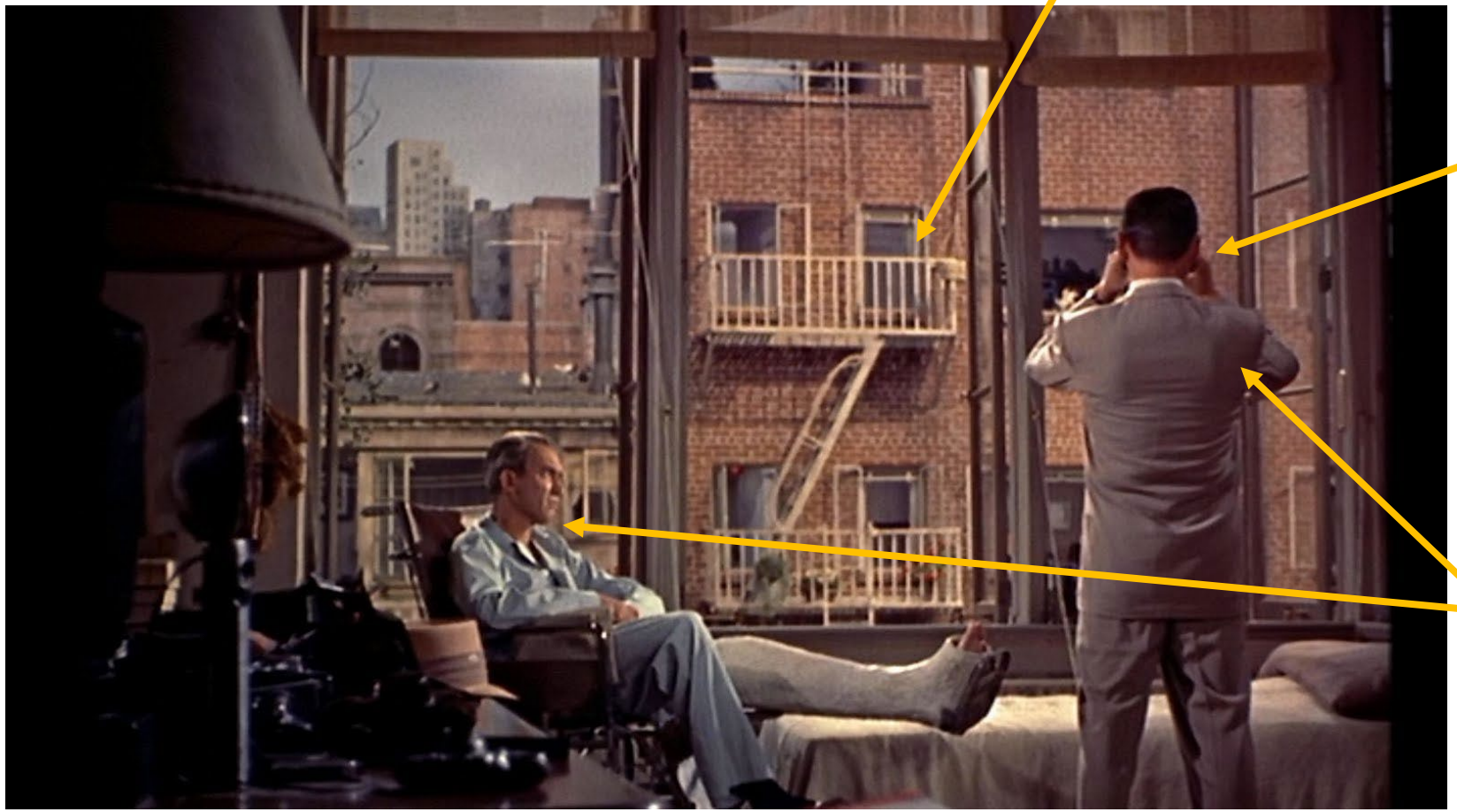
Day 34

Observable

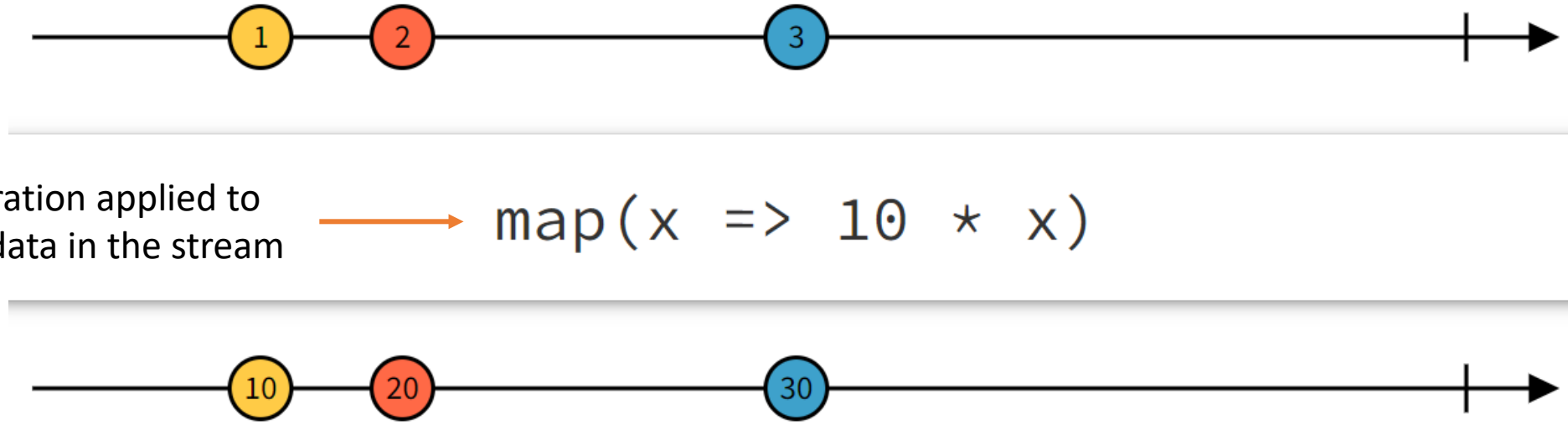


Using a binoculars

Scene from the movie Rear Window

Example of Observable

Observable – a stream of numbers



Operation applied to the data in the stream → `map(x => 10 * x)`

Resultant observable

From https://rxmarbles.com/

Example of using Observable

```
this.form.valueChanges.pipe(
  debounceTime(500),
  map(value => {
    return {
      name: value.name,
      email: value.email
    } as User
  }),
  tap(value => {
    this.newValue = value
  })
)
```

