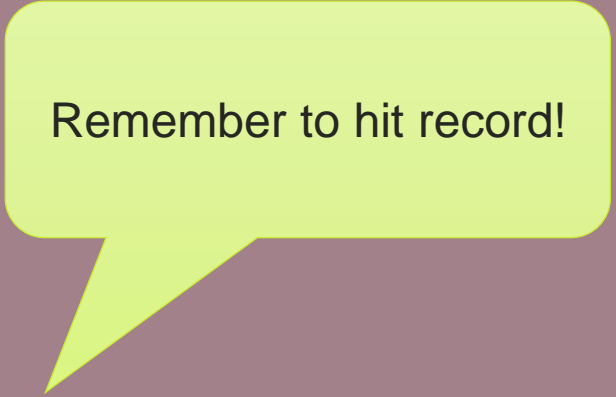


Application Architecture



Remember to hit record!

Valcademy Sept 2022
Chris Buckett

valcon

Agenda

Application Architecture Layers

- What are layers?
- A typical application architecture

Communicating between layers

- Passing data through service layers
- Coupling (tight vs loose)
- Dependency considerations

Application patterns

- Horizontal vs vertical scaling
- Monolith vs microservice
- Cloud vs On Prem
- Serverless vs Server Based

Cross cutting considerations

- Logging Monitoring and Alerting (LMA)
- Security & Privacy

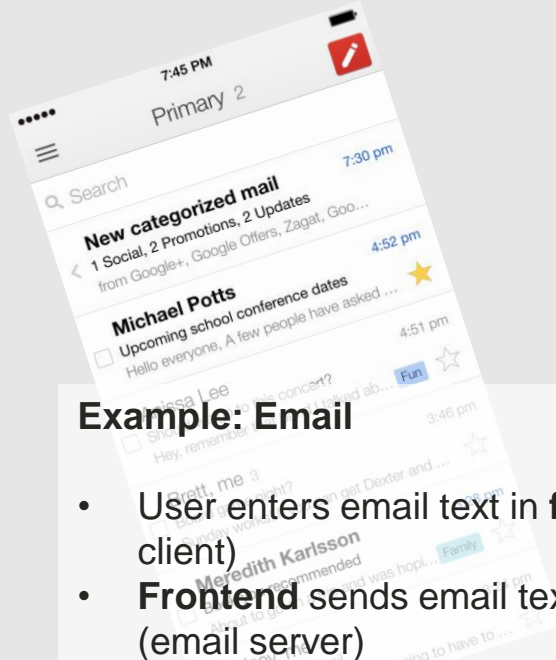
Application Architecture Layers

- What are layers
- Horizontal & vertical layers
- A typical application architecture at Valcon

What are application layers?

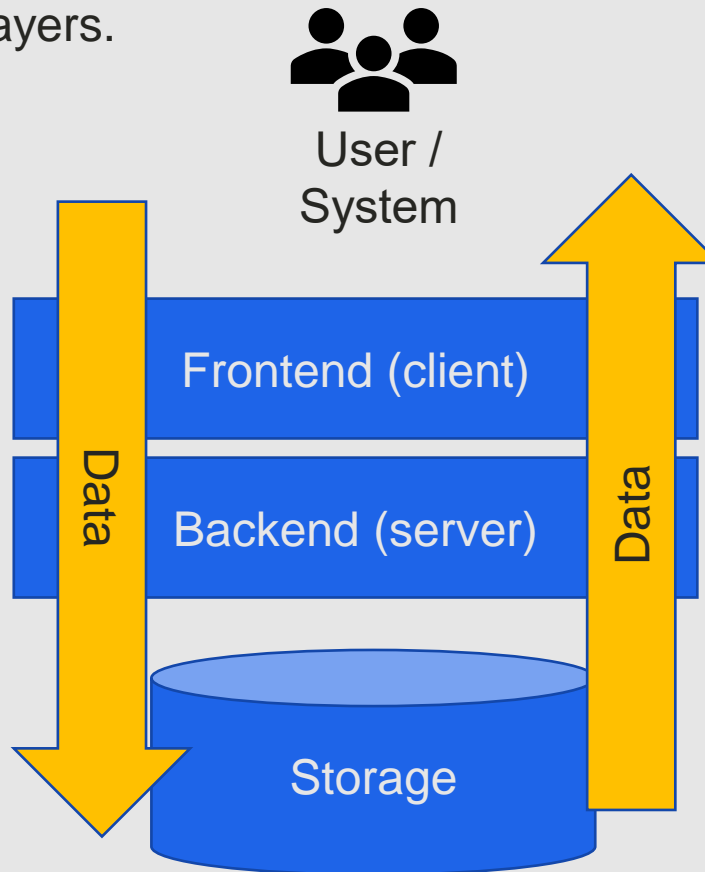
Application layers are discreet parts of application functionality that combine to produce an application.

Data flows up and down between the layers.



