Programming

Concepts, patterns, language notes, and programming-related knowledge that goes beyond single scripts or quick fixes.

- Angular 20
  - Part 1: Building Your First Angular 20 Application
  - Part 2 — Structuring User Interfaces with Components
- Git
  - Git-Grundlagen: Fork vs. Branch
- Golang
  - Basic Installation of Go and Writing Your First Program
  - Understanding Go Slices: The Mechanics of Reference Types
  - The Append Behavior: Length, Capacity, and the "Resizing" Trap
- Laravel
  - Suggested Learning Path
  - Links and Resources

Angular 20

Angular 20

Part 1: Building Your First Angular 20 Application

Web development has exploded in the last decade. Frameworks like **Angular** lead the charge, emphasizing **performance**, **developer ergonomics**, and **modern web standards**. Angular 20, released in May 2025, takes this further with stabilized signals, zoneless change detection (in developer preview), and a streamlined naming convention that drops those pesky suffixes like `.component.ts` by default.

In this guide, we'll set up your first Angular 20 app using the **Angular CLI**. Expect cleaner file names, faster reactivity, and less boilerplate. We'll keep it practical, step-by-step, and inspired by Flavio Copes — code-first, no fluff.

Requirements

Ensure your setup includes these essentials for a smooth Angular 20 workflow:

Tool	Why You Need It	Install / Check
Node.js (LTS, e.g., 20.x or later)	Powers the CLI and runs TypeScript	https://nodejs.org `node -v`
npm	Manages packages (bundled with Node)	`npm -v`
Git (optional)	Version control for your projects	https://git-scm.com `git --version`

> **Pro Tip**: Switch versions easily with `nvm`:
>
> ```
> nvm install --lts
> nvm use --lts
> ```

> Angular 20 supports modern evergreen browsers — check details at angular.dev/reference/versions#browser-support.

Install the Angular CLI

The **Angular CLI** is your go-to tool for scaffolding, testing, and deploying. It now aligns with Angular 20's new style guide, generating suffix-free files by default (e.g., `app.ts` instead of `app.component.ts`).

Install globally:

```
npm install -g @angular/cli@20
```

> “
> Windows? Run as **Administrator**.
> macOS/Linux? Add `sudo` if prompted:
>
> ```
> sudo npm install -g @angular/cli@20
> ```

Verify:

```
ng version
```

Output should show **Angular CLI 20.x.x** and related tools.

Create Your First Angular 20 App

Generate a new project named `my-blog-app`:

```
ng new my-blog-app
```

Prompts you'll see:

```
? Would you like to share pseudonymous usage data...? (y/N) ? N
? Which stylesheet format would you like to use? ? CSS
? Do you want to enable Server-Side Rendering (SSR)...? (y/N) ? N
```

Hit **Enter** for defaults (SSR is optional; we'll skip for simplicity).

The CLI will:

- Create `my-blog-app/` folder
- Install dependencies (including Angular 20 core)
- Apply the new naming: No `.component` suffixes!

This takes 1–3 minutes. Pro tip: Angular 20's CLI is faster thanks to optimized builds.

> **Legacy Mode?** If you prefer old-school suffixes, add `--strict=false` or configure in `angular.json` later.

Run the App

Enter the project:

```
cd my-blog-app
```

Launch the dev server:

```
ng serve
```

> Alias: `ng dev` for quick starts.

After building (faster in v20!), open:

http://localhost:4200

Boom — the Angular welcome page loads! Live reload is on: Edit code, save, and watch updates instantly.

Project Structure (Updated for Angular 20)

Angular 20 keeps things lean. Key folders/files:

```
my-blog-app/
??? src/                  ? Your source code hub
?   ??? app/
?   ?   ??? app.ts        ? Main component (no .component.ts!)
?   ?   ??? app.html      ? Template (no .component.html)
?   ?   ??? app.css       ? Styles (no .component.css)
?   ?   ??? app.config.ts ? App providers
?   ?   ??? app.routes.ts ? Routing config
?   ??? index.html        ? Entry HTML
?   ??? main.ts           ? Bootstrap
?   ??? styles.css        ? Global CSS
??? angular.json          ? Workspace config
??? package.json          ? Dependencies
??? tsconfig.json         ? TypeScript setup
```

> Focus on `src/app/` — that's your playground. New naming reduces clutter: `app.ts` handles logic, `app.html` the markup.

