later).
- **database/**, **docs/**, **tests/** - support files that live in the repo but aren't deployed to the web server.

What goes in .gitignore

Some files should never be committed to Git. Create a **.gitignore** file in the root with:

```
.env                # database passwords
node_modules/       # if we ever add Node tooling
.DS_Store           # macOS clutter
Thumbs.db           # Windows clutter
.vscode/            # editor settings
*.log               # log files
backend/uploads/    # user-uploaded files
.idea/              # IntelliJ / WebStorm settings
```

> **WARNING**
>
> **NEVER commit passwords or API keys.** If you accidentally do, even one commit, assume they are compromised and rotate them immediately. Git history is forever.

The README.md

Every project has a **README** at the root. It is the first thing visitors see on GitHub. A good README answers:

- What is this project?
- How do I run it locally?
- What does the folder structure mean?
- Who do I contact if something is broken?

Wanderly's starter README:

```
# Wanderly

Travel website for Wanderly tours - 12 Sri Lankan trip packages.

## Tech stack
- HTML, CSS, JavaScript (vanilla)
- PHP 8 + MySQL on the backend
- Hosted on shared web hosting

## Local setup

1. Install XAMPP and start Apache + MySQL.
2. Clone this repo into `htdocs/wanderly`.
3. Import `database/schema.sql` via phpMyAdmin.
4. Copy `backend/db.example.php` to `backend/db.php` and fill in
   your local DB credentials.
5. Visit http://localhost/wanderly/public/

## Project structure

See `docs/folder_structure.md`.

## Status

In active development. See `docs/blueprint.md` for the project plan.

## Author

Built by [Your Name] for Wanderly tours, 2026.
```

First commit

With the folder structure in place (even if most files are empty stubs), we are ready to make our first Git commit. That's chapter 7.

Chapter 7 - Git Basics - Tracking Every Change

Git is a version control system. It tracks every change you make to your code. You can see what changed, when, why, and by whom. You can roll back to yesterday's version if today's is broken. You can experiment safely. You can collaborate without overwriting each other's work.

Every professional project uses Git. Learning it is one of the best investments any developer can make.

How Git thinks - three places

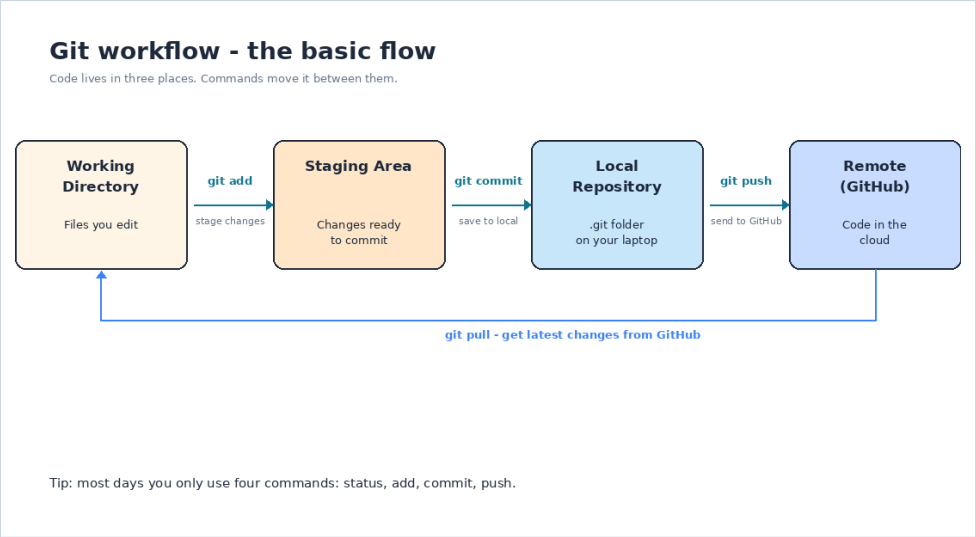


Figure 7 - The Git workflow. Code moves left to right with specific commands.

Code lives in three (or four) places:

- **Working directory** - the actual files on your laptop.
- **Staging area** - changes you have marked as ready for the next commit.
- **Local repository** - the .git folder on your laptop, holding every committed snapshot.
- **Remote (GitHub)** - the public copy of your repository in the cloud.

The most-used Git commands

Setup (one time)

```
git config --global user.name "Your Name"
git config --global user.email "[email protected]"
```

Starting a new project

```
cd ~/code/wanderly
git init                        # create a new local repo
git add .                       # stage all files
git commit -m "Initial commit - folder structure"
```

The everyday loop

```
git status                      # what changed?
git diff                        # show me the changes line by line

git add public/index.html       # stage one file
git add .                       # stage everything

git commit -m "Add hero section to homepage"
git push                        # send to GitHub
```

Useful inspection commands

```
git log                         # commit history
git log --oneline               # condensed history
git log --oneline -5            # just the last 5 commits

git show abc1234                # full details of one commit

git blame public/index.html     # who wrote each line
```

Writing good commit messages

A commit message is a note to your future self (and your team) explaining what changed and why. Bad messages waste everyone's time when they're trying to track down a bug six months later.

Bad message	Better message
fix	Fix mobile layout overflow on destinations page
update css	Increase header padding for better mobile spacing
wip	Add booking form HTML structure (form not wired yet)
asdf	Add 404 page and error message styling
fixed bug	Fix booking form failing when phone number contains spaces

> **TIP**
>
> Rule of thumb: a good commit message completes the sentence "This commit will..." - e.g. "This commit will fix mobile layout overflow". Use the imperative mood, present tense.

How often should I commit?

Commit early, commit often. A useful guideline: **commit every time you have something working that is better than what came before**. Don't commit broken code, but don't wait until the whole feature is perfect either.

- Added the navbar HTML? Commit.
- Got it styled? Commit.
- Made it responsive? Commit.
- Fixed a typo? Commit (small commit, but still a commit).