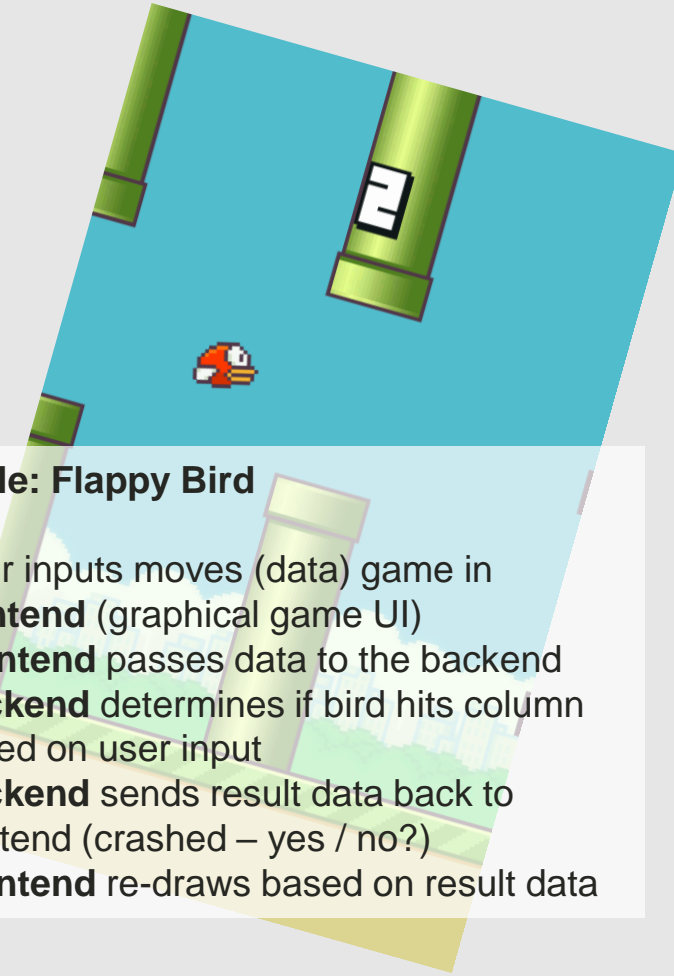
Example: Email

- User enters email text in **frontend** (email client)
- **Frontend** sends email text to **backend** (email server)
- **Backend** persists the message in storage
- **Backend** interfaces with other email servers to transfer message to designation



Example: Flappy Bird

- User inputs moves (data) game in **frontend** (graphical game UI)
- **Frontend** passes data to the backend
- **Backend** determines if bird hits column based on user input
- **Backend** sends result data back to frontend (crashed – yes / no?)
- **Frontend** re-draws based on result data