List of operations to be performed on the data stream

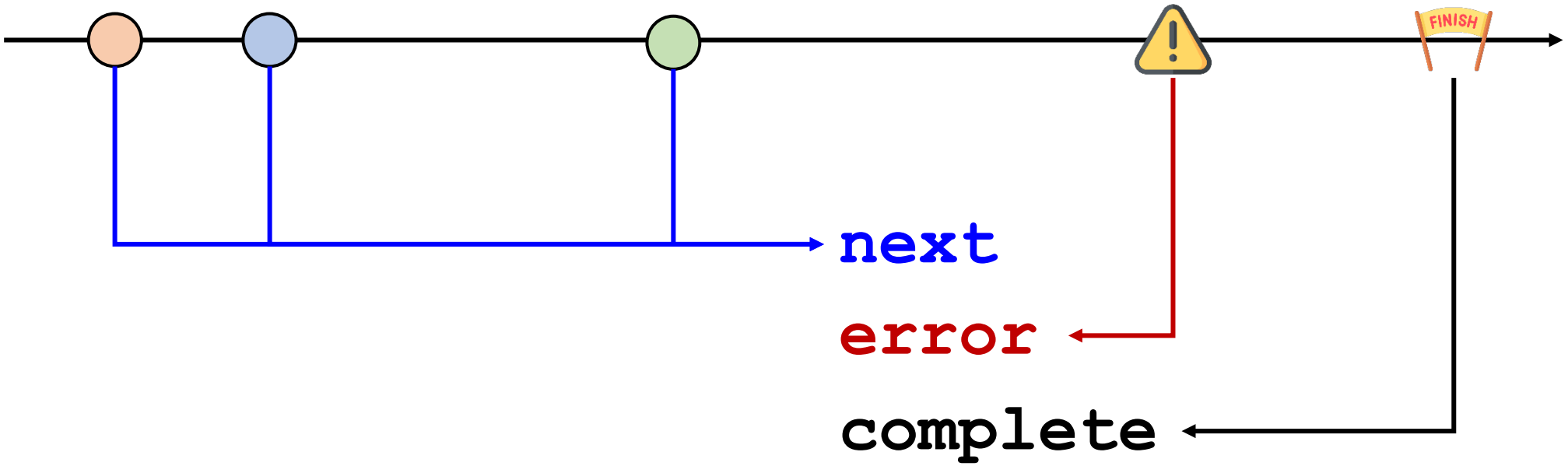
Common Observable Operators

- Observable operators are used inside `pipe()`
  - `filter` – filters data stream
  - `map` – converts a data from one type to another eg "1" to 1
  - `tap` – observe the data stream; used to perform side-effects
  - `take` – take the first n values
  - `takeWhile` – continue taking data until predicate is false
  - `skip`, `skipWhile` – skip first n values
  - `switchMap` – change to a different stream

Subscribing to Observables

```
this.sub$ = this.form.valueChanges. subscribe({
    next: (data) => { console.info(data[0].name) },
    error: (error) => { console.error(error) },
    complete: () => { this.sub$.unsubscribe() }
  })
```

Must unsubscribe when observable ends



Unsubscribing

- Need to unsubscribe if subscription is no longer in use
  - Otherwise can result in memory leak
- Typically unsubscribe occurs in `OnDestroy` callback

```
export class RegistrationComponent implements OnInit, OnDestroy {
   form!: FormGroup
   valueChanges$!: Subscription
   ngOnInit() {
      ...
      this.valueChanges$ = this.form.valueChanges.subscribe(v => { ... })
   }
   ngOnDestroy() {
      this.valueChanges$.unsubscribe()
   }
}
```

Defer and Promise

Promise

Customer will get coffee some time in the near future

Deferred

Proprietor prepares the cup of coffee

Resolved

When proprietor signals customer to collect coffee

Promise

- A promise represents a pending value
- Promises can either be
  - resolved – the value is valid and is available
  - reject – the value is not available
- Once a promise has been resolved, in stays resolved
  - Resolution means either the promise is resolve or reject
  - Cannot reset its state, use only once
- Used in JavaScript
  - Prevent blocking because JavaScript is a single threaded environment
  - To coordinate multiple serial or concurrent tasks

Promise - Provider

- Promise object is native to JavaScript
  - Do not need to import any modules to use it
- Pass the promise a callback with 2 parameters
  - The parameters are the resolve and reject function respective

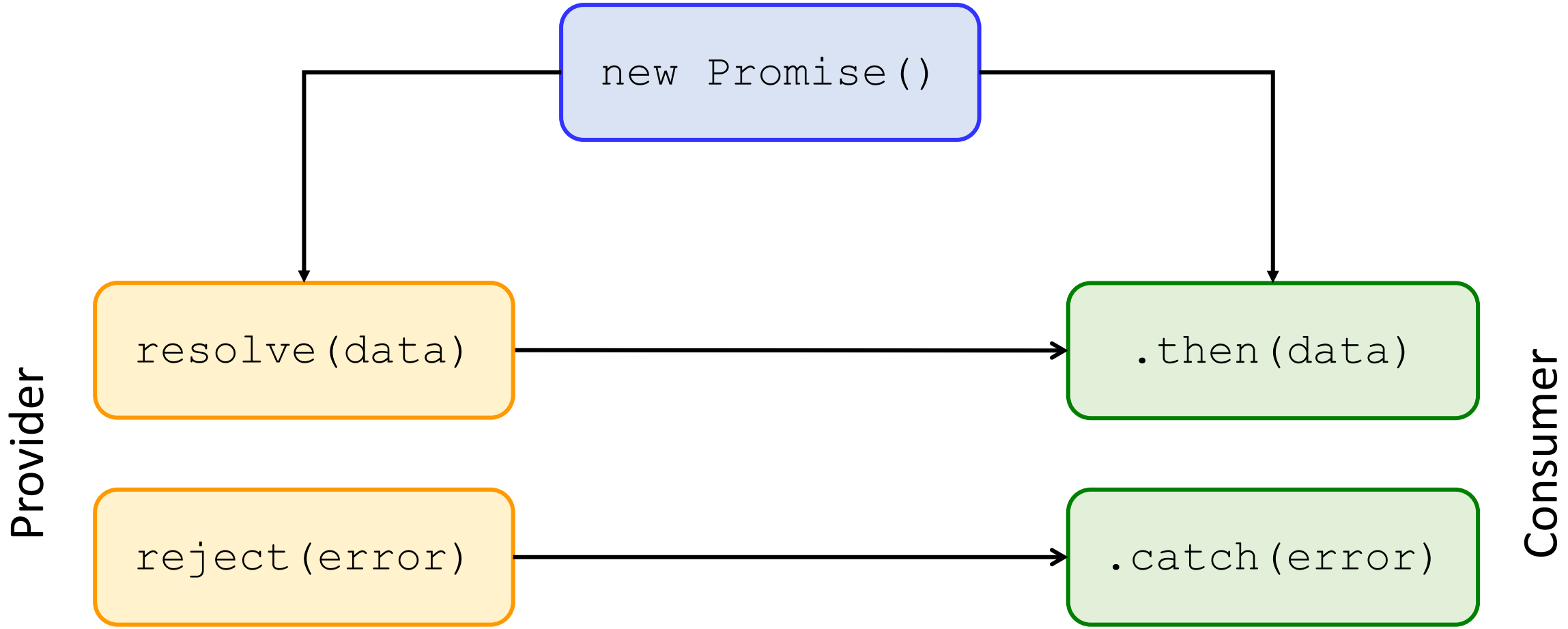
```
const callMe = new Promise((resolve, reject) => {
   //If resolve
   resolve(data);

   //If failed reject
   reject(error);
})
```

Promise - Consumer

- Promise object has 2 functions for listening to resolve and reject
  - Pass a callback
- `then()` for resolve
- `catch()` for reject

```
callMe
  .then((data) => {
    //Promise resolved
  })
  .catch((error) => {
    //Promise rejected
  })
```

Promise


new Promise()
Provider
resolve(data)
reject(error)
.then(data)
.catch(error)
Consumer

Promise Chains

- Any values return from the callbacks of `then()` and `catch()` will be wrapped as promise
- A return from `then()` will resolve to the next `then()`
- Throwing an exception will resolve to the next `catch()`

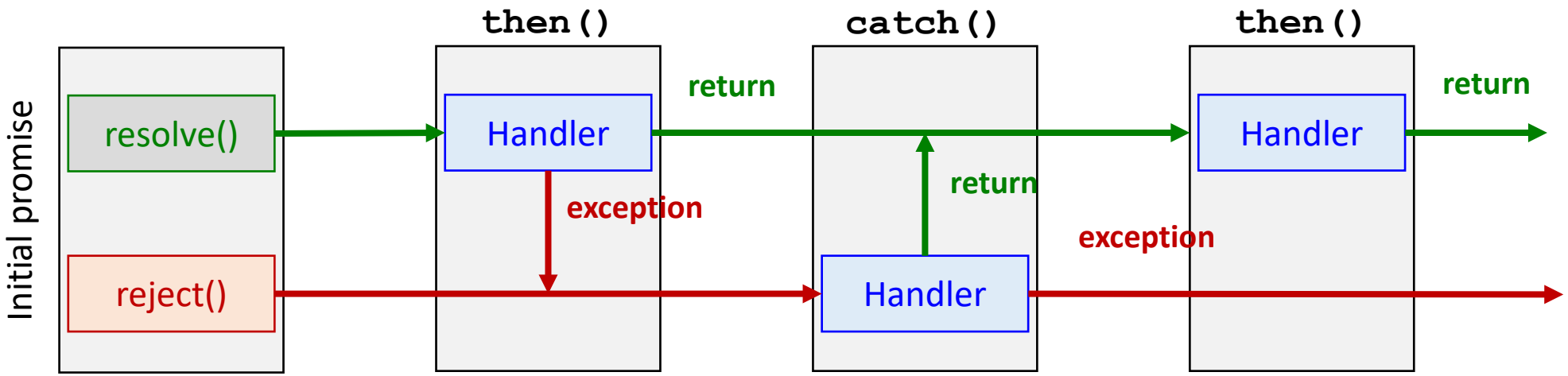


Image inspired from https://speakerdeck.com/danielbrookroberge/asynchronous-pipelines-in-javascript-with-es6-promises