Undoing things

Situation	Command
Discard unsaved changes to one file	git checkout -- file.html
Unstage a file	git reset HEAD file.html
Edit the last commit's message	git commit --amend
Undo the last commit but keep the changes	git reset HEAD~1
Get rid of the last commit completely (DANGEROUS)	git reset --hard HEAD~1

WARNING

git reset --hard destroys your work. There is no undo for an undo. Always check *git status* first and make sure you really mean it.

Chapter 8 - GitHub - Branches, Pull Requests, Code Reviews

GitHub is a website that hosts Git repositories online. It is where teams collaborate. We will push our local repo to GitHub and learn the workflow that almost every dev team in the world uses.

Step 1 - Create a GitHub repo

1. Sign in at **github.com**.
2. Click the **+** in the top right -> **New repository**.
3. Repo name: **wanderly**.
4. Visibility: Public (or Private if you prefer).
5. Do NOT initialise with README - we already have one locally.
6. Click Create.

GitHub shows you the exact commands to push your local repo. They look like:

```
git remote add origin https://github.com/yourname/wanderly.git
git branch -M main
git push -u origin main
```

Run those in your terminal. Refresh GitHub - your code is online.

> **NOTE**
>
> GitHub no longer accepts password login. You need a **Personal Access Token**: Settings -> Developer settings -> Personal access tokens -> Generate. Use it where Git asks for a password.

Why branches?

On the **main** branch we keep working code. When we want to build a new feature, we create a new branch, work on it without disturbing main, and only merge back when it's tested and reviewed. This is called **feature branching**.

The feature branch flow

```
# Make sure main is up to date
git checkout main
git pull

# Create a new branch and switch to it
git checkout -b feature/destinations-filters

# ... do your work, commit as usual ...
git add .
git commit -m "Add region filter to destinations page"

# Push the branch to GitHub
git push -u origin feature/destinations-filters
```

Branch naming conventions

Prefix	Used for
feature/...	Building new features
fix/...	Fixing bugs
hotfix/...	Urgent fixes that go to production immediately
refactor/...	Cleaning up code without changing behaviour
docs/...	Documentation-only changes

Pull Requests

When your feature branch is ready, you open a **Pull Request** (PR) on GitHub. A PR is a request to merge your branch into main. It opens a discussion thread where teammates can review your code, leave comments, request changes, and finally approve.

Anatomy of a good PR

- **Title**: short, descriptive. "Add region filter to destinations page"
- **Description**: what changed, why, screenshots if visual.
- **Linked issues**: "Closes #42" if it fixes a tracked issue.
- **Reviewers**: at least one teammate (sometimes two for important changes).
- **Tests**: did you test it? Mention browsers / scenarios.

Code review etiquette

When reviewing a PR, you are the second pair of eyes. Your job is to make the code better, not to make the author feel bad. Some rules:

- **Be specific**. "This loop is unclear" is unhelpful. "This loop runs O(n^2); could we use a Map for O(n)?" is helpful.
- **Suggest, don't demand**. "What about ..." rather than "You should ...".
- **Explain why**. "Move this to a constant - we use the same magic number on line 42."
- **Praise good code**. "Nice use of optional chaining here." Code review is also encouragement.
- **Ask, don't assume**. "Is this intentional?" - sometimes the author had a reason you didn't see.

After approval - merging

Once approved, you click **Merge pull request** on GitHub. Your branch's commits are now part of *main*. The branch can usually be deleted.

```
# After merging on GitHub, update your local main:
git checkout main
git pull

# Delete the now-merged branch locally
git branch -d feature/destinations-filters

# Start the next feature branch
git checkout -b feature/booking-form
```

> **TIP**
>
> Even when you work alone, this workflow is worth following. Branches give you a safety net - if a feature does not work out, you delete the branch and main is untouched. PRs give you a place to reflect on your own changes before merging them.

Chapter 9 - Building the Travel Site - HTML and CSS

Now the actual coding starts. We will build the HTML structure and CSS styling for the homepage first, in small commits.

Step 1 - The HTML skeleton

Open **public/index.html** in VS Code. Type *!* and hit Tab - VS Code's Emmet shortcut creates the HTML boilerplate for you. Then fill in the basics:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <title>Wanderly - Discover Sri Lanka</title>
    <meta name="description"
          content="Curated tours of Sri Lanka - cultural triangle,
                   hill country, beaches.">
    <link rel="stylesheet" href="css/main.css">
</head>
<body>
    <header class="site-header">
        <a href="/" class="logo">Wanderly</a>
        <nav>
            <a href="index.html">Home</a>
            <a href="destinations.html">Destinations</a>
            <a href="booking.html">Book</a>
            <a href="about.html">About</a>
        </nav>
    </header>

    <main>
        <!-- hero, featured tours, footer go here -->
    </main>

    <footer class="site-footer">
        <p>&copy; 2026 Wanderly Tours. All rights reserved.</p>
    </footer>

    <script src="js/main.js"></script>
</body>
</html>
```

> **TIP**
>
> Notice the <meta name="description"> - this is the text Google shows below your title in search results. Always write a good description (NFR-6).

Save. Commit:

```
git add public/index.html
git commit -m "Add HTML skeleton for homepage"
```

Step 2 - The hero section

Add this inside <*main*>:

```
<section class="hero">
    <div class="hero-content">
        <h1>Discover Sri Lanka</h1>
        <p>Beaches, mountains, ancient cities, tea country.</p>
        <p>Pick your perfect trip in three minutes.</p>
        <div class="hero-buttons">
            <a href="destinations.html" class="btn btn-primary">
                Browse Trips
            </a>
            <a href="booking.html" class="btn btn-outline">
                Sign Up
            </a>
        </div>
    </div>
</section>
```

Step 3 - Style the hero

Open **public/css/main.css**:

```
/* Reset and basics */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
}

body {
    font-family: system-ui, sans-serif;
    color: #1f2937;
    line-height: 1.6;
}