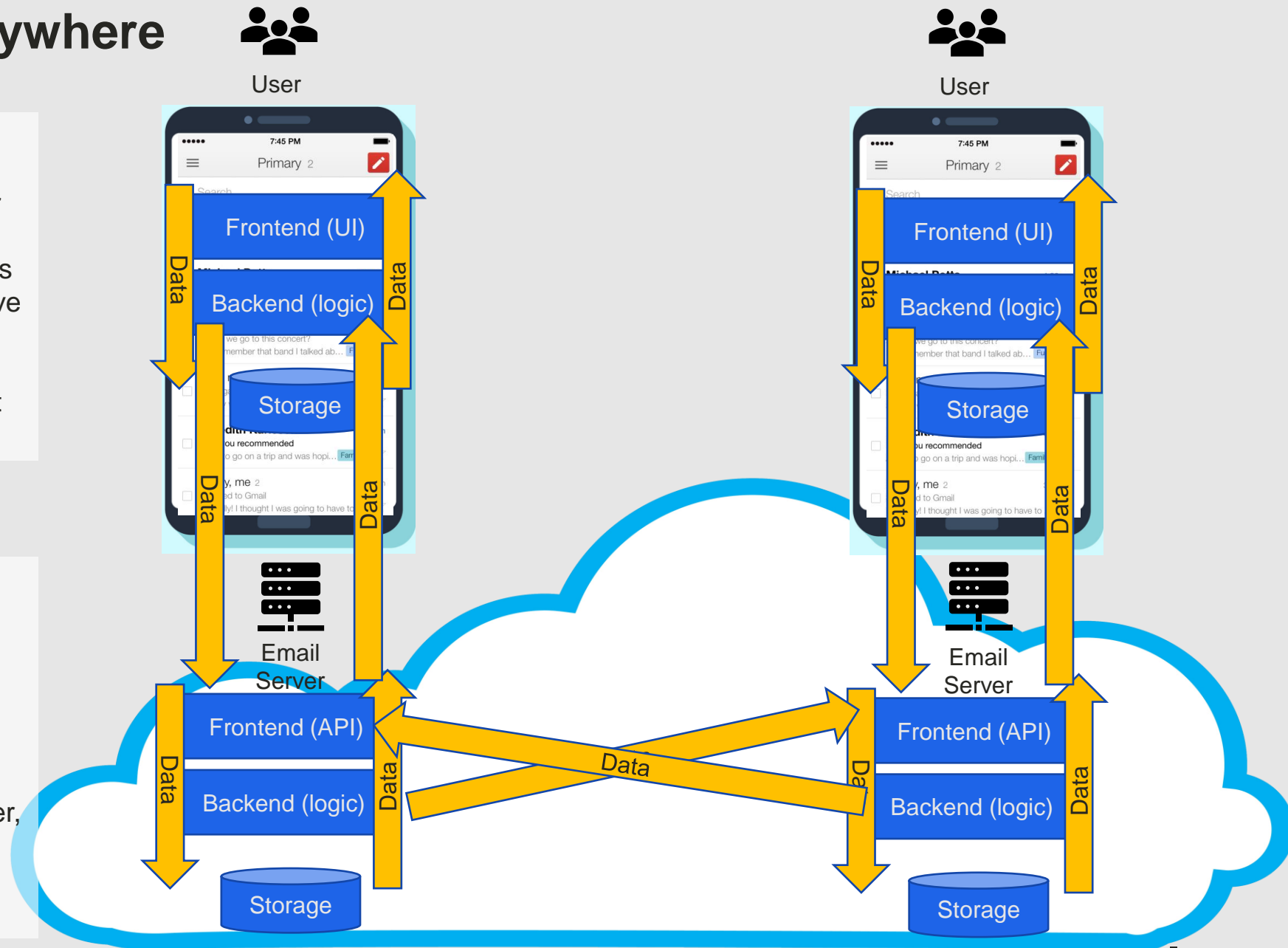
Application layers everywhere

Digging Deeper – Email – Mobile app

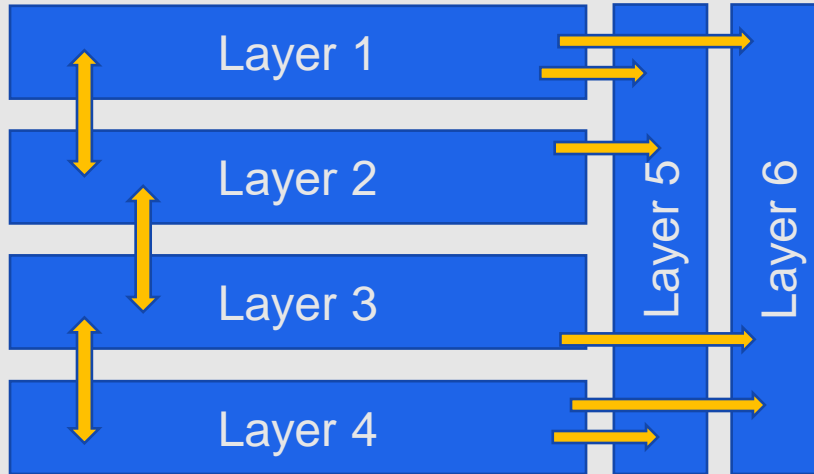
- Mobile app has a front-end that the user interacts with
- Mobile app has a back end that performs logic (am I logged in, does the email have an address, does it need delayed send...?)
- Mobile app has a local storage of recent messages.

Email – server code

- Email server has a “frontend” (although this is an API (application programming interface) – a defined way for other systems to talk to it.
- Email server backend performs logic (is the sender allowed to send on this server, is the message spam)
- Email server backend talks to OTHER email server frontends (APIs).

