Code Structure

CS 346: Application Development

Getting Started

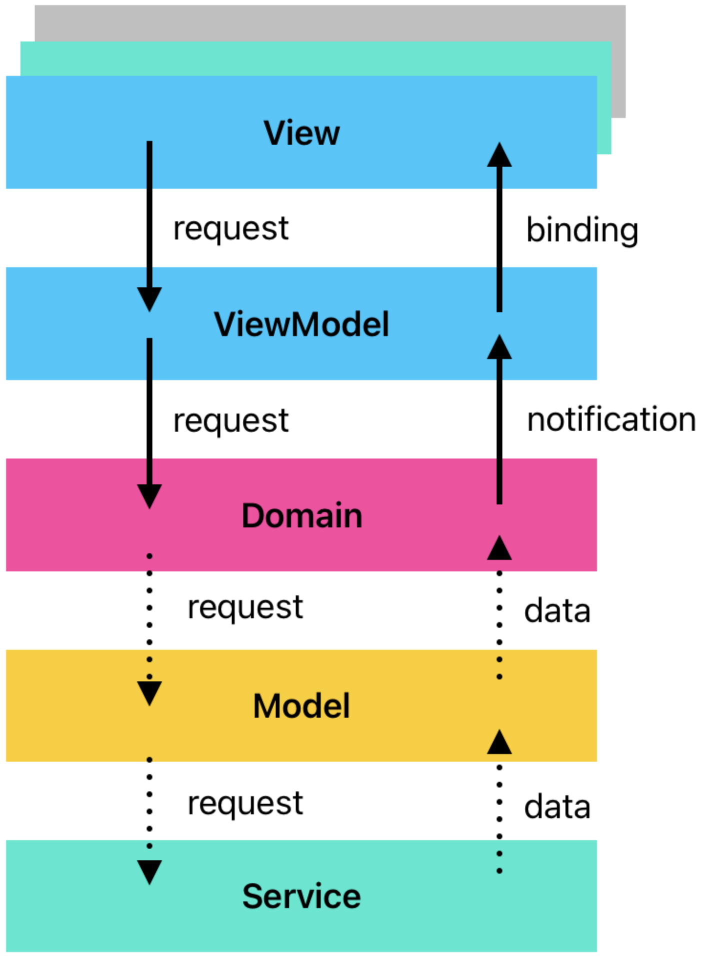
How do you *actually* get started?

- You know that:
  - You need layers for your user-interface, domain and model.
  - You have features that are orthogonal to these layers e.g., your recipe application will probably need to fetch data (model and persistence), modify and repackage it into a useful structure (domain) and display it (view).

Do you write all your views first? Do you build models first?