/* Header */
.site-header {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 1rem 2rem;
    background: white;
    border-bottom: 1px solid #e5e7eb;
}

.site-header .logo {
    font-size: 1.5rem;
    font-weight: bold;
    color: #0E7490;
    text-decoration: none;
}

.site-header nav a {
    color: #1f2937;
    text-decoration: none;
    margin-left: 1.5rem;
}

/* Hero */
.hero {
    background: linear-gradient(135deg, #5EEAD4 0%, #0EA5E9 100%);
    color: white;
    padding: 5rem 2rem;
    min-height: 380px;
    display: flex;
    align-items: center;
}

.hero h1 {
    font-size: 3rem;
    margin-bottom: 1rem;
}

.hero p {
    font-size: 1.1rem;
    margin-bottom: 0.5rem;
}

.hero-buttons {
    margin-top: 2rem;
}

.btn {
```

```
    display: inline-block;
    padding: 0.75rem 1.5rem;
    border-radius: 8px;
    text-decoration: none;
    font-weight: bold;
    margin-right: 1rem;
}

.btn-primary {
    background: #FB7185;
    color: white;
}

.btn-outline {
    border: 2px solid white;
    color: white;
}
```

Save and refresh the browser. Big teal-to-blue gradient hero with a heading, two CTAs, all looking professional. **This is the moment design becomes reality.**

Step 4 - Featured destinations grid

After the hero in HTML:

```
<section class="featured">
    <h2>Featured Destinations</h2>
    <div class="card-grid">
        <article class="card">
            <div class="card-image"
                 style="background:#FBBF24;"></div>
            <div class="card-body">
                <h3>Sigiriya</h3>
                <p>Ancient rock fortress.</p>
                <p class="price">From Rs. 8,500</p>
            </div>
        </article>

        <article class="card">
            <div class="card-image"
                 style="background:#5EEAD4;"></div>
            <div class="card-body">
                <h3>Ella</h3>
                <p>Tea country & nine arch.</p>
                <p class="price">From Rs. 6,200</p>
            </div>
        </article>

        <article class="card">
            <div class="card-image"
                 style="background:#0EA5E9;"></div>
            <div class="card-body">
                <h3>Galle Fort</h3>
                <p>Coastal Dutch heritage.</p>
                <p class="price">From Rs. 5,800</p>
            </div>
        </article>
    </div>
</section>
```

And the matching CSS:

```
.featured {
```

```
    padding: 3rem 2rem;
}

.featured h2 {
    margin-bottom: 1.5rem;
}

.card-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(260px, 1fr));
    gap: 1.5rem;
}

.card {
    background: white;
    border: 1px solid #e5e7eb;
    border-radius: 10px;
    overflow: hidden;
}

.card-image {
    height: 140px;
}

.card-body {
    padding: 1rem;
}

.card-body h3 {
    margin-bottom: 0.5rem;
}

.card-body p {
    color: #6b7280;
    margin-bottom: 0.5rem;
}

.price {
    color: #0E7490;
    font-weight: bold;
}
```

Step 5 - Make it responsive

On a phone, the hero text needs to be smaller. Add to **main.css** (or a separate *responsive.css*):

```
@media (max-width: 600px) {
    .hero {
        padding: 3rem 1.5rem;
    }
    .hero h1 {
        font-size: 2rem;
    }
    .site-header {
        flex-direction: column;
        align-items: flex-start;
    }
    .site-header nav {
        margin-top: 1rem;
    }
    .site-header nav a {
        margin-left: 0;
        margin-right: 1rem;
    }
}
```

> **TIP**
>
> Test on a real phone, not just Chrome's responsive mode. Things like touch-tap target size, font rendering, and scrolling behaviour are different on real devices. NFR-3 demands 320px minimum.

Commit your work

```
git add .
git commit -m "Add hero section, featured destinations, responsive styles"
git push
```

Chapter 10 - Building the Travel Site - JavaScript and Forms

HTML and CSS gave us a beautiful but static site. JavaScript makes it interactive. We will add: filter logic on the destinations page, form validation on the booking page.

Step 1 - Create destinations.html

Copy the basic HTML skeleton from chapter 9. Inside <main>:

```
<section class="destinations-page">
    <h1>All Destinations</h1>
    <p class="subtitle">Browse all 12 trips.</p>

    <div class="page-grid">
        <!-- Sidebar with filters -->
        <aside class="filters">
            <h3>Filters</h3>

            <fieldset>
                <legend>Region</legend>
                <label><input type="checkbox" value="Cultural"
                       data-filter="region"> Cultural Triangle</label>
                <label><input type="checkbox" value="Hill"
                       data-filter="region"> Hill Country</label>
                <label><input type="checkbox" value="South"
                       data-filter="region"> South Coast</label>
                <label><input type="checkbox" value="East"
                       data-filter="region"> East Coast</label>
            </fieldset>
        </aside>

        <!-- Destinations grid -->
        <div class="results" id="results">
            <!-- Cards generated by JS -->
        </div>
    </div>
</section>

<script src="js/destinations.js"></script>
```

Step 2 - The destinations data

For now we'll hard-code the destinations as a JavaScript array. Later (when we add PHP) we'll fetch them from the database instead. Open **public/js/destinations.js**:

```
// public/js/destinations.js
const destinations = [
    { id: 1, name: 'Sigiriya & Dambulla',
      region: 'Cultural', price: 8500, days: 2 },
    { id: 2, name: 'Ella & Nine Arch',
      region: 'Hill',     price: 6200, days: 3 },
    { id: 3, name: 'Galle Fort',
      region: 'South',    price: 5800, days: 1 },
    { id: 4, name: 'Yala Safari',
      region: 'South',    price: 9200, days: 2 },
    { id: 5, name: 'Anuradhapura',
      region: 'Cultural', price: 5500, days: 1 },
    { id: 6, name: 'Nuwara Eliya',
      region: 'Hill',     price: 7400, days: 2 },
    // ... add 6 more
];

function renderResults(items) {
    const container = document.getElementById('results');
    if (items.length === 0) {
        container.innerHTML =
            '<p class="empty">No tours match your filters.</p>';
        return;
    }
    container.innerHTML = items.map(d => `
        <article class="card">
            <div class="card-image"></div>
            <div class="card-body">
                <h3>${d.name}</h3>
                <p>${d.region} - ${d.days} day(s)</p>
                <p class="price">From Rs. ${d.price.toLocaleString()}</p>
                <a href="booking.html?id=${d.id}"
                   class="btn btn-primary">View</a>
            </div>
        </article>
    `).join('');
}

function applyFilters() {
    const checked = document.querySelectorAll(
        'input[data-filter="region"]:checked'
    );
    const regions = Array.from(checked).map(c => c.value);

    if (regions.length === 0) {
        renderResults(destinations);
    } else {
        renderResults(
            destinations.filter(d => regions.includes(d.region))
        );
    }
}

// Wire up checkboxes
document.querySelectorAll('input[data-filter]').forEach(input => {
    input.addEventListener('change', applyFilters);
});

// Initial render
```

```
renderResults(destinations);
```

What this code does

- **destinations** array - the data, hard-coded for now.
- **renderResults(items)** - generates HTML for each item and puts it in the page. Uses *.map(...).join('')* - the standard pattern for templating in vanilla JS.
- **applyFilters()** - reads which checkboxes are checked, filters the data, calls renderResults with the filtered list.
- **addEventListener('change', applyFilters)** - re-runs the filter every time a checkbox is toggled.

Step 3 - The booking form

Create **public/booking.html**. Inside <main>:

```
<section class="booking-page">
    <h1>Book your trip</h1>

    <form id="booking-form" class="booking-form"
          action="../backend/save_booking.php" method="POST">
        <label>
            Full name
            <input type="text" name="name" required>
        </label>

        <label>
            Email
            <input type="email" name="email" required>
        </label>

        <label>
            Phone
            <input type="tel" name="phone"
                   pattern="[0-9+\\s-]+" required>
        </label>

        <label>
            Trip date
            <input type="date" name="trip_date" required>
        </label>

        <label>
            Number of travellers
            <input type="number" name="travellers"
                   min="1" max="10" required>
        </label>

        <p id="form-error" class="error"></p>
        <button type="submit" class="btn btn-primary">
            Send Booking
        </button>
    </form>
</section>

<script src="js/booking.js"></script>
```

Step 4 - Form validation

public/js/booking.js:

```
// public/js/booking.js
const form = document.getElementById('booking-form');
const errorEl = document.getElementById('form-error');

form.addEventListener('submit', (e) => {
    errorEl.textContent = '';

    // Trip date must be in the future (FR-11)
    const tripDate = new Date(form.trip_date.value);
    const today = new Date();
    today.setHours(0, 0, 0, 0);

    if (tripDate <= today) {
        e.preventDefault();
        errorEl.textContent = 'Trip date must be in the future.';
        return;
    }

    // Travellers must be a sensible number
    const travellers = Number(form.travellers.value);
    if (travellers < 1 || travellers > 10) {
        e.preventDefault();
        errorEl.textContent =
            'Travellers must be between 1 and 10.';
        return;
    }

    // If we get here, the form is valid - let it submit normally
});
```

Why client-side AND server-side validation?

Client-side validation (this JavaScript) is for **user experience** - it gives immediate feedback. But a malicious user can disable JavaScript and submit anyway. So the server (our PHP script) MUST also validate. Trust nothing from the browser.

> **WARNING**
>
> There's a software security mantra: **never trust user input**. Validate on the client for friendliness, validate on the server for safety. Both, always.

Step 5 - The PHP backend (just enough)

backend/save_booking.php:

```
<?php
require 'db.php';

// Validate again on the server
$name        = trim($_POST['name'] ?? '');
$email       = trim($_POST['email'] ?? '');
$phone       = trim($_POST['phone'] ?? '');
$trip_date   = $_POST['trip_date'] ?? '';
$travellers  = (int)($_POST['travellers'] ?? 0);

$errors = [];
if ($name === '') $errors[] = 'Name is required';
if (!filter_var($email, FILTER_VALIDATE_EMAIL))
    $errors[] = 'Invalid email';
if ($trip_date <= date('Y-m-d'))
    $errors[] = 'Trip date must be in the future';
if ($travellers < 1 || $travellers > 10)
    $errors[] = 'Travellers must be 1-10';

if ($errors) {
    http_response_code(400);
    echo implode(', ', $errors);
    exit;
}

// All good - save it
$stmt = $pdo->prepare(
    'INSERT INTO bookings (destination_id, trip_date, travellers,
                           total_price, status)
     VALUES (?, ?, ?, ?, ?)'
);
$stmt->execute([1, $trip_date, $travellers, 0, 'pending']);

// Redirect to a thank-you page
header('Location: ../public/thanks.html');
exit;
?>
```

We are not implementing the full backend in this tutorial - this is the SDLC tutorial, not a PHP one - but the principle is shown: server validates, server stores, server redirects.

> **WARNING**
>
> Notice **$pdo->prepare(...)** with **?** placeholders. This is a parameterised query - it stops SQL injection attacks. NEVER concatenate user input into SQL. See our PHP tutorial on egotechworld.com for the full story.

Commit and push