Array, Object, Observable, Promise

Multiple values (*)

Single value (1)

Array

```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)

res.forEach(x =>
  ...
```

Observable

```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)

res.forEach(x =>
  ...
```

Object

```
var y = f(x);
var z = g(y);
```

Promise

```
fAsync(x).then(...);
gAsync(y).then(...);
```

Synchronous

Asynchronous

Method, Resource and Status

Operation	Verb	Noun	Outcome
Read	GET	/customer/1	200 OK
Create	POST	/customer	201 Created
Update	PUT	/customer/1	200 OK
Delete	DELETE	/customer/1	200 OK

REQUEST

RESPONSE

HTTPClientModule

- `HTTPClient` is a service available in the http module
- Need to be installed and imported

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    HttpClientModule
  ]
})
export class AppModule {
```

HttpClient Service

- The HttpClientModule exports the HttpClient service
- Need to be injected into components or services to be used

```
import { HttpClient } from '@angular/common/http';

@Component({ ... })
export class AppComponent {

  constructor(private httpClient: HttpClient) { }
  ..
}
```

Making HTTP Calls

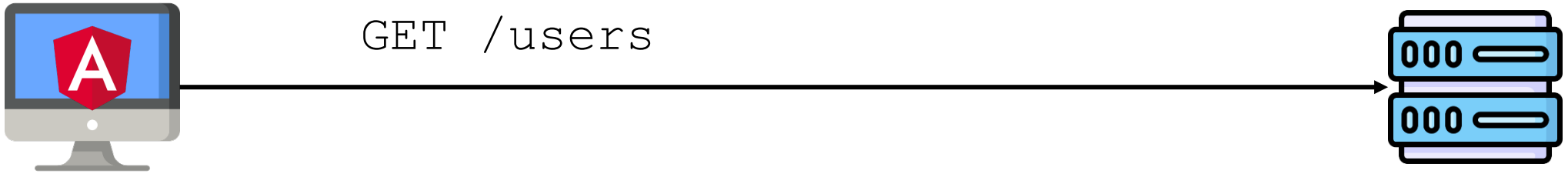
- The **`HttpClient`** is the service for making HTTP request
- **`HttpClient`** provides the following method that maps to its corresponding HTTP method
  - `HttpClient.get(url, configuration)`
  - `HttpClient.post(url, configuration)`
- **`HttpClient`** returns an observable
  - Use `subscribe()` to get the data
  - Or convert to a promise with either `firstValueFrom()` or `lastValueFrom()`
- **`HttpClient`** assumes all request and response payload are JSON

HTTP Request

```
export interface User {
   name: string
   email: string
}
```

Response - an array of the following object

```
{
  name: "fred",
  email: "fred@gmail.com"
}
```



HTTP Method – GET with Observable

```
this.sub$ = this.httpClient.get<User[]>(url)
  .subscribe({
    next: (data) => { console.info(data[0].name) },
    error: (error HttpErrorResponse) => { console.error(error) },
    complete: () => { this.sub$.unsubscribe() }
  })
```

Data type return

Returns an observable

- Subscribe to the observable
  - `next` returns the data from the HTTP request
  - `error` returns the error encountered
    - `HttpErrorResponse` object has the following properties
      - `status` – status code
      - `error` – error message
  - `complete` when the observable closes. Need to unsubscribe from it

HTTP Method – GET with Promise

Converts an observable into a promise

```
lastValueFrom(
    this.httpClient.get<User[]>(url)
)
  .then((data) => { data.name })
  .catch((error: HttpErrorResponse) => { /* error */ });
```

- `firstValueFrom()` and `lastValueFrom()` converts and observable to a promise
  - Promise only returns a single value

Working with Asynchronous Data

```
@Component(…)
export class UsersComponent implements OnInit{

   users: User[] = []

   constructor(private http: HttpClient) { }

   ngOnInit() {
      firstValueFrom(
         this.http.get<User[]>(url)
      ).then(result => this.users = result)
   }
}
```

Wait for the promise to resolve, then assign the value to the member

```
<ul>
   <li *ngFor="let u of users">
      {{ u.name }}
   </li>
</ul>
```

Async Pipe

- `{{ $promise | async }}`
- Asynchronously assign the value of an observable or promise
- If it is an observable, async pipe will automatically unsubscribe from the observable before the component is destroyed
- Used with `*ngIf` and `*ngFor`
  - Angular will update DOM whenever the async pipe returns a value

Async Pipe Example

```
this.data$ = firstValueFrom(this.http.get<User>get(url))

<div *ngIf="data$ | async as u; else loading">
   <user-info [user]="u"></user-info>
<div>

<ng-template #loading>
   <h2>Loading…
</ng-template>
```

Wait for promise to resolve
Assign the resolved value to `u`

```
this.data$ = this.http.get<User[]>get(url)

<ul>
   <li *ngFor="let u of data$ | async>
      <user-info [user]="`"></user-info>
   </li>
</ul>
```

Returns an observable

Wait for promise to resolve before looping
Will unsubscribe automatically

HTTP Method - GET

- Making an invocation with query parameters
  - Create query params with `HttpParams` class

```
HttpParams queryParams = new HttpParams()
   .set("custId", 1234);

this.httpClient.get(url, { params: queryParams } )
  ...
```

Creating a query parameters

Add to the call configuration

`url?custId=1234`

HTTP Method - POST

- **`HttpClient.post`** sends data to as JSON
  - Not as `application/x-www-form-urlencoded`

```
const customer: Customer = {
  name: 'barney',
  email: 'barney@bedrock.com'
}

this.httpClient.post<any>(url, customer)
  ...
```

`customer` as the payload

Angular assumes all content are in JSON

HTTP Method - POST

- **`HttpClient.post`** sends custom headers
  - Not as `application/x-www-form-urlencoded`

```
const jwt = //our token
const headers = new HttpHeaders()
   .set('Authorization', `Bearer ${jwt}`);

this.httpClient.post<any>(url, customer,
   { headers: headers })
  ...
```

Additional headers

Angular assumes all content are in JSON

HTTP Method - POST

- Sending a `x-www-form-urlencoded` payload

```
const customer = new HttpParams()
    .set('name', 'barney')
    .set('email', 'barney@bedrock.com');
```

Construct the playload using `HttpParams` instead of an object

```
const headers = new HttpHeaders()
    .set('Content-Type',
        'application/x-www-form-urlencoded');
```

Set the appropriate content type