Make Your First Change

Tweak the welcome message to feel the reactivity.

1. Edit the Main Component

Open `src/app/app.ts`:

```
export class AppComponent {
  title = 'My Awesome Blog';  // Updated title
}
```

2. Update the Template

Open `src/app/app.html` and find (around line 20):

```
<h1>Hello, {{ title }}</h1>
```

Change to:

```
<h1>Welcome to {{ title }}! ?</h1>
```

Save. Browser auto-refreshes — see **"Welcome to My Awesome Blog! ?"**?

This uses **interpolation** (`{{ }}`) for data binding. Angular 20's signals make this even snappier under the hood.

How It Works (Quick Peek Under the Hood)

1. `index.html` has `<app-root></app-root>` — Angular's mount point.
2. `main.ts` bootstraps:
   ```
   bootstrapApplication(AppComponent, appConfig);
   ```
3. `app.ts` (root component) renders into the DOM.
4. Signals (stable in v20) handle reactivity efficiently — no more full tree diffs by default.

Useful Angular CLI Commands

Command	Alias	What It Does
`ng new <name>`	`n`	Scaffold a new app
`ng serve`	`dev`	Dev server with HMR
`ng build`	`b`	Production build
`ng generate component <name>`	`g c`	New component (suffix-free!)
`ng test`	`t`	Run tests
`ng update`	-	Upgrade to latest

Docs: angular.dev/cli

> Generate with legacy suffixes: `ng g c my-comp --suffix=component`.

Recommended Tools

1. **VS Code** + Extensions

- Angular Language Service — Auto-complete in templates.
- Material Icon Theme — Spot Angular files easily.

2. **Angular DevTools** (Chrome/Firefox)

Profile components, inspect signals: angular.dev/tools/devtools. v20 adds OnPush badges!

3. **Communities**

- Tech Stack Nation — Beginner-friendly study group.
- Angular Discord — Official hub.

What's Next?

Your app's running — level up:

```
ng g c blog-post  # Creates blog-post.ts/html/css (no suffixes)
```

Add to `app.html`:

```
<app-blog-post></app-blog-post>
```

Explore signals for state: `signal('Hello')`. Dive into routing or SSR next.

> Angular 20's zoneless preview? Opt-in for blazing-fast apps.

Final Words

You've built an Angular 20 app: Cleaner names, stable signals, and CLI magic. It's not just a framework — it's a full ecosystem for scalable web apps.

Build iteratively. Experiment. The future's reactive.

Inspired by Flavio Copes — practical, dev-focused.
Based on Angular 20 docs and Learning Angular.

Angular 20

Part 2 — Structuring User Interfaces with Components

Angular applications are built from components: small, focused building blocks that each own a part of the user interface and its behavior.

In this chapter you will learn:

- How an Angular 20 component is structured
- How the CLI generates components with the new naming scheme
- How to display and control data in templates (with the modern `@if`, `@for`, `@switch` syntax)
- How components talk to each other using inputs and outputs
- How to style components and manage CSS encapsulation
- How lifecycle hooks and change detection work at a high level
- Where older syntax (`*ngIf`, `@Input`, etc.) still appears and how to read it

---

Anatomy of an Angular Component (Angular 20 Style)

In Angular 20, when you generate a component, the CLI now uses **simpler file names**:

- `product-list.ts` (instead of `product-list.component.ts`)
- `product-list.html`
- `product-list.css` (Ninja Squad Blog)

The idea is to reduce redundancy: the file name already tells you what this unit is.

A minimal root component looks like this:

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css',
  imports: [RouterOutlet],
  standalone: true
})
export class App {
  title = 'World';
}
```

Explanation:

- `selector: 'app-root'` – the tag you will use in `index.html`.
- `templateUrl` / `styleUrl` – point to the external HTML and CSS files.
- `imports: [RouterOutlet]` – because Angular 16+ uses **standalone components**, every component explicitly imports what it needs (other components, directives, pipes).
- `standalone: true` – tells Angular that this class stands on its own and is not declared in an NgModule.
- The class is named `App` (not `AppComponent`) to match the new naming style.

Legacy note: In older Angular versions, you would typically see:

- File: `app.component.ts`
- Class: `AppComponent`
- No `standalone: true` and no `imports` array (components were declared in NgModules).

Creating a Component with the CLI

To generate a feature component in Angular 20:

```
ng generate component product-list
```

With the new naming convention, this will create:

```
src/app/product-list/product-list.ts
src/app/product-list/product-list.html
src/app/product-list/product-list.css
src/app/product-list/product-list.spec.ts
```

The TypeScript file might look like this:

```
// src/app/product-list/product-list.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true
})
export class ProductList {
}
```

Explanation:

- The class name is `ProductList` (not `ProductListComponent`), consistent with the updated Angular 20 style guide.(Ninja Squad Blog)
- This component does not import anything yet; we will add imports later when needed.
- `standalone: true` makes this component directly usable in other components via the `imports` array.

To **use** this component inside `App`, you import it and add it to the `imports` array:

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductList } from './product-list/product-list';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  styleUrl: './app.css',
  standalone: true,
  imports: [RouterOutlet, ProductList]
})
export class App {
  title = 'World';
}
```

And in `app.html`:

```
<!-- src/app/app.html -->
<div class="content">
  <app-product-list></app-product-list>
</div>
```

Explanation:

- Importing `ProductList` in `App` and putting it into `imports` makes Angular aware of the `app-product-list` selector in this template.
- The template then simply uses `<app-product-list>`, which Angular binds to the `ProductList` class.

Displaying Data in the Template

Component templates can render values from the class using **interpolation** or **property binding**.

Interpolation

```
<h1>Hello, {{ title }}</h1>
```

Explanation:

- `{{ title }}` is interpolation; Angular evaluates `title` in the component instance and inserts its string value into the DOM.

Property binding

```
<h1 [innerText]="title"></h1>
```

Explanation:

- `[innerText]="title"` binds the DOM property `innerText` of the `<h1>` to the `title` property of the component.
- The square brackets indicate **one-way binding** from the component to the DOM.

Modern Control Flow: @if, @for, @switch

Angular 17+ introduced a new control-flow syntax that is more readable and more efficient than the older directive-based approach.

Conditional rendering with @if

Example with a product list:

```
// product-list.ts
import { Component } from '@angular/core';

interface Product {
  id: number;
  title: string;
}

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true
})
export class ProductList {
  products: Product[] = [];
}
```

```
<!-- product-list.html -->
@if (products.length > 0) {
  <h1>Products ({{ products.length }})</h1>
} @else {
  <p>No products found!</p>
}
```

Explanation:

- `@if` decides whether the block of HTML should exist in the DOM at all.
- If `products.length > 0`, Angular adds the `<h1>` to the DOM; otherwise, it adds the `<p>`.

Legacy note (older Angular):

```
<h1 *ngIf="products.length > 0">Products ({{ products.length }})</h1>
<p *ngIf="products.length === 0">No products found!</p>
```

`*ngIf` is still supported, but the new `@if` syntax is the recommended style going forward.

Looping over data with @for

Let’s populate some mock products:

```
// product-list.ts
export class ProductList {
  products: Product[] = [
    { id: 1, title: 'Keyboard' },
    { id: 2, title: 'Microphone' },
    { id: 3, title: 'Web camera' },
    { id: 4, title: 'Tablet' }
  ];
}
```

Now, use `@for` in the template:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
```

```
    <li class="pill">{{ product.title }}</li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

Explanation:

- `@for (product of products; track product.id)` iterates over `products` and exposes each item as `product`.
- `track product.id` tells Angular to use the `id` field to keep DOM nodes stable when items change, improving performance.
- `@empty` defines what to show when `products` is an empty array.

Legacy note:

```
<li *ngFor="let product of products">{{ product.title }}</li>
```

`*ngFor` is the older syntax with similar behavior.

Switching templates with @switch

You can pick different content based on a value:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill">
      @switch (product.title) {
        @case ('Keyboard') { ? }
        @case ('Microphone') { ? }
        @default { ? }
      }
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

Explanation:

- `@switch (product.title)` compares the title for each product.
- `@case` defines what to render when the expression matches a specific value.
- `@default` is rendered when no case matches.

Legacy note:

```
<div [ngSwitch]="product.title">
  <span *ngSwitchCase="'Keyboard'">?</span>
  <span *ngSwitchCase="'Microphone'">?</span>
  <span *ngSwitchDefault>?</span>
</div>
```

Again, `[ngSwitch]` and `*ngSwitchCase` are the older equivalents.

Handling User Interaction (Event Binding)

To send information from the template back to the component, Angular uses **event bindings**.