```
git add .
git commit -m "Add destinations page with region filter and booking form"
git push
```

Chapter 11 - Phase 5 - Testing Fundamentals

Code that works on your laptop is not finished. It needs **testing** - the deliberate process of trying to break it before users do. Testing is what separates hobby projects from professional software.

Three levels of testing

Level	What it tests	Example
Unit	One function or component in isolation.	validatePhone('07712345') returns true.
Integration	Multiple pieces working together.	Submitting the form saves a row in the database.
End-to-end (E2E)	The whole system from a user's perspective.	Visit homepage, click Browse, filter, submit booking, see thanks page.

Most teams use a mix: many unit tests (fast, cheap), some integration tests (medium speed, more setup), few E2E tests (slow, expensive, but they catch the bugs that matter).

Manual vs automated testing

- **Manual testing** - a human follows a list of steps and checks each result. Slower but flexible. Good for visual things and exploratory testing.
- **Automated testing** - a script runs the test and reports pass/fail. Fast and repeatable. Good for repetitive checks like "my form still validates correctly after this code change".

For a small project like Wanderly, a thorough **manual test plan** is enough. Larger teams add automated tests using tools like Jest, Cypress, or Playwright.

Writing a manual test plan

A test plan is a list of **test cases**. Each test case has:

- **ID** (TC-1, TC-2, ...)
- **Title** - what is being tested
- **Pre-condition** - what state must be set up first
- **Steps** - what to do
- **Expected result** - what should happen
- **Pass / Fail** - tick after running
- **Linked requirement** - which FR this verifies

Wanderly test cases (sample)

```
TC-1: Homepage loads
Pre-condition: None
Steps: Navigate to /
Expected: Hero section visible, 3 featured cards visible
Verifies: FR-1, FR-2, FR-3
Status: [ ] Pass [ ] Fail


TC-2: Region filter shows only matching tours
Pre-condition: 12 tours seeded in database
Steps:
  1. Visit /destinations.html
  2. Tick the "Hill Country" checkbox
Expected: Only tours where region = "Hill" are visible
Verifies: FR-5
Status: [ ] Pass [ ] Fail


TC-3: Booking form rejects past trip date
Pre-condition: None
Steps:
  1. Visit /booking.html
  2. Fill in name, email, phone, travellers
  3. Set trip date to yesterday
  4. Click Send Booking
Expected: Form does NOT submit. Error message:
          "Trip date must be in the future."
Verifies: FR-11
Status: [ ] Pass [ ] Fail


TC-4: Booking saves to database
Pre-condition: Database is empty
Steps:
  1. Visit /booking.html
  2. Fill in valid data, submit
  3. Check phpMyAdmin -> bookings table
Expected: One new row exists with the submitted data
Verifies: FR-13
Status: [ ] Pass [ ] Fail


TC-5: Site is mobile-friendly
Pre-condition: None
Steps:
  1. Open Chrome DevTools (F12)
  2. Toggle device toolbar
  3. Switch between iPhone SE and iPad
  4. Visit homepage, destinations, booking
Expected: All pages readable, no horizontal scroll, buttons tappable
Verifies: NFR-3
Status: [ ] Pass [ ] Fail
```

> **TIP**
>
> A test that doesn't link back to a requirement is suspicious. Why are you testing it? Conversely, a requirement without a test is also suspicious - how will you know it works?

Browser compatibility testing

NFR-4 says we support last 2 versions of Chrome, Firefox, Safari, Edge. So our test plan needs to repeat critical tests in each browser:

- Chrome on Windows / Mac / Linux
- Firefox on Windows / Mac
- Safari on Mac (and iPhone)
- Edge on Windows
- Chrome on Android phone
- Safari on iPhone

BrowserStack and **Sauce Labs** are paid services that let you test in many browsers without owning every device. For small projects, free options like **BrowserStack's free tier** or just borrowing your friend's phone work fine.

Performance testing

NFR-1 demands pages load in under 3 seconds. We test this with:

- **Chrome Lighthouse** (built into DevTools - the Lighthouse tab). Gives scores for Performance, Accessibility, Best Practices, SEO. Aim for 80+.
- **WebPageTest.org** - tests from real locations and slow networks.
- **GTmetrix** - similar idea, with different metrics.

Common performance fixes: compress images (WebP format), minify CSS/JS, lazy-load images below the fold, avoid render-blocking scripts in <head>.

Chapter 12 - Quality Assurance and the Bug Lifecycle

Quality Assurance (QA) is broader than just testing. Testing is one activity within QA. QA also covers process, documentation, code reviews, and the system that catches and fixes problems before they reach users.

Testing vs QA - the difference

Testing	Quality Assurance
Reactive - finds bugs that exist.	Proactive - prevents bugs from existing.
Done by testers / developers.	Done by everyone, all the time.
A specific phase or activity.	A continuous mindset.
Measures the product.	Measures the process.

QA activities throughout the SDLC

- **During Planning**: review the blueprint - is the scope realistic? Are success criteria measurable?
- **During Requirements**: review the SRS - is each requirement clear and testable?
- **During Design**: review wireframes - do they cover all requirements? Any usability issues?
- **During Development**: code reviews on every PR. Style linters. Pair programming.
- **During Testing**: execute test cases. Log bugs. Re-test after fixes.
- **During Deployment**: smoke tests in production. Monitor logs.
- **During Maintenance**: track recurring issues. Update tests.

How a bug travels through QA

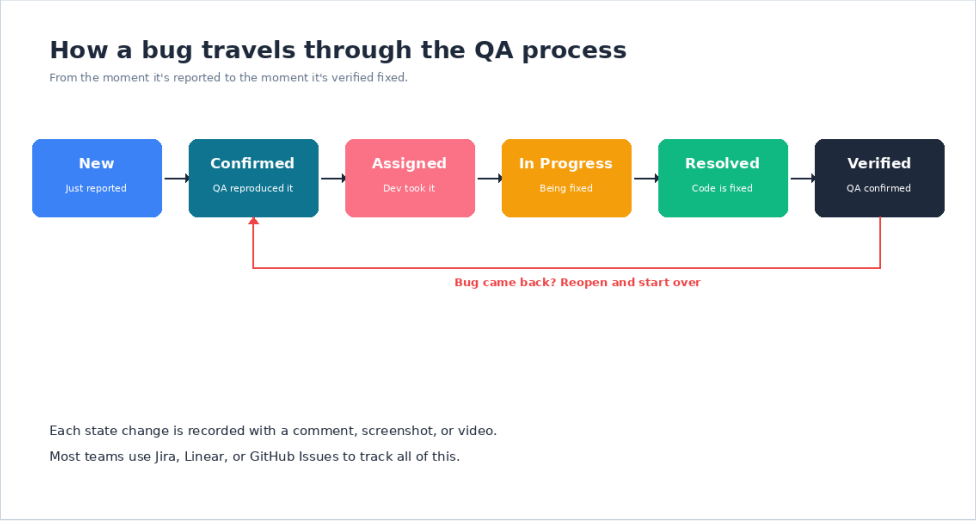


Figure 8 - Six states a bug passes through. Note the loopback - if QA can't verify the fix, the bug reopens.

Each state explained

State	What just happened
New	A bug was reported. Hasn't been verified yet.
Confirmed	QA reproduced it. We know it's real.
Assigned	A specific developer is responsible for fixing it.
In Progress	Developer is actively working on the fix.
Resolved	Developer says the fix is done. Code merged to main.
Verified	QA tested the fix. Bug is gone for good.
Reopened	QA tried to verify but the bug is still there.

Anatomy of a good bug report

When you report a bug, give the developer enough info to fix it without coming back to ask questions. A good report has:

- **Title**: short, specific. "Booking form accepts past dates on Safari" - not "Form is broken".
- **Steps to reproduce**: numbered list of exact actions.
- **Expected result**: what should happen.
- **Actual result**: what does happen.
- **Environment**: browser, OS, screen size, logged in or not.
- **Severity**: how serious is it (see below).
- **Screenshot or video**: a picture is worth 1,000 lines of text.
- **Console errors**: paste any red errors from DevTools.

Bug severity levels

Severity	Meaning	Example
Critical / Blocker	App is unusable. Drop everything.	Site won't load. Can't submit any booking.
High / Major	Important feature broken.	Booking form accepts past dates.
Medium	Annoying but workaround exists.	Filter checkboxes don't update count text.
Low / Minor	Cosmetic.	Spacing slightly off on tablet.

Sample bug report