```
this.httpClient.post<any>(url,
    customer.toString(),
    { headers: headers })
  ...
```

Call `toString()` to serialize the payload

Services

- Services are abstractions for encapsulating reusable code
  - Like component but has no UI (HTML)
- Service provides cross-cutting concerns
  - “Horizontal” services like authentication, logging, persistence, etc.
- Services are singletons - there is only one instance of the service in the module
  - Provided at the module level
- Services can access other services or components thru dependency injection
  - Eg. `HttpClient` service is available for injection
- Service class must be annotated with `@Injectable()`

Use Cases for Service

- Implement business logic that is independent of any components or services
  - Eg. logging, authentication and access control
- Passing data between components or other services
  - Eg. passing data between 2 peer components, instead of using the parent as a proxy
  - `AddComponent` -(event)-> `AppComponent` -[attribute]-> `CartComponent`
  - `AddComponent` -(event)-> `CartComponent`
- External interactions
  - Eg. making HTTP request

Shared Business Logic

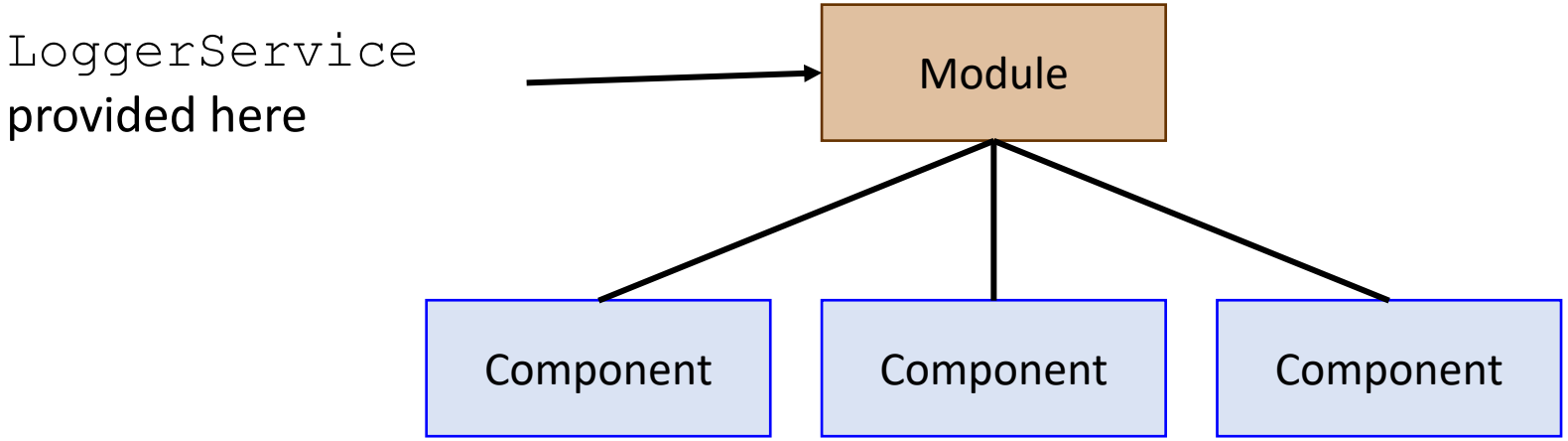
Defining a service

```
@Injectable()
export class Logging {
   constructor() { }

   info(msg: string) {
      console.info(`[new Date()]: ${msg}`)
   }
   error(msg: string) {
      console.error(`[new Date()]: ${msg}`)
   }
}
```

Shared Business Logic

```
import { LoggerService } from './logger.service';

@NgModule({
  ...
  providers: [ LoggerService ]
})
export class AppModule {
  ...
```

All components and services in a module share the same instance of the service if the service is provided at the module level

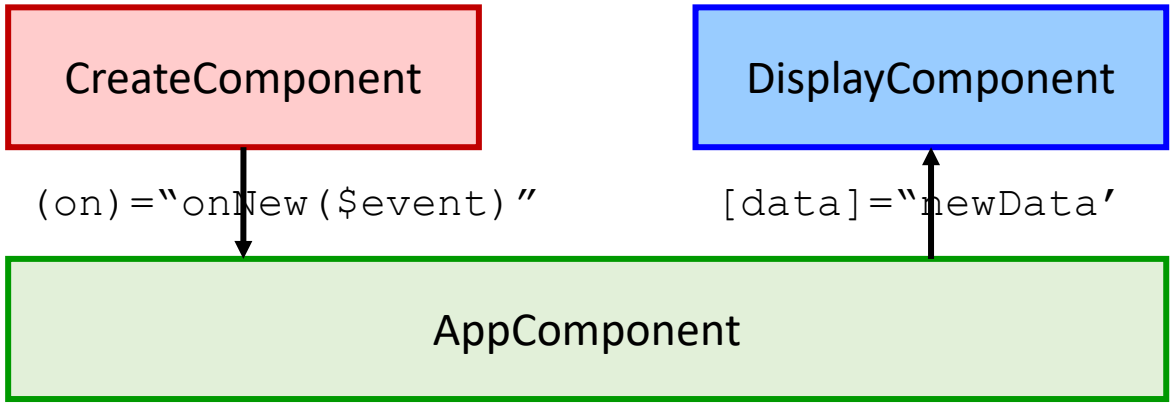


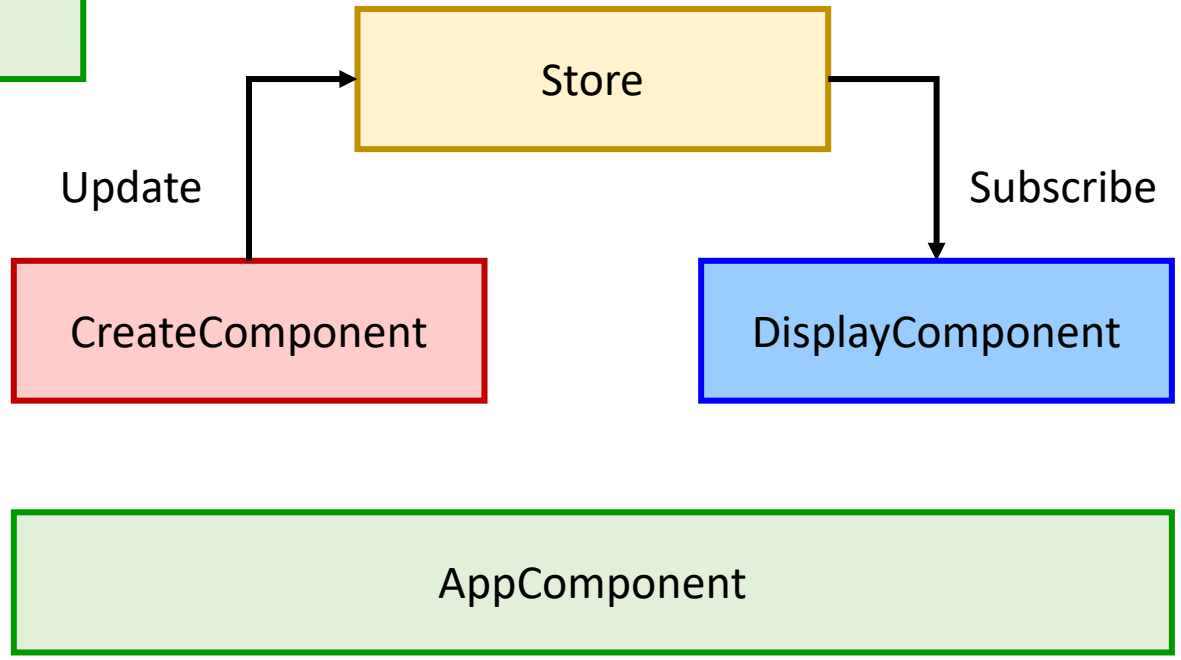
Shared Business Logic

```
import { LoggerService } from './logger.service';

@Component({ ... })
export class AppComponent {

  constructor(private loggerSvc: LoggerService) { }
  ...
}
```

Once a service has been provided, can be injected into any components in the module

Event Binding





Event Binding

Angular performs subscription on the subject when we do an event binding