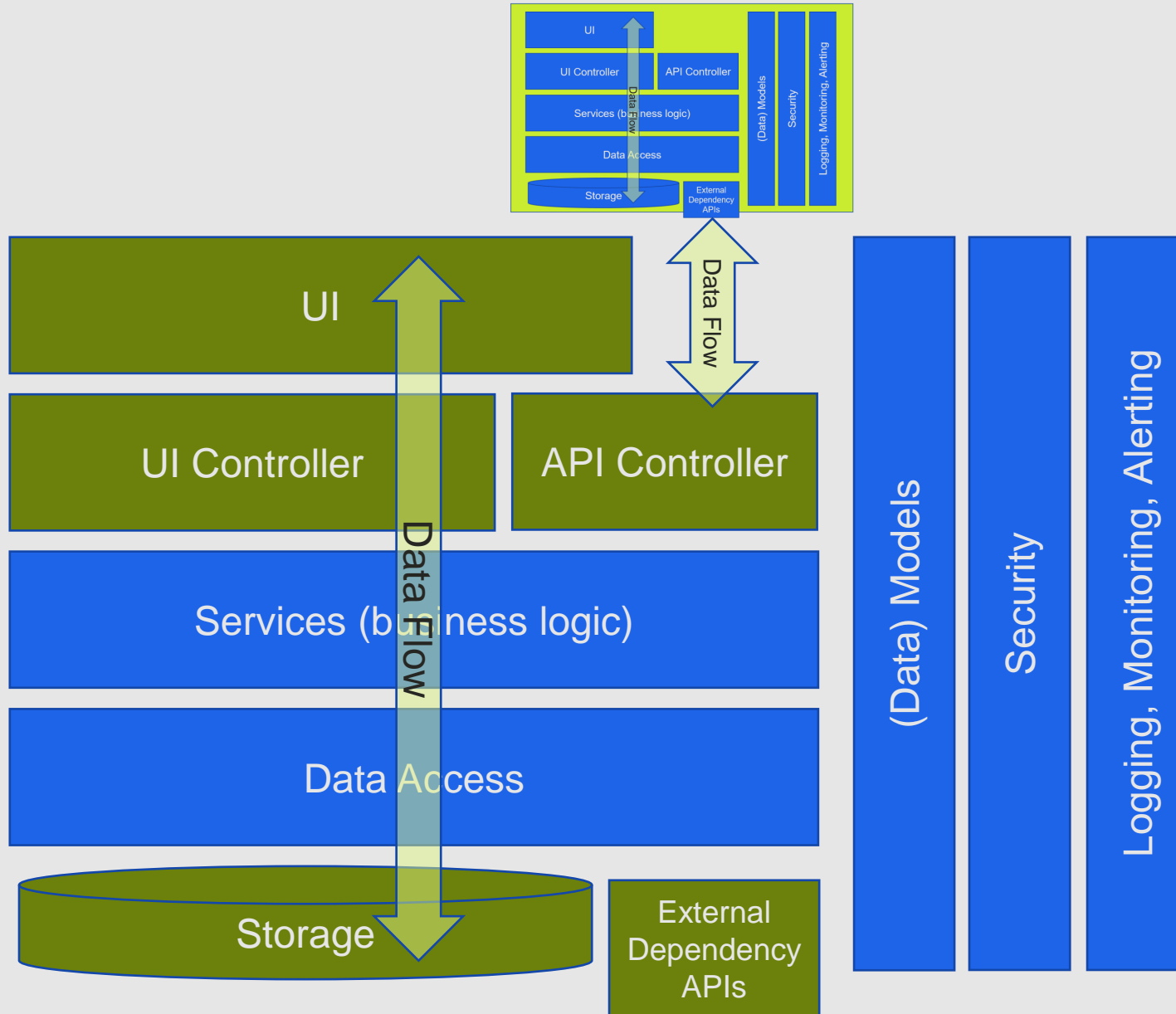


Horizontal and Vertical Layers



- Horizontal layers provide business functionality.
- Horizontal layers only talk directly to layers above or below them
- They can also use the vertical layers
- Vertical layers provide support to the horizontal layers.
- Vertical layers are used by the horizontal layers (not the other way around)
- These are commonly known as **cross-cutting concerns**
- Layers themselves may be blocks of code OR entire applications.

Architectural Application Layers



Clean layers defined by classes / packages / applications.

⚠ Service layers are stateless – they don't retain state of data passing through it. Make sure to distinguish between classes that contain data, and classes that process that data.

Each layer only talks to the next. Eg, UI / API controller code doesn't communicate directly with the database; instead it communicates through a Service layer.

Communicating between layers

- Service layers and data models
- Coupling – tight and loose
- Dependency considerations

Service vs model example

Data (defined by “model” classes) pass through service layers

- **UI Controller Layer: POST /api/v1/order**
 - Order order = // order from the UI
 - orderService.process(order) // UI controller interacts with the orderService
 - return order.status

- **OrderService.process(...Order...) (business logic layer)**
 - stockService.checkStock(order)
 - If (order.hasEnoughStock)
 - stockService.allocateStock(order)
 - invoicingService.sendInvoice(order)
 - shippingService.ship(order)
- **StockService.checkStock(...Order...) (business logic layer)**
 - For (item in order.items)
 - stockLevel = stockData.getStock(item.productCode)
 - if (stockLevel < order.items.qty)
 - items.hasEnoughStock = false

- **StockData.getStock(...OrderItem...)**
 - select qtyInStock from inventory where productCode = ...

```
// A Model class - only stores data
Order {
    items[]    // list of OrderItem
    hasEnoughStock // true/false
    status      // fulfilled / cancelled
}
```

Notes:

- Order class contains the state – the order request from the UI or API
- The services perform actions on the order, but once the service method is complete, the service class knows nothing about that specific order – stateless services.
- Each service can be implemented independently. Eg, stockService could call out to an external supplier API.
- Each layer (UI, service, data) can be replaced by an equivalent layer (eg, for testing,
 - the unit test might take the place of the UI and pass data to the UI controller, and validate the output.
 - the unit test might take the place of the orderService, and create mocks implementations for stockService, invoicingService and shippingService to test the logic in orderService.process.
- We can achieve this “replacing of layers” using dependency injection

Definitions: Coupling (and dependency injection)

Dependency: When layer 1 depends on layer 2 to provide some service, layer 1 has a dependency on layer 2.

Abstraction: Instead of letting layer 1 know directly about layer 2, instead define an interface (ie, tell layer 1 that a layer 2 will definitely have a `saveData()` function it can use.

Dependency Injection

Using configuration or programming practices to tell a layer 1 which implementation of layer 2 to use.

Tight Coupling is when one layer depends on the exact implementation of another layer.

Eg:

- Mac OSX is tightly coupled to the Apple Mac hardware.
- A frontend UI that knows how to store data in a database is tightly coupled to that database

Loose Coupling is where a layer only deals with abstractions and well defined interfaces to the next layer

Eg:

- Linux is loosely coupled to lots of hardware
- A frontend UI that knows it needs to call a `saveData()` function in the next layer down, but doesn't know where or how that data is saved.

Loose coupling (dependency injection)

A technique for switching between implementations at compile time / runtime

- Example – you need to access a data lookup service that returns a list of countries.

Tight Coupling == (Mostly) Bad

// Pseudocode

```
class LookupService
    constructor(url)
        this.url = url

    getCountries()
        return (GET url + "/countries")
```

```
class MyBusinessLogic
    // get URL from env variable or other properties
    LookupService lookupSvc new LookupService(env.lookupUrl);

    doCountryLookup()
        List countryList = lookupSvc.getCountries();
        return countryList
```

This is bad because the application can only call `doCountryLookup()` if the `LookupService` can actually call out to an external dependency, and that external dependency is running. This especially might not be the case if the external dependency is being built by another team.