```
Title: Booking form submits successfully with empty phone field

Reproduction:
  1. Open https://wanderly.lk/booking.html
  2. Fill in: name = "Test", email = "[email protected]"
  3. Leave phone field empty
  4. Pick a future trip date and travellers = 1
  5. Click Send Booking

Expected: Form should NOT submit; error "Phone is required"
Actual:   Form submits and creates a booking with empty phone
Severity: High (FR-10 says phone is required)
Browser:  Chrome 130 on Windows 11
Verifies: FR-10
Screenshot: bug-1024-empty-phone.png attached
```

Tools for tracking bugs

- **GitHub Issues** - free, integrated with your repo. Good for solo and small-team projects.
- **Jira** - industry standard for medium-large companies. Powerful but heavy.
- **Linear** - modern alternative to Jira. Faster and prettier.
- **Trello** / **Notion** - lightweight kanban for tiny teams.
- **YouTrack** - JetBrains' alternative, generous free tier.

When can I close the project?

There is no such thing as zero bugs. The question is: which bugs are acceptable to ship with? A typical exit criteria for launch:

- Zero Critical or High bugs open.
- All Medium bugs have either been fixed or have a documented workaround.
- Low bugs are in the backlog for the next iteration.
- All FR-* requirements have at least one passing test case.
- Lighthouse score 80+ on key pages.
- The client has reviewed and signed off.

> **TIP**
>
> If you wait for zero bugs, you will never ship. The art of QA is knowing what's good enough TODAY, while keeping the longer list visible for tomorrow.

Chapter 13 - Phase 6 - Deployment and Going Live

Time to put the site online. **Deployment** is the act of moving code from your laptop to a server where users can reach it. For Wanderly, we are going with shared hosting (cheapest, simplest).

Choosing a host

Type	Cost / mo	Notes
Shared hosting	Rs. 200-500	Hostinger, BlueHost. Cheap. PHP + MySQL ready.
VPS	Rs. 1,000-3,000	DigitalOcean, Linode. You manage the server.
Cloud (PaaS)	Rs. 0-thousands	Vercel, Netlify (frontend), Railway, Render.
Static-only	Free	GitHub Pages, Netlify free tier. No PHP/DB.

For Wanderly with PHP + MySQL, we go with shared hosting like **Hostinger**. Buy a plan + domain (around Rs. 1,500/year), follow their getting-started wizard.

Step 1 - Get the production database ready

1. Log into your hosting cPanel.
2. Open MySQL Databases.
3. Create a database called **wanderly_prod**.
4. Create a user with a strong password; grant it ALL on the new database.
5. Open phpMyAdmin from cPanel and import **database/schema.sql**.
6. Add a few real destinations via Insert (or import them too).

Step 2 - Update production config

On your laptop, in **backend/db.php**, the credentials are for local XAMPP. Production needs different ones. Solution: use a **.env** file or two separate db files.

Cleanest approach - have **db.example.php** in Git but never **db.php**:

```
// backend/db.example.php (committed to Git)
<?php
$DB_HOST = 'localhost';
$DB_NAME = 'CHANGE_ME';
$DB_USER = 'CHANGE_ME';
$DB_PASS = 'CHANGE_ME';