Extend the `ProductList` class:

```
// product-list.ts
export class ProductList {
  products: Product[] = [
    { id: 1, title: 'Keyboard' },
    { id: 2, title: 'Microphone' },
    { id: 3, title: 'Web camera' },
    { id: 4, title: 'Tablet' }
  ];

  selectedProduct: Product | undefined;
}
```

Update the template:

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>

@if (selectedProduct) {
  <p>You selected: <strong>{{ selectedProduct.title }}</strong></p>
}
```

Explanation:

- `(click)="selectedProduct = product"` listens for the browser's `click` event and executes the assignment in the component instance.
- `selectedProduct` becomes the currently clicked product, and the `@if` block below reacts by showing its title.

Styling Components and View Encapsulation

Angular lets you bind classes and styles dynamically.

Class binding

```
<li
  class="pill"
  [class.selected]="selectedProduct && selectedProduct.id ===
product.id"
>
  {{ product.title }}
</li>
```

Explanation:

- `[class.selected]="...condition..."` will add or remove the `selected` class based on whether the condition evaluates to `true` or `false`.

You can also bind an entire object:

```
// product-list.ts
isSelected(product: Product) {
  return this.selectedProduct?.id === product.id;
}
```

```
<li
  class="pill"
  [class.selected]="isSelected(product)"
>
  {{ product.title }}
</li>
```

Style binding

```
<p [style.color]="selectedProduct ? 'green' : 'inherit'">
  {{ selectedProduct ? 'Product chosen' : 'No product selected' }}
</p>
```

Explanation:

- `[style.color]` controls a single style property dynamically, based on component state.

View encapsulation

By default, Angular scopes CSS per component (Emulated mode), so styles from `product-list.css` will only affect that component's template.

```
import { Component, ViewEncapsulation } from '@angular/core';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true,
  encapsulation: ViewEncapsulation.Emulated // default
```

```
})
export class ProductDetail {
}
```

If you explicitly set:

```
encapsulation: ViewEncapsulation.None
```

then styles defined in `product-detail.css` can leak into other parts of the app. This can be useful for global styling, but must be used carefully.

Passing Data Between Components (Inputs and Outputs)

Real-world applications rarely keep all UI in a single component. Often, a parent component owns the data and passes a piece of it down to a child component.

Passing data down with input()

Create a detail component:

```
// src/app/product-detail/product-detail.ts
import { Component, input } from '@angular/core';
import type { Product } from '../product-list/product-list';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail {
  product = input<Product>();
}
```

Template:

```
<!-- product-detail.html -->
@if (product()) {
  <p>
    You selected:
    <strong>{{ product()!.title }}</strong>
  </p>
}
```

Explanation:

- `product = input<Product>()` defines an input signal for this component.
- In the template, `product()` reads the current value of that input.
- The `@if` guard ensures we only render details when a product is actually provided.

Now, use `ProductDetail` in `ProductList`:

```
// product-list.ts
import { Component } from '@angular/core';
import { ProductDetail } from '../product-detail/product-detail';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true,
  imports: [ProductDetail]
})
export class ProductList {
  products: Product[] = [ /* ... */ ];
  selectedProduct: Product | undefined;
}
```

```
<!-- product-list.html -->
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      {{ product.title }}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

```
<app-product-detail [product]="selectedProduct"></app-product-detail>
```

Explanation:

- `[product]="selectedProduct"` binds the parent's `selectedProduct` property into the child's `product` input.
- Angular takes care of updating the child when the parent selection changes.

Legacy note: Previously, you would see:

```
@Input() product!: Product;
```

instead of `product = input<Product>()`.

Sending events up with output()

Let the detail component notify the parent that the user wants to add the product to a cart.

In `ProductDetail`:

```
import { Component, input, output } from '@angular/core';
import type { Product } from '../product-list/product-list';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail {
  product = input<Product>();
  added = output<Product>();

  addToCart() {
    if (this.product()) {
      this.added.emit(this.product()!);
    }
  }
}
```

Template:

```
<!-- product-detail.html -->
@if (product()) {
  <div>
    <p>
      You selected:
      <strong>{{ product()!.title }}</strong>
    </p>
    <button (click)="addToCart()">Add to cart</button>
  </div>
}
```

Explanation:

- `added = output<Product>()` declares an output event that can carry a `Product` payload.
- `this.added.emit(...)` triggers the event.