```
@Output()
onNewRegistration = new Subject<Registration>()
```

```
<app-registration (onNewRegistration)="processNewRegistration($event)">
</app-registration>
```

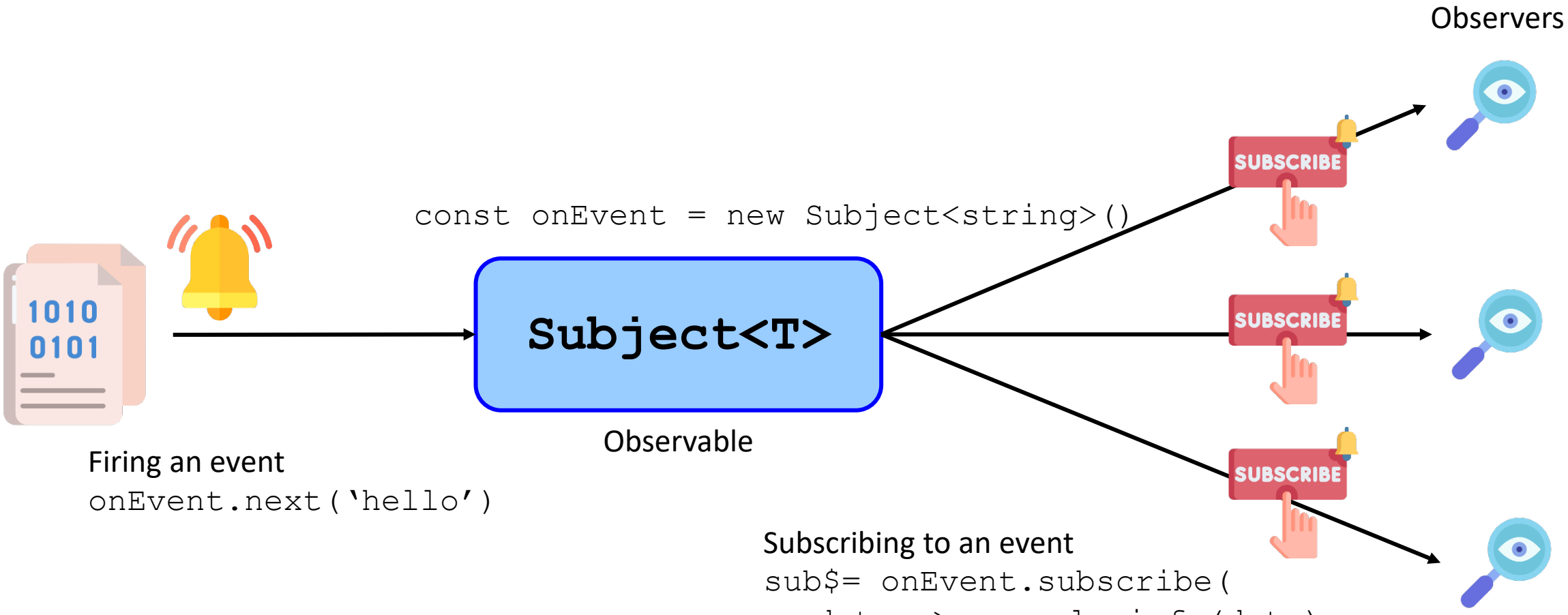
```
this.onNewRegistration.subscribe(
   this.processNewRegistration.bind(this)
)
```

```
this.onNewRegistration.next(newRegistration)
```

When event fires, subject will broadcast the event to all subscriptions

```
processNewRegistration(reg: Registration) {
   //
}
```

Passing Data Between Components



External Interaction

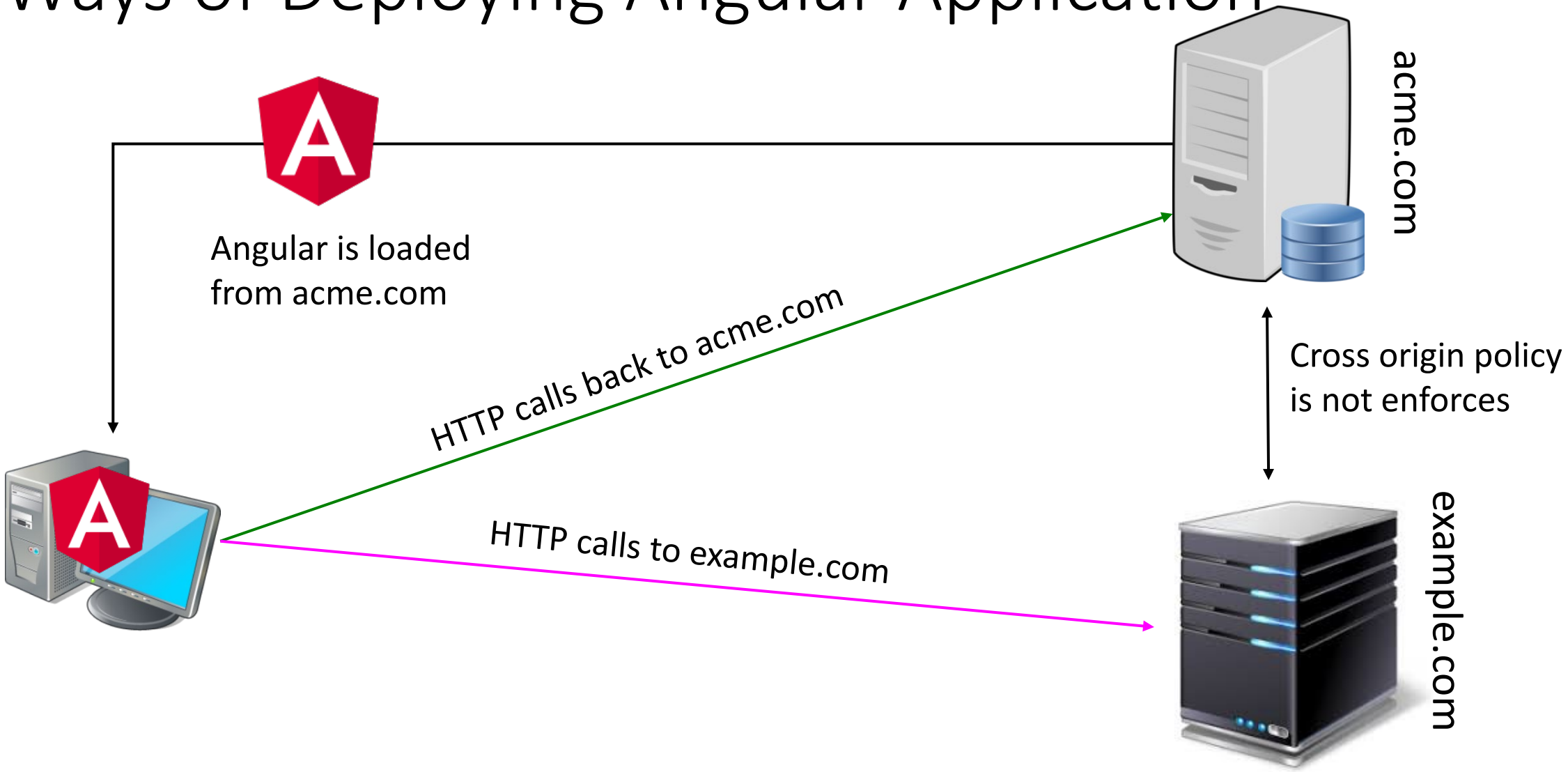
```
@Injectable()
export class WeatherService {
  constructor(private http: HttpClient) { }