// backend/db.php (in .gitignore - real local credentials)
<?php
$DB_HOST = 'localhost';
$DB_NAME = 'wanderly_dev';
$DB_USER = 'root';
$DB_PASS = '';
```

On the production server, you create db.php there with the production credentials. Each environment has its own credentials, and Git never sees real ones.

Step 3 - Upload the files

1. Connect via FTP (FileZilla) or use cPanel's File Manager.
2. Upload everything in your **public/** folder to the server's **public_html/**.
3. Upload your **backend/** folder ABOVE public_html (so it's not directly browseable).
4. Create the production **db.php** on the server.
5. Visit your domain in a browser - the homepage should load.

> **TIP**
>
> A simpler alternative: upload everything via Git. Most modern shared hosts let you connect a GitHub repo and auto-deploy when you push. Saves manual FTP and is much less error-prone.

Step 4 - Smoke test

A **smoke test** is a quick sanity check that nothing is obviously broken. Run through these in production:

- Homepage loads.
- Click Destinations - tours appear.
- Click a tour card - detail page loads.
- Open booking form - all fields visible.
- Submit a test booking - check phpMyAdmin to confirm it saved.
- Visit a non-existent URL - 404 page appears (FR-15).
- Check on a phone - layout works.

Step 5 - Production hardening

A few extra steps before announcing the launch:

- **Enable HTTPS** - free Let's Encrypt cert via cPanel. Required by NFR-5.
- **Disable error display** - in production we never show PHP errors to users (security risk). Set *display_errors = 0* in php.ini.
- **Strong DB password** - not the same as dev.
- **Backups** - set up automated daily DB backups in cPanel.
- **Set up monitoring** - free services like UptimeRobot can notify you when the site goes down.

Step 6 - Announce the launch

The website is live. Time for the social media post, the email to the client's mailing list, the WhatsApp announcement to friends. The development phase ends; the maintenance phase begins.

> **TIP**
>
> Launch day rule: **do nothing else risky**. No new features, no big migrations, no schema changes. Watch the logs, fix small bugs as users find them, celebrate the small win.

Chapter 14 - Maintenance and the Iteration Loop

Software is never "finished". It is just **shipped**. After launch, the maintenance phase begins - and it lasts as long as the software is in use. For most projects, this is where 80 percent of the total time and cost goes.

Four types of maintenance

Type	What it means	Example
Corrective	Fixing bugs reported by users.	"Booking form lets people pick yesterday" reported - we fix it.
Adaptive	Adjusting to changes outside our control.	PHP 9 ships - we update our code to be compatible.
Perfective	Improving things even though they aren't broken.	Refactoring slow queries, adding caching, prettier UI.
Preventive	Reducing future risk.	Adding tests, updating dependencies, security patches.

The post-launch backlog

Within a week of launch, you will have a list of:

- Bugs users found.
- Features the client "forgot" to mention earlier.
- Things that worked in testing but reveal problems in real use.
- Performance issues only visible at scale.
- Browser-specific quirks not caught in QA.

Triage these by severity and value, prioritise, and start the next iteration. **The SDLC loops back to planning.**

Phase 2 of Wanderly - what's next?

After watching real users for a month, we might decide phase 2 should add:

- Online payments (so people can confirm bookings instantly).
- Customer reviews and star ratings (boosts trust).
- Sinhala and Tamil translations (NFR-8 said "out of scope" - but maybe phase 2 brings it in).
- Email confirmations (currently the staff calls everyone manually).
- An admin panel to add new tours without editing the database.

Each of these would be a mini-SDLC: blueprint, requirements, design, build, test, deploy. The cycle never ends.

Monitoring and observability

You can't fix problems you don't see. Set up:

- **Uptime monitoring** - UptimeRobot, Pingdom. Tells you when the site is down.
- **Error tracking** - Sentry, Rollbar. Catches JavaScript errors users hit and emails you.
- **Analytics** - Google Analytics, Plausible. Shows you what pages people visit, where they drop off.
- **Server logs** - check Apache and PHP error logs weekly. Patterns of errors point to bugs you don't even know about.

Updating dependencies

Every library you use will eventually have security vulnerabilities. Set a calendar reminder to update dependencies monthly. For PHP this means watching for security advisories. For Node projects, **npm audit** shows known vulnerabilities.

> **WARNING**
>
> Out-of-date software is hacked software. Most major data breaches happen on systems running outdated dependencies. Set the calendar reminder. Future you will be grateful.

When to retire software

The last sub-phase of the SDLC - rarely talked about - is **retirement**. Eventually every system is replaced by something better. Plan for this:

- Export user data in standard formats so it can be migrated.
- Keep the source code archived even after the system is offline.
- Notify users with enough lead time.
- Redirect old URLs to new equivalents if there is a successor.

Chapter 15 - Methodologies - Waterfall, Agile, Scrum

We have been walking through SDLC phases as if they happen in order: Planning -> Requirements -> Design -> Development -> Testing -> Deployment. That is one approach, called **Waterfall**. There are others. The methodology you pick determines how the phases overlap, repeat, and interact.

Waterfall

Waterfall - phases happen in strict order. Each phase must finish before the next begins. Output of each phase is signed off as a formal document.

Pros

- Clear milestones and deadlines.
- Easy to budget - you know upfront what you're building.
- Works well for projects with rigid, well-known requirements (government contracts, hardware-coupled software).

Cons

- By the time the software ships (months later), the requirements may have changed.
- Bugs found in testing are expensive - they may require redoing design.
- Users don't see the product until the end, so feedback comes too late.

Agile

Agile is a philosophy, not a specific process. It emphasises:

- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.
- Individuals and interactions over processes and tools.

In practice: instead of one big delivery at the end, deliver small working pieces every 1-4 weeks. Get user feedback. Adjust. Repeat.

Scrum

Scrum is the most popular Agile *framework*. It gives Agile concrete structure:

Term	Meaning
Sprint	Fixed time-box (usually 2 weeks) of focused work.
Product Backlog	All possible work, prioritised.
Sprint Backlog	Subset chosen for this sprint.
Daily Standup	15-minute team meeting, every day.
Sprint Review	Demo what was built at the end of the sprint.
Retrospective	What went well? What could improve? Done after each sprint.
Product Owner	The person who decides what gets built next.
Scrum Master	The person who keeps the process running smoothly.

Kanban

Kanban is the simplest Agile method. No sprints, no fixed cadence. Just a board with columns - *Todo*, *In Progress*, *Done* - and cards that move left to right. The rule: limit how many cards can be in *In Progress* at once. Forces focus.

Which methodology should you use?

Use Waterfall when...	Use Agile/Scrum when...
Requirements are crystal clear and won't change.	Requirements are vague or evolving.
The end product must be perfect on day one.	It's OK to ship a v1 and improve.
Heavy regulatory / contractual constraints.	Fast-moving market or tech.
Hardware involved (printed circuit boards, devices).	Pure software, frequent deployments.
Solo developer, very small project.	Team work, ongoing product.

Most modern web teams use Scrum or Kanban. Most traditional engineering still uses Waterfall. Many real teams use a hybrid: Waterfall for the first version, Agile for improvements.

DevOps and CI/CD - the modern bonus

Beyond methodology, modern teams use **DevOps** practices to shorten the cycle even further:

- **CI** (Continuous Integration) - every commit triggers automated tests. You know within minutes if something is broken.
- **CD** (Continuous Deployment) - passing tests automatically deploy to production. Some teams deploy 50+ times a day.
- **Infrastructure as Code** - servers and networks defined in code (Terraform, Ansible) so the whole environment is reproducible.
- **Monitoring and alerting** - production is observable. Problems trigger pager alerts.

DevOps blurs the line between development, testing, and operations. Bugs are caught earlier, fixes ship faster.

Final thoughts

The SDLC isn't a rigid set of rules. It is a **set of questions** every project must answer:

- What problem are we solving? (Planning)
- What exactly should the software do? (Requirements)
- How will we structure it? (Design)
- How do we build it? (Development)
- Does it actually work? (Testing & QA)
- How do we get it to users and keep it running? (Deployment & Maintenance)

Waterfall, Agile, Scrum - they all answer these same questions, just in different orders and at different speeds. Pick the approach that matches your project's needs, your team's size, and the speed of change in your domain.

You now understand how real software is built. The next step is to apply this on a real project of your own. Pick a small idea - a bookmark organiser, a habit tracker, a recipe site - and run it through every phase. The first time you do it, the process feels heavy. By the third project, it will feel natural. By the tenth, you will spot scope creep and missing test cases instinctively.

Welcome to professional software development. Happy building!

Quick Reference Card

The 6 SDLC phases

Phase	Purpose	Output
1. Planning	Decide if and what to build.	Project blueprint
2. Requirements	Detail what software must do.	SRS / user stories
3. Design	Plan how to build it.	Wireframes, ERD, architecture
4. Development	Write the code.	Working software
5. Testing & QA	Find and fix problems.	Test plan, bug reports
6. Deployment	Ship it. Keep it running.	Live system, monitoring

Most-used Git commands

Command	What it does
git status	What's changed?
git diff	Show line-by-line changes
git add file	Stage a file
git add .	Stage everything
git commit -m "message"	Save staged changes
git push	Send to GitHub
git pull	Fetch latest from GitHub
git checkout -b branch-name	Create + switch to a new branch
git checkout main	Switch to main
git log --oneline	Compact commit history
git branch	List branches
git merge feature-branch	Merge a branch into current

Bug states

State	Meaning
New	Reported, not yet checked
Confirmed	QA reproduced it
Assigned	Developer responsible
In Progress	Being worked on
Resolved	Code fixed, awaiting QA
Verified	QA confirmed fix
Reopened	Bug came back; restart cycle

Bug severity quick guide

Severity	Action
Critical / Blocker	Drop everything. Hotfix immediately.
High / Major	Fix in current sprint.
Medium	Fix in next sprint.
Low / Minor	Backlog. Fix when convenient.

Methodology cheat sheet

Methodology	Best for
Waterfall	Fixed scope, regulated industries, hardware projects
Agile	Evolving requirements, web apps, startups
Scrum	Teams of 5-9, 2-week iterations, regular delivery
Kanban	Continuous flow, support teams, no fixed iterations
DevOps	Frequent deployments, automated pipelines, observability

Final project structure