In the parent (`ProductList`):

```
// product-list.ts
onAdded(product: Product) {
  alert(`${product.title} added to the cart!`);
}
```

```
<!-- product-list.html -->
<app-product-detail
  [product]="selectedProduct"
  (added)="onAdded($event)"
></app-product-detail>
```

Explanation:

- `(added)="onAdded($event)"` listens to the child's `added` output.
- `$event` contains the product emitted by `addToCart()`.
- The parent can now update a cart, fire analytics, or display a message.

Legacy note: Older Angular projects use:

```
@Output() added = new EventEmitter<Product>();
```

instead of `added = output<Product>()`.

Template Reference Variables and viewChild

Sometimes you need direct access to a child component instance.

Template reference variable

```
<!-- product-list.html -->
<app-product-detail
  #detail
  [product]="selectedProduct"
  (added)="onAdded($event)"
></app-product-detail>

<p *ngIf="detail.product()">
  Detail says: {{ detail.product()!.title }}
</p>
```

Explanation:

- `#detail` creates a template reference to the `ProductDetail` instance.
- This reference exposes the public API of `ProductDetail` (here: `product()`).

Querying a child in TypeScript with viewChild

You can also get the child instance from the parent class:

```
// product-list.ts
import { Component, AfterViewInit, viewChild } from '@angular/core';
import { ProductDetail } from '../product-detail/product-detail';

@Component({
  selector: 'app-product-list',
```

```
  templateUrl: './product-list.html',
  styleUrl: './product-list.css',
  standalone: true,
  imports: [ProductDetail]
})
export class ProductList implements AfterViewInit {
  productDetail = viewChild(ProductDetail);

  ngAfterViewInit(): void {
    console.log('Detail product:', this.productDetail()?.product());
  }
}
```

Explanation:

- `viewChild(ProductDetail)` tells Angular to look for a `ProductDetail` in this component's view.
- `ngAfterViewInit` is the hook where the child is guaranteed to be created and accessible.

Legacy note: Previously:

```
@ViewChild(ProductDetail) productDetail!: ProductDetail;
```

Change Detection Strategy

Angular automatically refreshes views when data changes. By default, it runs change detection for the entire component tree on each relevant event.

You can optimize this with `ChangeDetectionStrategy.OnPush`:

```
import { Component, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true,
```

```
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProductDetail {
  // ...
}
```

Explanation:

- With `OnPush`, Angular will only re-check this component when:
  - An input reference changes
  - An event handler on this component runs
  - An observable bound in the template emits (via async pipe), etc.
- This significantly improves performance in large and complex UIs.

Lifecycle Hooks Overview

Lifecycle hooks allow you to run custom logic at specific moments in a component's life.

Common hooks:

- `ngOnInit` – runs after the component's inputs are first set.
- `ngOnDestroy` – runs right before Angular removes the component from the DOM.
- `ngOnChanges` – runs whenever an input binding changes.
- `ngAfterViewInit` – runs after the view and child views have been initialized.

Example:

```
import {
  Component,
  OnInit,
  OnDestroy,
  OnChanges,
  SimpleChanges,
```

```
  AfterViewInit
} from '@angular/core';

@Component({
  selector: 'app-product-detail',
  templateUrl: './product-detail.html',
  styleUrl: './product-detail.css',
  standalone: true
})
export class ProductDetail
  implements OnInit, OnDestroy, OnChanges, AfterViewInit {

  ngOnInit(): void {
    // Good place to fetch data or initialize values
    console.log('ProductDetail initialized');
  }

  ngOnChanges(changes: SimpleChanges): void {
    // React to input changes
    console.log('Changes:', changes);
  }

  ngAfterViewInit(): void {
    // Child components and view are ready
    console.log('View initialized');
  }

  ngOnDestroy(): void {
    // Cleanup: timers, subscriptions, listeners, etc.
    console.log('ProductDetail destroyed');
  }
}
```

Explanation:

- Each hook gives you a predictable place to put specific kinds of logic:
  - `ngOnInit` instead of doing heavy work in the constructor.
  - `ngOnDestroy` to release resources.
  - `ngOnChanges` to react to new input values.
  - `ngAfterViewInit` to work with child components or DOM elements that weren’t available earlier.