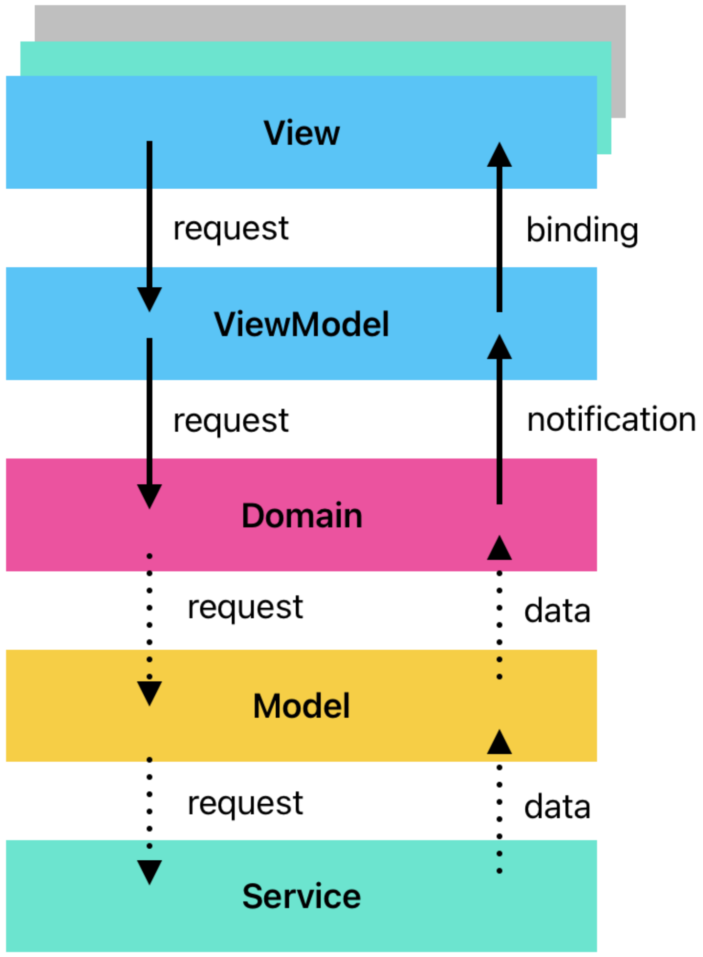
Two approaches

- Technical Partitioning
  - Also called Horizontal slicing
  - Idea that you build all like-minded classes first.
  - i.e. create the views first, then hook them up!
- Domain partitioning
  - Also called Vertical slicing
  - Idea that you build complete features first.
  - i.e., create View, Model, etc. for a single feature.
  - Test that, then iterate on the next one.



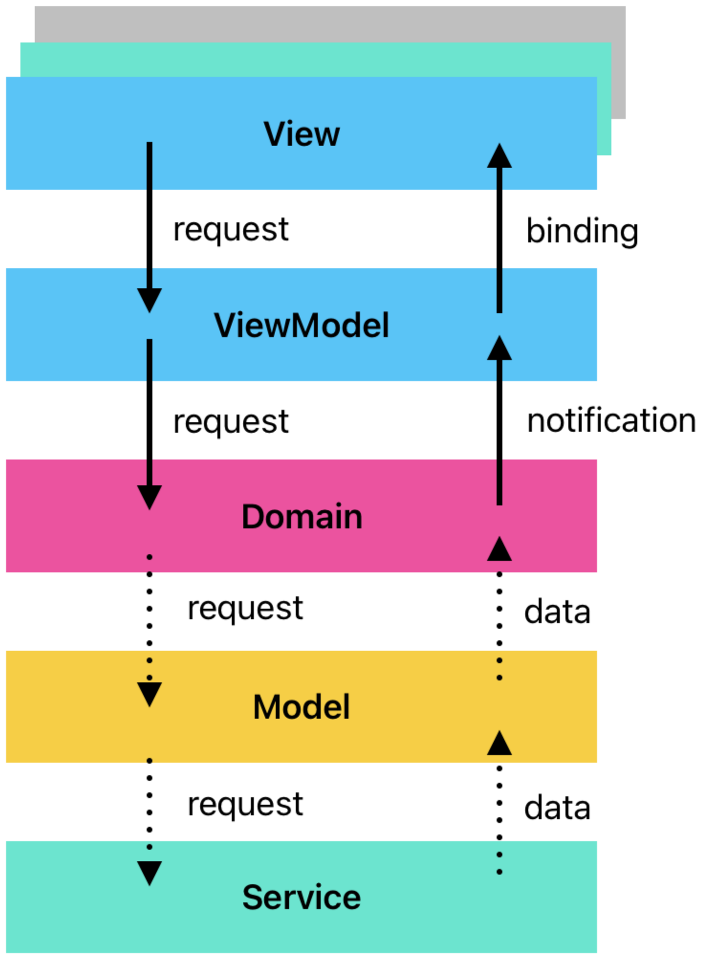
Technical Partitioning

- Horizontal slicing
- Build all like-minded classes first.
- When to use it?
  - This is a great approach if you need to solidify aspects of your design early.
  - Beneficial with a high-risk layer e.g., service.
- Advantages?
  - You can deliver a standalone layer earlier.
  - You will need to test each layer independently (unit tests only).



Domain Partitioning ⭐

- Vertical slicing
- Build a complete feature first.
- When to use it?
  - This approach lets you test your full development stack with each feature!
  - No rush to complete any one layer.
- Advantages?
  - You get working features to demo.
  - You will reduce integration challenges which are a major source of risk in projects.
  - You can write complete tests, every iteration (i.e. unit + integration tests).



What does Domain partitioning look like?

Your approach will determine the structure of your packages and code.

Domain partitioning has subfolders for each significant feature.

- e.g. a recipe application with both a card (detail) view and a list (summary) view of recipe data.

You implement that feature in its entirety!

Classes shared across features can be stored in a shared package.

```
.
├── main
│   └── kotlin
│       ├── recipe-card
│       │   ├── domain
│       │   ├── model
│       │   └── view
│       ├── recipe-list
│       │   ├── domain
│       │   ├── model
│       │   └── view
│       └── shared
│           ├── domain
│           ├── model
│           └── view
│
…
```