  getWeather(city: string, key: string): Promise<Weather> {
    const params = new HttpParams()
        .set('q': city)
        .set('appid': key);
    return (lastValueFrom(
      this.http.get<Weather>(
          'http://api.openweathermap.org/data/2.5/weather',
          { params: params })
    ));
  }
}
```

External Interaction

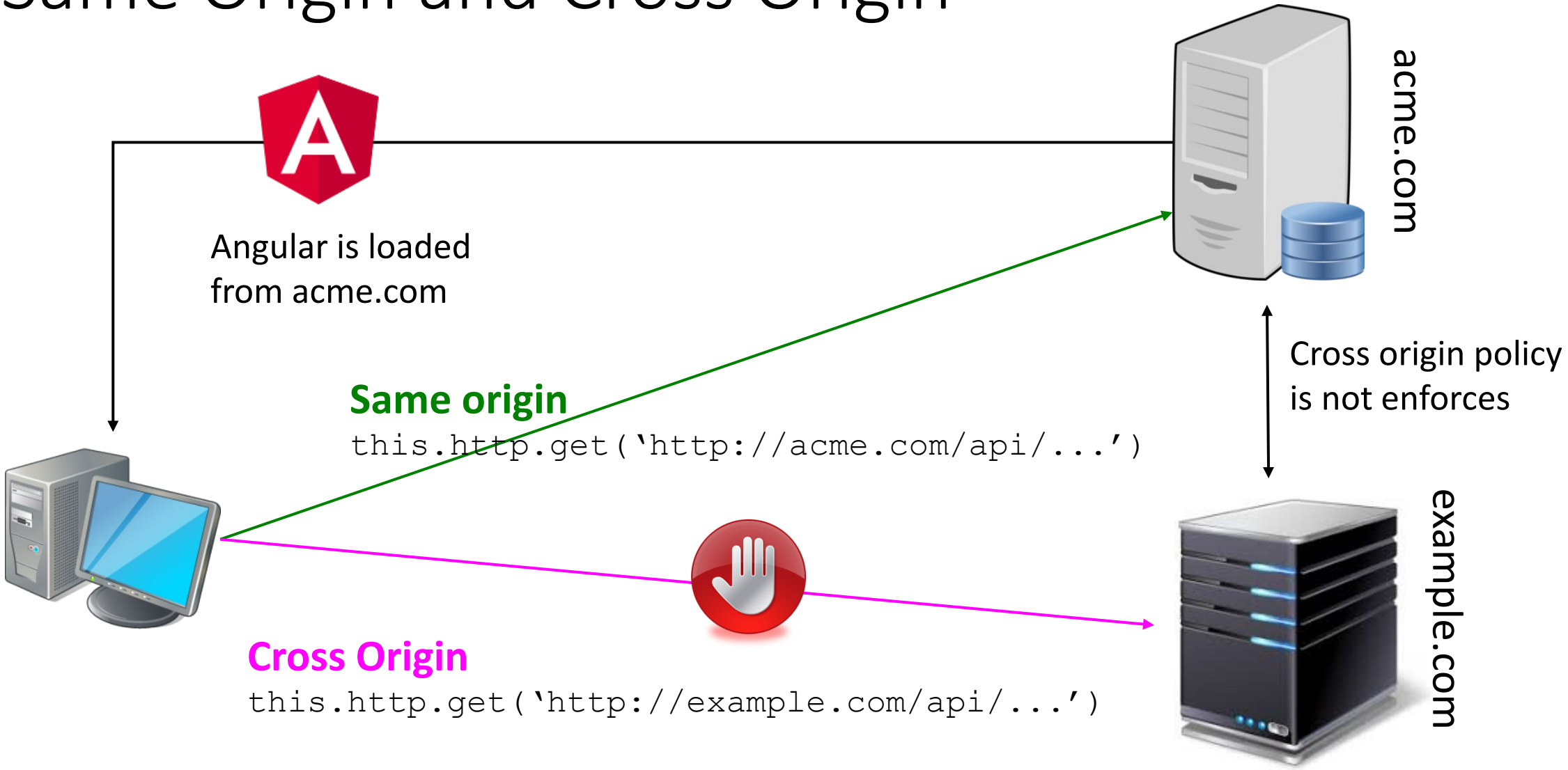
```
import { WeatherService } from ‘./weather.service’;

@Component({ ... })
export class AppComponent implements OnInit {

   weather!: Weather

   constructor(private weatherSvc: WeatherService) { }

   ngOnInit() {
      this.weatherSvc.getWeather(‘Singapore’, ‘abc123’)
         .then(result => this.weather = result)
   }
}
```

Inject external interaction service to where it is needed HTTP details are hidden in the service

Ways of Deploying Angular Application



Same Origin and Cross Origin



Cross Origin Error

- Browser reject cross origin request for certain type of media eg. JSON, XML

```
Elements  Console  Sources  Network  Performance  Memory  Application  »   3 1
top ▼   Filter   Default levels ▼

> const response = await fetch('https://ionicframework.com');
Access to fetch at 'https://ionicframework.com/' from origin 'http://localhost:8100' has been  :8100/app/orders:1
blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an
opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
Uncaught (in promise) TypeError: Failed to fetch                                                :8100/app/orders:1
▶Cross-Origin Read Blocking (CORB) blocked cross-origin response https://ionicframework.com/ with MIME    VM18:1
type text/html. See https://www.chromestatus.com/feature/5629709824032768 for more details.
▶Uncaught (in promise) TypeError: Failed to fetch TypeError: Failed to fetch
>
```

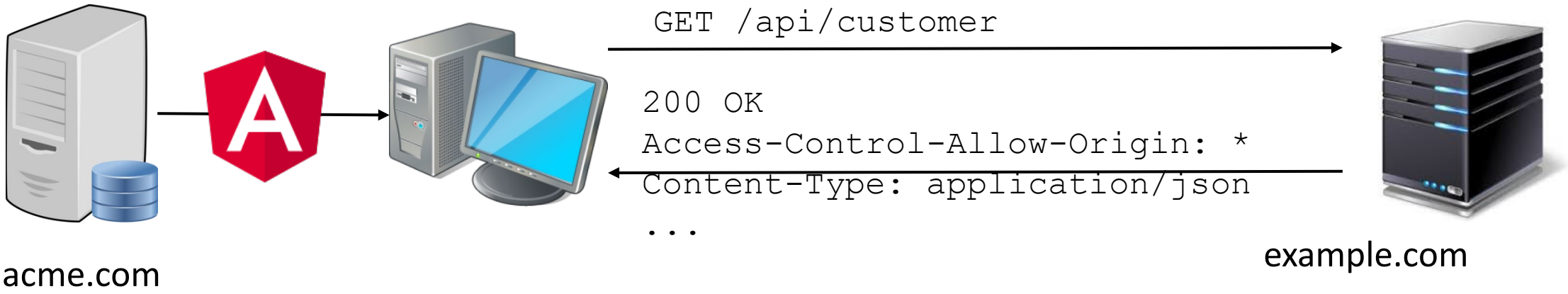
Displayed in Developer Tools

Cross Origin Resource

- Browser will only permit certain types of cross origin resource access
  - GET method
  - Media type include CSS, JavaScript , media (eg images, videos)
  - All other methods and media types are blocked
- Cross origin resource allows clients to make cross origin request
  - Using any method POST, PUT with any media type
- REST servers must opt-in
  - By adding extra headers in the response
- CORS is not enforce if request is from server to server
  - Eg. SpringBoot/Express calling a API endpoint

Setting Angular Development for Cross Origin

- Angular HTTP is making request to a REST endpoint that is hosted on a different origin
  - Different from the one that the Angular application is served from
- The REST endpoint needs to have CORS headers in its response
  - `Access-Control-Allow-Origin` header
  - To indicate if a response can be shared with request from a different origin



Enabling CORS in SpringBoot with Annotations

```
@RestController
@RequestMapping(path="/api/customer")
@CrossOrigin(origins="*")
public class CustomerRESTController {

   @GetMapping(path="{custId}")
   @CrossOrigin(origins="*")
   public ResponseEntity<String> getCustomer(
         @PathVariable String custId) {
      ...
   }
}
```

Annotation can be added to the controller or specific method

Response will include the following header

```
Access-Control-Allow-Origin: *
```

Enabling CORS in SpringBoot Globally

Implement the `WebMvcConfigurer` interface