```
wanderly/
+-- public/                # what the web server serves
|   +-- index.html
|   +-- destinations.html
|   +-- booking.html
|   +-- 404.html
|   +-- about.html
|   +-- css/
|   |   +-- main.css
|   |   `-- responsive.css
|   +-- js/
|   |   +-- main.js
|   |   +-- destinations.js
|   |   `-- booking.js
|   `-- images/
+-- backend/               # PHP scripts
|   +-- save_booking.php
|   +-- get_destinations.php
|   +-- db.php             # in .gitignore
|   +-- db.example.php
|   `-- admin/
|       +-- index.php
|       `-- bookings.php
+-- database/
|   `-- schema.sql         # CREATE TABLE statements
+-- docs/
|   +-- blueprint.md
|   +-- srs.md
|   `-- design/
|       `-- wireframes.fig
+-- tests/
|   `-- manual_test_plan.md
+-- .gitignore
`-- README.md
```

Documents you'll produce

Document	Phase	Length
Project blueprint	Planning	1-3 pages
SRS	Requirements	5-50 pages
Wireframes	Design	1 per page
High-fidelity mockups	Design	Figma file
ER diagram	Design	1 picture
README.md	Development	1-2 pages
Test plan	Testing	10-100 cases
Deployment runbook	Deployment	1-3 pages

Where to learn more

- **Atlassian Agile Coach** (atlassian.com/agile) - free, in-depth articles on Scrum and Kanban.
- **The Pragmatic Programmer** (book) - timeless software engineering wisdom.
- **Code Complete** (book) - exhaustive guide to writing good code.
- **The Phoenix Project** (book) - DevOps explained as a novel.
- **egotechworld.com** - more Sinhala and English coding tutorials, free project source code, and AI tools.

Closing note from egotechworld.com

If this tutorial helped you, share it with a friend who's learning to code. Bookmark **egotechworld.com** for more tutorials on PHP, Python, React, Laravel, Node, and the SDLC. We post new content regularly and welcome guest articles from learners like you.

Happy building!