It's also very hard to unit test `doCountryLookup()` as the unit test would depend on that external dependency to be up and running. Unit tests should be entirely self-contained.

Let's look at a better alternative

Dependency Injection (Loose coupling)

- Example – you need to access a data lookup service that returns a list of countries.

Loose Coupling == (Mostly) Good

// Pseudocode

```
interface ILookupService
    getCountries()
```

```
class RealLookupService implements ILookupService
    constructor(url)
        this.url = url

    getCountries()
        return (GET url + "/countries")
```

```
class MockLookupService implements ILookupService
    getCountries()
        return "UK, France, Germany"
```

```
class MyBusinessLogic
    // get URL from env variable or other properties
    ILookupService lookupService

    constructor(ILookupService injectedLookupService)
        this.lookupService = injectedLookupService

    doCountryLookup()
        List countryList = lookupService.getCountries()
        return countryList
```

Now we've got two different implementations of LookupService, both of which will work.

In a unit test we can use:

```
ILookupService mockService = new MockLookupService()
MyBusinessLogic logic = new MyBusinessLogic(mockService)
testResult = logic.doCountryLookup()
```

In live code, we can either create instances of RealLookupService manually, or use a dependency injection framework (eg Spring for Java), or through language features (eg in Python)

Note – there are a few types of dependency injection – the example here is constructor injection. Others include property and method.

Further reading:

- https://python-dependency-injector.ets-labs.org/introduction/di_in_python.html
- <https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/>

Dependencies (Internal and External)

Internal dependencies are dependencies within your application that you have control over.

- Eg: database schema, API layer, security framework.

External dependencies are dependencies that you don't have control over.

- Eg: external APIs, external code libraries, runtime environments

Considerations: **Internal dependencies**

- Different team members may be responsible for maintaining different dependencies, or may be shared across the team.
- Communication about changes is key.
- Timing of breaking changes should be managed by project lead.
- Make backwards compatible changes where possible.

Considerations: **External dependencies**

- External dependencies may be from other teams within Valcon, the customer, external suppliers or open source
- Where possible, only use well versioned external dependencies (nothing worse than an external dependency changing functionality without you knowing). If not possible, ensure there is good communication with the team responsible for the external dependency so that you get alerted about changing functionality.

Considerations: **Generally**

- Where dependencies are being built as part of the project (eg a data lookup service that your application make requests to, but being built by another team, whether Valcon or other) – **ensure the interface is versioned**. This lets you write mock requests against a dummy implementation while you're waiting for the dependency to be built.
- Remember, you are likely to also be a dependency for another team / project.

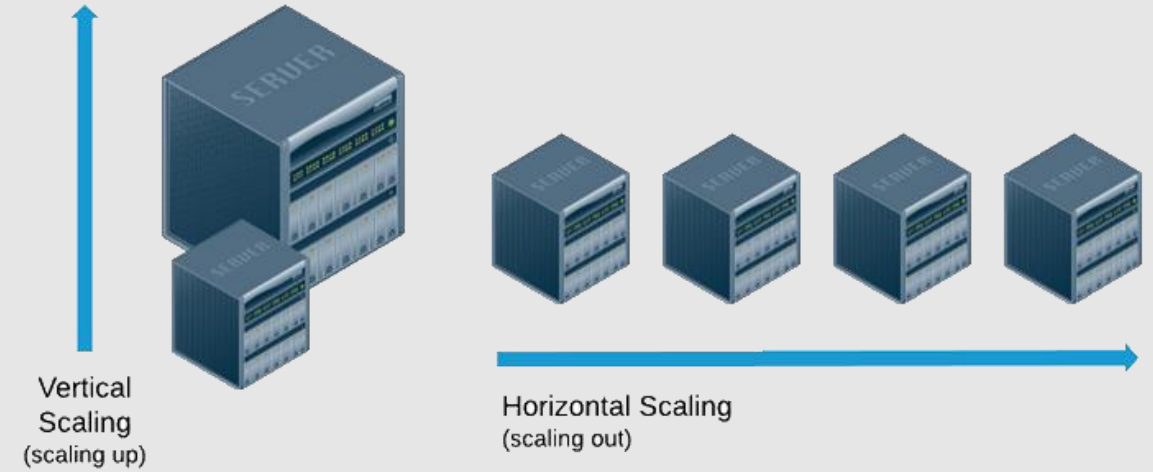
Application patterns

- Horizontal vs Vertical scaling
- Monolith vs Microservice
- Cloud vs On Prem
- Serverless vs Server

Vertical vs Horizontal scaling

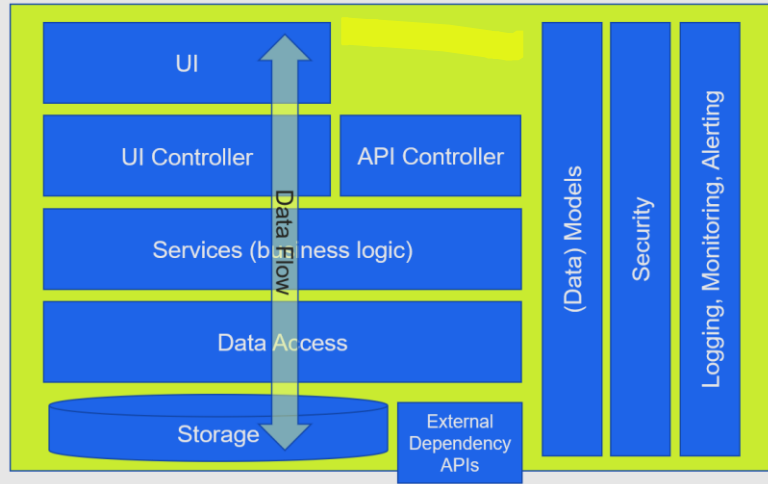
How to handle the load

- Vertical scaling – get a bigger box (scaling up)
- Horizontal Scaling – get more boxes (scaling out)