Summary

In this chapter you have seen how, in **Angular 20**:

- Components are generated with **simpler file names** like `product-list.ts`, `product-list.html`, `product-list.css`.
- Standalone components (`standalone: true`) and explicit `imports` have become the default way to structure an app.
- The modern control-flow syntax (`@if`, `@for`, `@switch`) replaces older structural directives in new code, while you still need to understand `*ngIf`, `*ngFor` and `ngSwitch` for legacy templates.
- Data flows into components via `input()` and out via `output()`, replacing `@Input` and `@Output` in new code.
- Class and style bindings, along with view encapsulation, give you fine control over component-level CSS.
- Template reference variables and `viewChild` let you reach deeper into the component tree when necessary.
- Change detection strategies and lifecycle hooks help you tune both performance and behavior.

Git

Git

Git-Grundlagen: Fork vs. Branch

Um die Versionsverwaltung effizient zu nutzen, ist es wichtig, den Unterschied zwischen einem **Branch** (Zweig) und einem **Fork** (Abspaltung) zu verstehen. Beide dienen der Isolation von Code-Änderungen, setzen aber auf unterschiedlichen Ebenen an.

1. Der Branch (Der Zweig)

Ein Branch ist ein Zeiger auf einen bestimmten Entwicklungsstand innerhalb eines **einzelnen Repositorys**. Er ist die kleinste Einheit, um Änderungen getrennt vom Hauptcode (meist `main` oder `master`) zu entwickeln.

- **Ebene:** Repository-intern.
- **Lebenszyklus:** Ein Branch ist oft kurzlebig. Er wird für ein Feature oder einen Bugfix erstellt und nach Abschluss der Arbeit per *Merge* oder *Rebase* wieder in den Hauptstamm integriert.
- **Berechtigung:** Man benötigt Schreibrechte für das Repository, um einen Branch zu pushen.
- **Praxis-Beispiel:** Wenn ich in meinem Dashboard-Projekt eine neue Funktion teste, erstelle ich dafür einen Branch, um den stabilen Code nicht zu gefährden.

2. Der Fork (Die Abspaltung)

Ein Fork ist eine **vollständige Kopie** eines gesamten Repositorys unter einem neuen Besitzer-Account. Technisch gesehen ist es ein neues, eigenständiges Repository, das jedoch die Verbindung zum Original (dem „Upstream“) beibehält.

- **Ebene:** Account- / Server-Ebene.
- **Lebenszyklus:** Ein Fork ist oft langlebiger. Er wird genutzt, um unabhängig am gesamten Projekt zu arbeiten, ohne das Original zu beeinflussen.
- **Berechtigung:** Jeder kann einen Fork von einem öffentlichen Projekt erstellen, ohne Schreibrechte am Original zu besitzen.

- **Praxis-Beispiel:** Bei komplexen Experimenten (z. B. wenn K.I.-Agenten großflächig Code umbauen) nutze ich einen Fork. So kann ich das gesamte Projekt spiegeln und experimentieren, ohne mein Haupt-Repo mit unzähligen Test-Branches zu fluten.

3. Direkter Vergleich

Merkmal	Branch	Fork
Speicherort	Im selben Repository	In einem neuen, eigenen Repository
Abhängigkeit	Fest mit dem Hauptprojekt verbunden	Eigenständig (mit Link zum Original)
Zusammenführung	`git merge` oder `git rebase`	Pull Request (PR) an das Original
Sichtbarkeit	Für alle Projektbeteiligten sichtbar	In meinem eigenen Account-Bereich

4. Kombination im Workflow

In der Praxis werden beide Konzepte oft kombiniert. Ein typischer Workflow sieht so aus:

1. Man erstellt einen **Fork** eines Projekts, um eine eigene Arbeitsumgebung zu haben.
2. Innerhalb dieses Forks arbeitet man mit verschiedenen **Branches**, um einzelne Features sauber zu trennen.
3. Ist ein Feature fertig, wird es im Fork gemerged und bei Bedarf per Pull Request dem Original-Projekt zur Verfügung gestellt.

Golang

Golang

Basic Installation of Go and Writing Your First Program

Before diving into complex projects, you need to set up your local environment so you can run, build, and compile Go code. Once the environment is ready, we will write a simple "Hello World" application to test it out.

Part 1: Installing the Go Runtime

To build and execute Go programs, you must install the Go Runtime.