```
public class EnableCORS implements WebMvcConfigurer {
    final String path;
    final String origins;
    public EnableCORS(String path, String origins) {
        this.path = path;
        this.origins = origins;
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping(path)
            .allowedOrigins(origins)
    }
}
```

Override the `addCorsMappings` method

Configure the resource path and the allowed origins

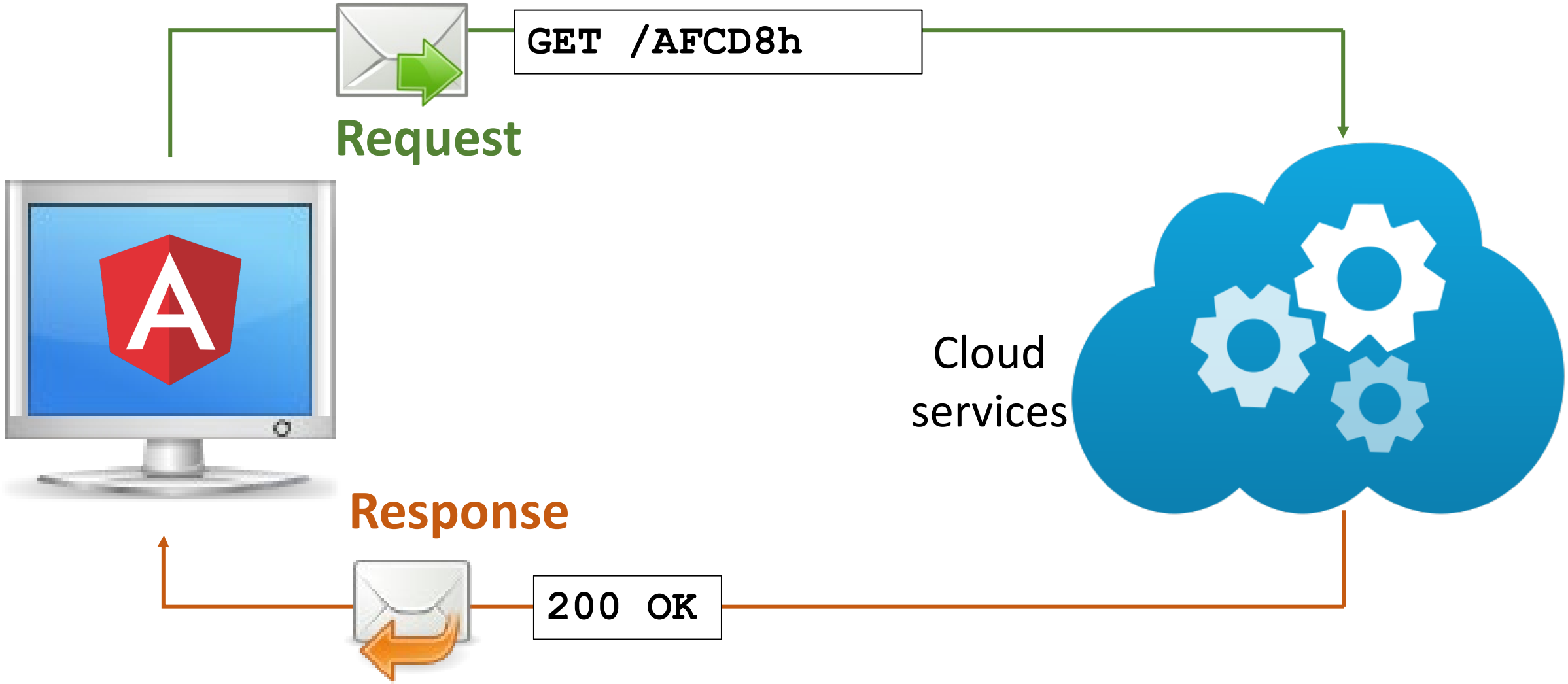
Enabling CORS in SpringBoot Globally

```
@SpringBootApplication
public class CustomerRestApplication {
   public static void main(String[] args) {
      SpringApplication.run(CustomerRestApplication.class, args);
   }