Monolith vs Microservices

Monolith



Monolith

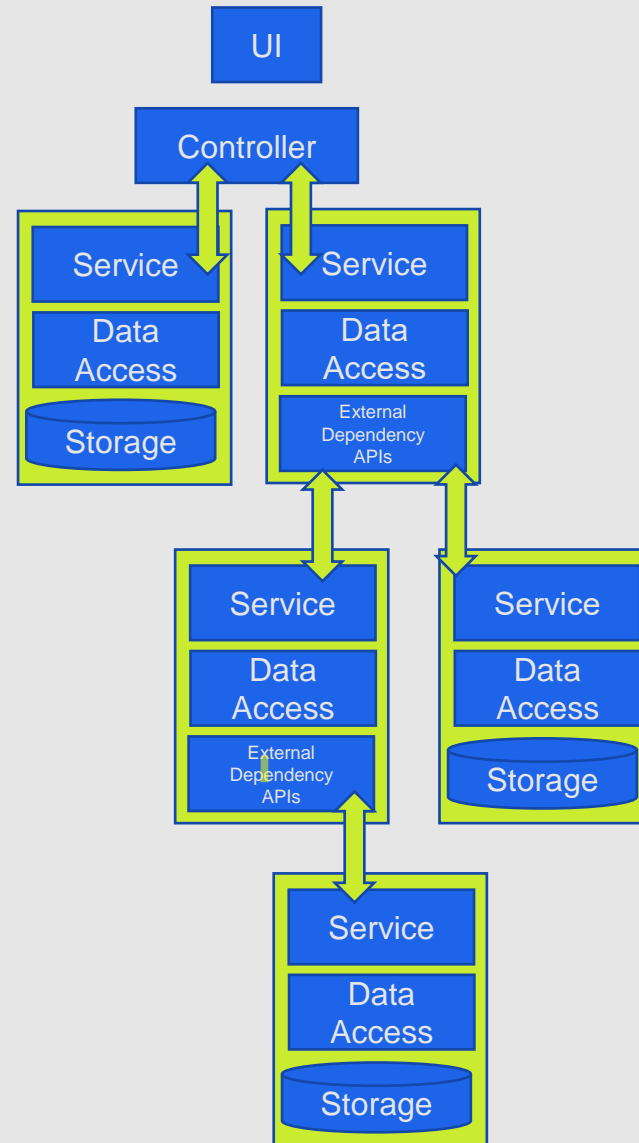
Pros

- Single or few deployment packages (UI + Backend)
- Simple to develop
- Simple to test
- All code shares the same version
- Simple to horizontally scale
- Easy to implement database transactions

Cons

- Large applications can be complex to understand
- Impact of change may impact multiple areas of the application
- Severe bug in one part may bring down the entire application
- Harder to introduce new frameworks and technologies

Microservices



Microservices

Pros

- Reduces complexity by cutting the application into areas of functionality
- Each service can introduce different technologies as long as the interface remains stable
- Each service can be scaled independently
- Each service can be developed independently

Cons

- More deployment and infrastructure complexity
- More testing complexity
- Changes that cover multiple services need to be carefully managed
- May (or may not) separate data storage across multiple persistence locations
- Distributed transactions can be complex if required
- May be constrained by synchronous load to upstream services

As always a balance between the two is likely to be found, with an application core being monolith, but relying on multiple other services

Cloud vs OnPrem (or CloudPrem)

Where to host? - discussion

- My own on-premise physical / virtual machines

- Pros:

- Security – physical access restricted
 - Control
 - Application is portable

- Cons:

- Storage – requires a lot of storage – need to maintain / manage storage
 - Cost, time
 - Not core competency
 - Resilience
 - Need to do patching, sw updates etc
 - Need to order, install

- Virtual Machines, but in the cloud

- Pros:

- Global scale
 - Scale up / down

- Cost effective
 - Instant provision
 - Physical security
 - Application is portable

- Cons:

- Need to do patching, sw updates
 - Configuration, networking etc
 - Cloud training
 - On one VM only one OS at any one time –use containers

- Dedicated Cloud services (Eg AWS S3, Lambda, DynamoDB)

- Pros:

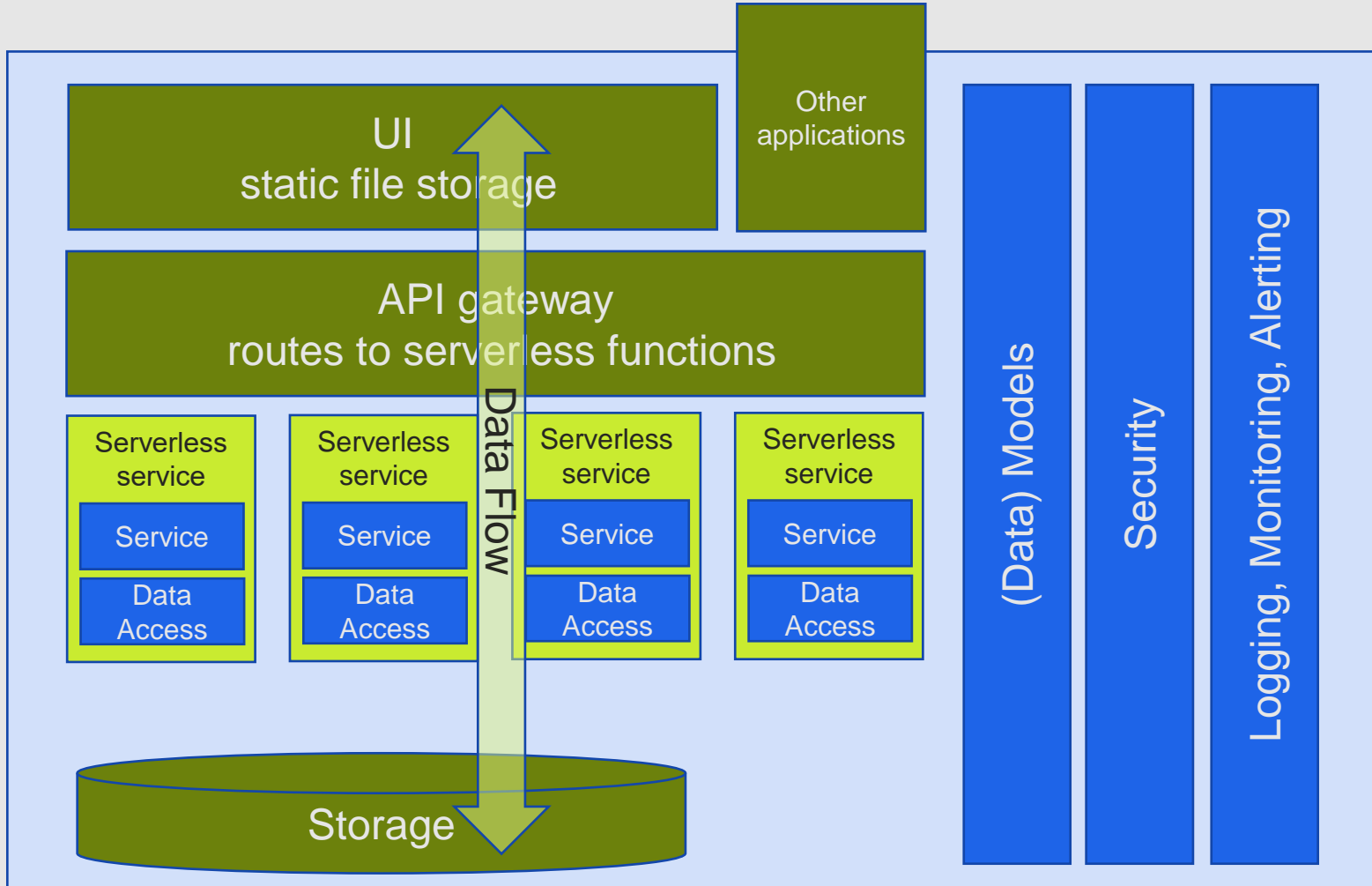
- Global scale
 - Scale up / down
 - Cost effective

- Cons:

- Vendor lock-in – application is not portable between vendors

Serverless vs Server Based

Where to host your application logic



Similar pros/cons to microservices

Serverless architecture can be an implementation of microservices, but not always (could deploy a single serverless service)

Pros

- Reduces complexity by cutting the application into areas of functionality
- Each service can introduce different technologies as long as the interface remains stable
- Each service can be scaled independently
- Each service can be developed independently

Cons

- More deployment and infrastructure complexity
- More testing complexity
- Changes that cover multiple services need to be carefully managed
- May (or may not) separate data storage across multiple persistence locations
- Distributed transactions can be complex if required

Cross-cutting concerns

- Logging Monitoring and Alerting (LMA)
- Secure by design
- Privacy by design

Logging, Monitoring and Alerting (LMA)

Your best friend in production

Logging: Outputting messages describing what the application is doing

Monitoring: Watching the health of the application

Alerting: Being notified when something out of the ordinary occurs

Logging:

- Typically has multiple levels, eg Trace, Debug, Info, Warn, Error. Levels are cumulative left to right (eg, if the application is set to report Info level messages, it will also output Warn and Error messages). Production applications are typically set to Info or Warn levels.
- Output messages that are relevant to enable diagnosis of issues in the application
- Balance output of messages with logging volume. Data storage does cost money!
- Log messages should be easily searchable
- Trace and Debug messages are typically to aid developers when debugging – it's ok to be verbose here, because this level shouldn't be used in production.

Monitoring:

- Memory, CPU, Disk, I/O load
- Server response time
- Server errors (may also monitor Error messages in logs)

Alerting:

- Trigger alerts for investigation when a threshold is breached, eg
 - CPU usage is too high or too low for a period
 - Too many error messages in a period

Log messages and Alerts should be linked to a “runbook” that documents what to do or look for if you see those messages.