1. Open your browser and navigate to `golang.org/dl`.
2. Find the download link for your operating system (macOS, Windows, or Linux) and grab the installer.
3. Run the installer and click through the standard installation prompts.
4. **Verify the installation:** Open your computer's terminal and type the single word `go`, then hit enter. You should see a long help message appear on the screen. This `go` command is the primary tool you will use to interact with the Go language.

Part 2: Configuring the Editor (VSCode)

While you can use any editor (like Atom or Sublime Text), **Visual Studio Code (VSCode)** is highly recommended because it offers some of the best built-in integration with Go.

1. Download VSCode from `code.visualstudio.com` and install it.
2. Open VSCode, navigate to the top menu bar, click on **View**, and select **Extensions**.
3. Search for "Go" and install the extension named **"Rich Go language support for Visual Studio"**.
4. **Important Step:** To ensure the extension can successfully install its underlying command-line tools, you must completely quit and restart the VSCode editor.

5. Open a new file and change the language mode in the bottom right corner to **Go**. A yellow prompt will appear saying **"Analysis Tools Missing"**—click **Install** to allow a terminal window to grab the final tools needed to analyze your code.

Part 3: Writing Your First Program

Now that the environment is ready, let's write a tiny application.

1. Create a new folder on your computer called `Hello World` and open this folder in your code editor.
2. Inside this directory, create a new file named `main.go`.
3. Add the following code exactly as it appears. Ensure you use double quotes (not single quotes) around your strings:

```
package main

import "fmt"

func main() {
    fmt.Println("Hi there")
}
```

Part 4: How to Run the Code

To run your project, open your terminal and navigate inside your `Hello World` directory. You can use the Go Command-Line Interface (CLI) to execute the code in two different ways:

- `go run main.go`: This command takes your file, compiles it, and immediately executes the result. When you run this, you will instantly see `Hi there` printed on the screen.
- `go build main.go`: This command will *only* compile your program; it does not execute it automatically. Running this will spit out a runnable executable file named `main` (or `main.exe` on Windows) directly into your folder. You can then execute that file manually.

Part 5: Breaking Down the Code

Even though this is a simple file, it reveals the fundamental structure of all Go programs.

- `package main`: A package is a collection of common source code files. The name `main` is sacred in Go; it specifically tells the compiler that you are making an *executable* package that will spit out a runnable file, rather than a reusable dependency library. Any time you make an executable package, it must contain a function called `main`.
- `import "fmt"`: By default, your package is isolated. The `import` statement gives your package access to functionality written in another package. `fmt` (short for "format") is a part of Go's Standard Library, and it is primarily used to print information out to the terminal.
- `func main()`: `func` is short for function. We declare a function by providing the keyword `func`, the function's name, a set of parentheses for arguments, and curly braces containing the body of our logic.

Golang

Understanding Go Slices: The Mechanics of Reference Types

In Go, passing a **struct** to a function typically results in an independent copy. If you modify that struct inside the function, the original remains untouched. However, slices exhibit a surprising behavior: modifying a slice inside a function updates the original caller's data. This often leads developers to believe Go has special rules for slices, but the behavior is actually a logical result of how Go manages memory and data structures.

The Anatomy: Slices vs. Arrays

To understand this, we must first distinguish between an array and a slice. In Go, an **array** is a primitive, fixed-length data structure. Because arrays cannot grow or shrink, they are rarely used directly. Instead, we use **slices**, which are essentially a sophisticated "header" that sits on top of an array.

When you declare a slice, Go internally creates two separate entities in memory. The first is the **slice header**, a small data structure containing three specific fields: a **pointer** to the underlying data, the current **length** of the slice, and its total **capacity**. The second entity is the **underlying array**, which contains the actual elements and exists at a separate memory address.

The "Pass-by-Value" Crux

Go is strictly a "pass-by-value" language. When you pass a slice into a function, Go does exactly what it does with a struct: it makes a copy of the value. However, the value being copied is the **slice header**, not the underlying array.

This is the "gotcha" of Go development. Even though the function receives a brand-new copy of the header, that copy contains the exact same memory address in its pointer field. Therefore, both the original slice header and the function's copy are pointing to the same underlying array. When you modify an element inside the function, you are reaching through the pointer to the "true" source of data in memory. This is why slices are categorized as **reference types**, alongside maps, channels, and pointers.

Contrast with Value Types

In contrast, **value types**—which include integers, floats, booleans, strings, and structs—behave differently. For these types, the "value" is the data itself. When you pass a struct, the entire set of fields is copied to a new memory location. Without using an explicit pointer (using the `*` and `&` operators), a function is only ever working on a local, temporary version of that data.