Secure by design

Principles:

1. **Minimise attack surface area** – more features mean more attack options
2. **Establish secure defaults** –access should be something granted rather than restricted
3. **The principle of Least privilege** – give only minimum set of privileges
4. **The principle of Defence in depth** – eg captcha, 2fa, monitoring and alerting of unusual activity
5. **Fail securely** – in failure, don't output logging to the user/attacker; don't provide details about the internals of your app
6. **Don't trust services** – always assume external data is bad (whether from a user, or an external service). Validate data received (typically each layer should validate data received – UI validates user input, service layer validates data from the UI etc...)
7. **Separation of duties** – ensure roles and services only have access to perform limited functions
8. **Avoid security by obscurity** – assume everything is visible
9. **Keep security simple** – complexity increases risk of errors
10. **Fix security issues correctly** – determine root cause, test root cause is resolved, look for root cause in other areas of the application

Further Reading:

- OWASP Principles: <https://patchstack.com/security-design-principles-owasp/>
- OWASP Top 10: <https://owasp.org/www-project-top-ten/>
- OWASP Software Security Assurance Module: <https://drive.google.com/file/d/1ZWmk4dpS3zpXjE28wi4cf5Lq6TUjeA5x/view>
- UK Government Security Design Principles: <https://www.ncsc.gov.uk/collection/cyber-security-design-principles>

Privacy by design - principles

- **Proactive not reactive; preventive not remedial**
 - The privacy by design approach is characterized by **proactive rather than reactive** measures. It **anticipates and prevents privacy invasive events** before they happen. Privacy by design does not wait for privacy risks to materialize, nor does it offer remedies for resolving privacy infractions once they have occurred — it **aims to prevent them from occurring**. In short, privacy by design comes before-the-fact, not after.
- **Privacy as the default**
 - Privacy by design seeks to deliver the maximum degree of privacy by **ensuring that personal data are automatically protected** in any given IT system or business practice. **If an individual does nothing, their privacy still remains intact**. No action is required on the part of the individual to protect their privacy — it is built into the system, by default.
- **Privacy embedded into design**
 - Privacy by design is embedded into the design and architecture of IT systems as well as business practices. It is not bolted on as an add-on, after the fact. The result is that **privacy becomes an essential component of the core functionality** being delivered. Privacy is integral to the system without diminishing functionality.
- **Full functionality – positive-sum, not zero-sum**
 - Privacy by design seeks to **accommodate all legitimate interests and objectives** in a positive-sum “win-win” manner, not through a dated, zero-sum approach, where unnecessary trade-offs are made. Privacy by design avoids the pretense of false dichotomies, such as privacy versus security, demonstrating that it is possible to have both.
- **End-to-end security – full lifecycle protection**
 - Privacy by design, having been embedded into the system prior to the first element of information being collected, extends securely throughout the entire lifecycle of the data involved — **strong security measures are essential to privacy**, from start to finish. This ensures that all data are securely retained, and then securely destroyed at the end of the process, in a timely fashion. Thus, privacy by design ensures cradle-to-grave, secure lifecycle management of information, end-to-end.
- **Visibility and transparency – keep it open**
 - Privacy by design seeks to assure all stakeholders that whatever the business practice or technology involved, it is in fact, **operating according to the stated promises and objectives**, subject to independent verification. Its component parts and operations remain visible and transparent, to users and providers alike. Remember, trust but verify.
- **Respect for user privacy – keep it user-centric**
 - Above all, privacy by design requires architects and operators to keep the interests of the individual uppermost by offering such measures as strong privacy defaults, appropriate notice, and empowering user-friendly options. Keep it user-centric.

Further Reading:

- ICO: <https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/accountability-and-governance/data-protection-by-design-and-default/>

Further reading / self study

- Semver
- 12 factor application

Semver – Semantic versioning

<https://semver.org/>

Always use semantic versioning. Eg 1.2.3

- Given a version number MAJOR.MINOR.PATCH, increment the:
 1. MAJOR version (eg 1.x.x) when you make incompatible API changes,
 2. MINOR version (eg x.1.x) when you add functionality in a backwards compatible manner, and
 3. PATCH version (eg x.x.1) when you make backwards compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.
 - eg 1.2.3-SNAPSHOT; 1.2.3-MY-SPECIAL-BUILD, 1.2.3-TESTS etc...

12 Factor Application

<https://12factor.net/>

1. Codebase

- One codebase tracked in revision control, many deploys

2. Dependencies

- Explicitly declare and isolate dependencies

3. Config

- Store config in the environment

4. Backing services

- Treat backing services as attached resources

5. Build, release, run

- Strictly separate build and run stages

6. Processes

- Execute the app as one or more stateless processes

7. Port binding

- Export services via port binding

8. Concurrency

- Scale out via the process model

9. Disposability

- Maximize robustness with fast startup and graceful shutdown

10. Dev/prod parity

- Keep development, staging, and production as similar as possible

11. Logs

- Treat logs as event streams

12. Admin processes

- Run admin/management tasks as one-off processes

Self Study

- Read through the links provided
- Think about the code you've written.
 - Does it have clean layers at the class / package level?
 - Does it separate services from data?
- Attempt to implement a Dependency Injection pattern in Java / Python
 - Next week, we'll get hands on with Java + Spring, and you'll see lots of dependency injection in action