Code Example: Slices vs. Structs

The following code demonstrates how Go treats a value type (struct) versus a reference type (slice) when passed to a function.

```
package main

import "fmt"

type Person struct {
    Name string
}

func main() {
    // 1. Value Type Behavior (Struct)
    myPerson := Person{Name: "Alice"}
    updateStruct(myPerson)
    fmt.Println("Original Struct:", myPerson.Name) // Output: Alice
(Unchanged)

    // 2. Reference Type Behavior (Slice)
    mySlice := []string{"Apple", "Banana"}
    updateSlice(mySlice)
    fmt.Println("Original Slice:", mySlice[0])     // Output: Orange
(Changed!)
}

func updateStruct(p Person) {
    p.Name = "Bob"
}

func updateSlice(s []string) {
    s[0] = "Orange"
}
```

In the example above, `updateStruct` receives a full copy of the `Person` object, so the original `myPerson` remains "Alice." However, `updateSlice` receives a copy of the slice header. Since that header points to the same underlying array as `mySlice`, changing the first element to

"Orange" updates the data that both headers reference.

Golang

The Append Behavior: Length, Capacity, and the "Resizing" Trap

While passing a slice to a function allows you to modify existing elements, using the `append` function inside that same function introduces a common pitfall. To understand why, we have to revisit the **Slice Header**—the small data structure containing the pointer, length, and capacity.

The Mechanism of Append

When you call `append` on a slice, Go performs a specific set of operations:

1. It checks if the **capacity** of the underlying array is large enough to hold the new elements.
2. If there is room, it adds the elements to the array and increments the **length**.
3. If there is **not** enough room, Go allocates a brand-new, larger array, copies the old elements over, and updates the **pointer** to this new memory location.

Why Changes to Length/Capacity Don't "Stick"

Because Go is **pass-by-value**, the function receives a copy of the slice header. While this copy points to the same underlying array, the `length` and `capacity` fields are local variables within the function's scope.

- **Scenario A (Within Capacity):** If you append an item and the array has space, the function updates the shared underlying array. However, it only updates the **local copy** of the `length` field. When the function returns, the caller’s slice header still has the old `length`, so it "doesn't see" the new element, even though it exists in the array.
- **Scenario B (Exceeding Capacity):** If the append forces a reallocation, the function creates a new array. The local slice header's **pointer** is updated to this new address. The original slice header in the calling function still points to the **old array**. At this point, the two slices are completely disconnected.

Code Example: The Append Disconnect

```
package main
```

```
import "fmt"

func main() {
    // Slice with length 2, capacity 5
    mySlice := make([]string, 2, 5)
    mySlice[0] = "Stay"
    mySlice[1] = "Stay"

    attemptAppend(mySlice)

    fmt.Println("Original Slice Length:", len(mySlice)) // Output: 2
    fmt.Println("Original Slice Data:", mySlice)       // Output: [Stay
Stay]

    // Note: The data "Added" IS in the array, but the caller's
    // length field prevents us from seeing it.
}

func attemptAppend(s []string) {
    s = append(s, "Added")
    fmt.Println("Inside Function:", s) // Output: [Stay Stay Added]
}
```

The Solution: Returning the Slice

Because the slice header is passed by value, any operation that modifies the header itself (like changing the length or reallocating the pointer via `append`) must be communicated back to the caller. The standard Go idiom is to **return the updated slice**:

```
func main() {
    mySlice := []string{"Alpha"}
    mySlice = successfulAppend(mySlice)
    fmt.Println(mySlice) // Output: [Alpha Beta]
}

func successfulAppend(s []string) []string {
    return append(s, "Beta")
}
```

Alternatively, you could pass a **pointer to the slice** (`*[]string`), which allows the function to modify the caller's header directly, though returning the slice is generally considered cleaner and more idiomatic in the Go community.

Laravel

Suggested Learning Path

- Install + run first Laravel app
- Routing + controllers + Blade views
- Forms + validation + CSRF
- Database + Eloquent + migrations + seeders
- Auth + middleware + policies
- Build a small CRUD app (final project)
- If you want, I can set up a Laravel Sail project in this workspace now and start Lesson 1 immediately.

Laravel

Links and Resources

https://laravel.com/docs/12.x/

Laraval Sail