   @Bean
   public WebMvcConfigurer corsConfigurer() {
      return new EnableCORS("/api/*", "*");
   }
}
```

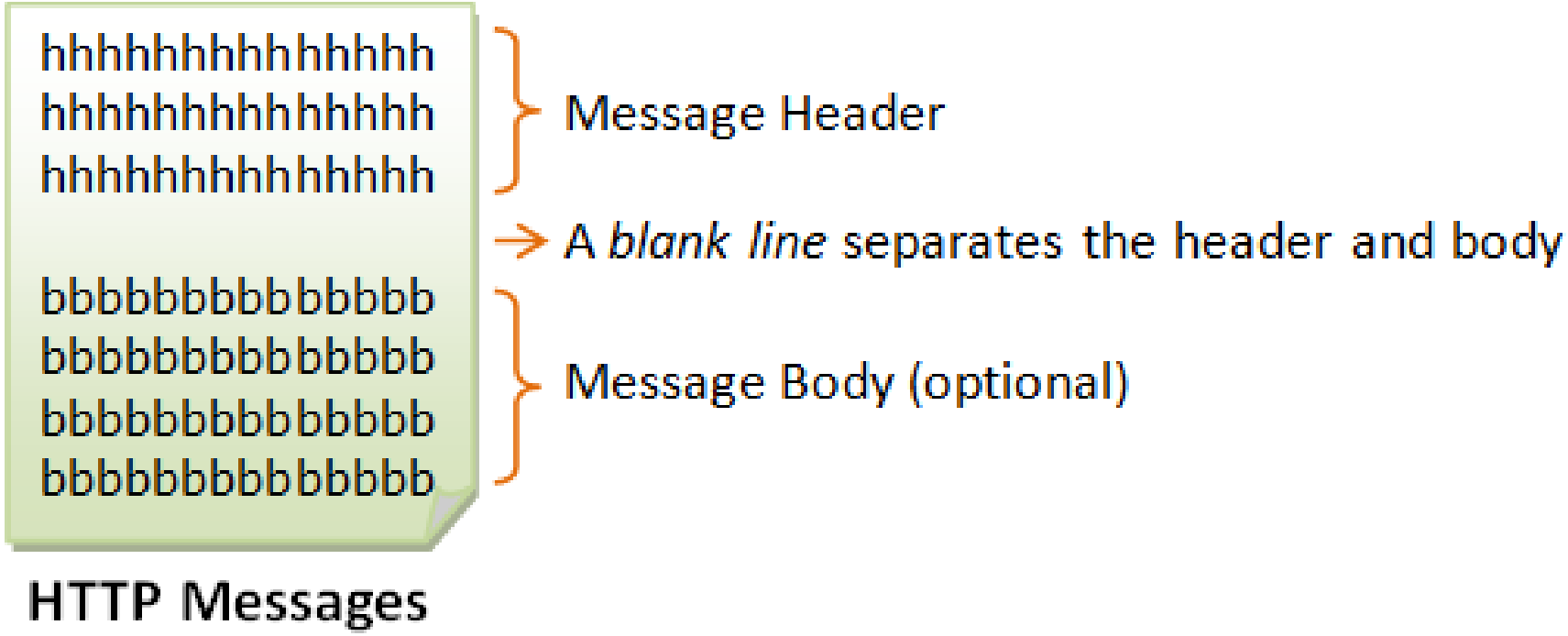
Configure CORS globally by returning the configured CORS configuration
Allow CORS on `/api` for all origins

Unused

HTTP Request

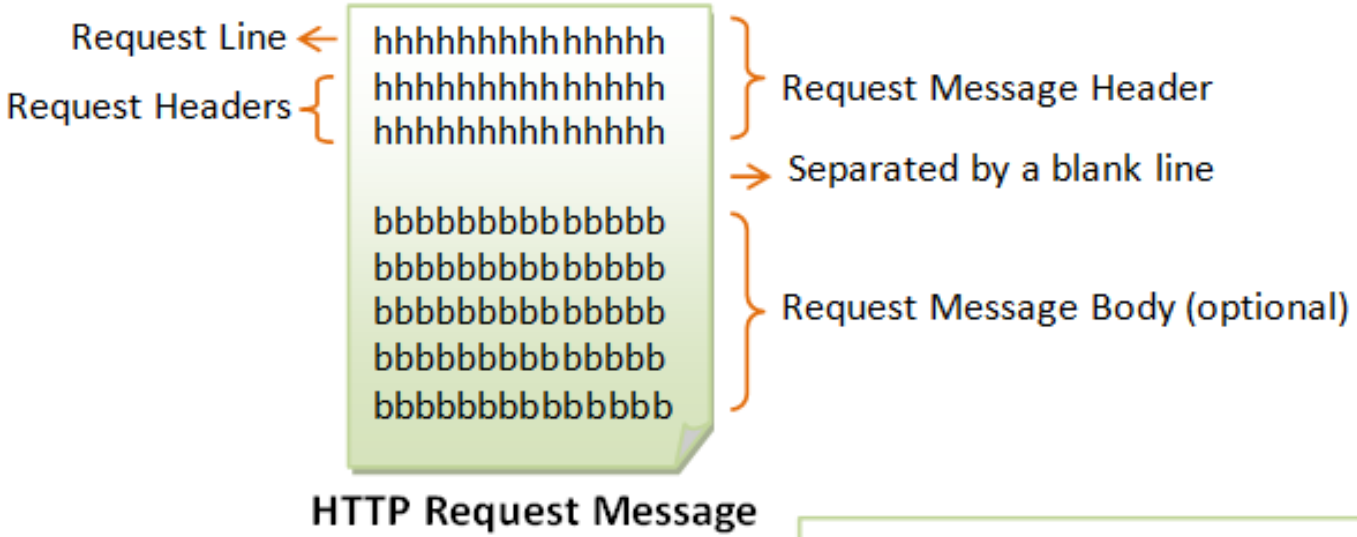


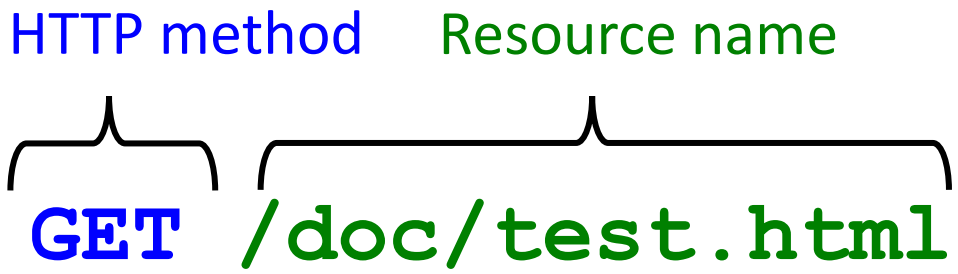
HTTP Message Structure



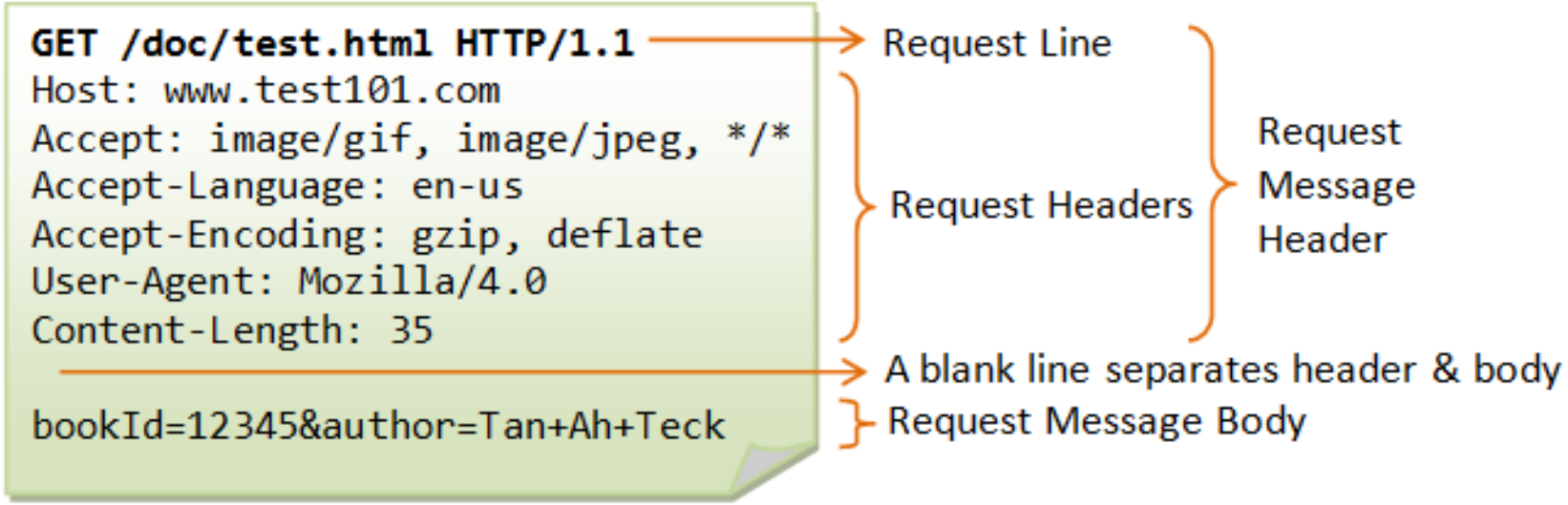
Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

HTTP Request



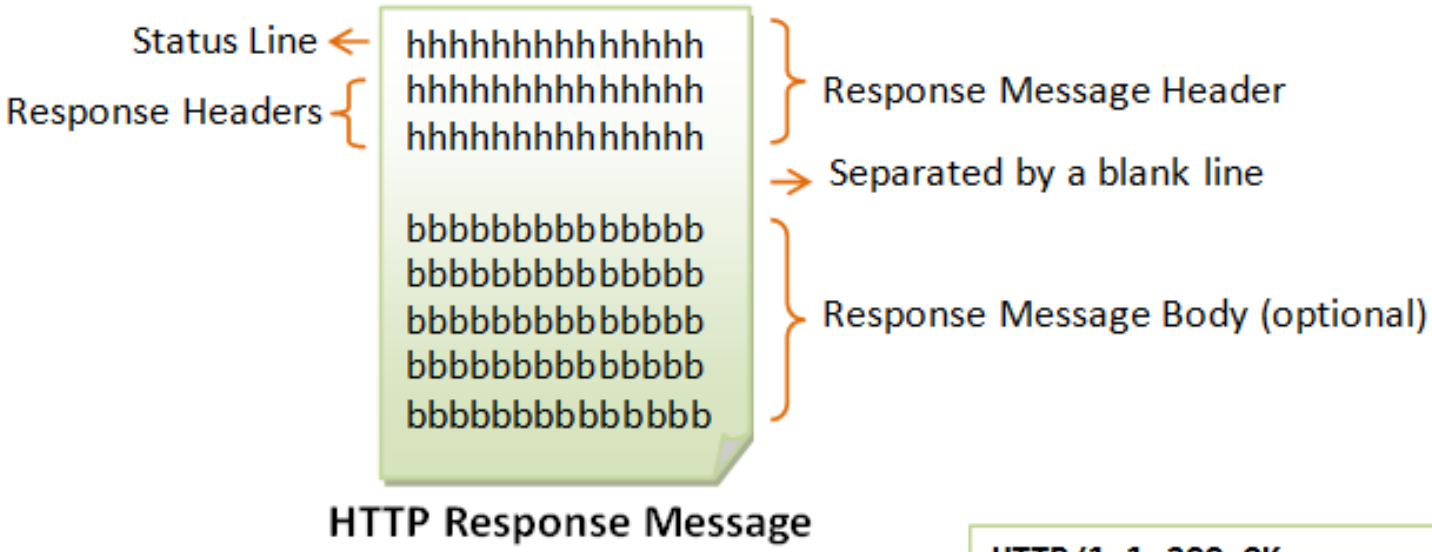


GET
POST
PUT
DELETE
HEAD
OPTION
TRACE



Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

HTTP Response

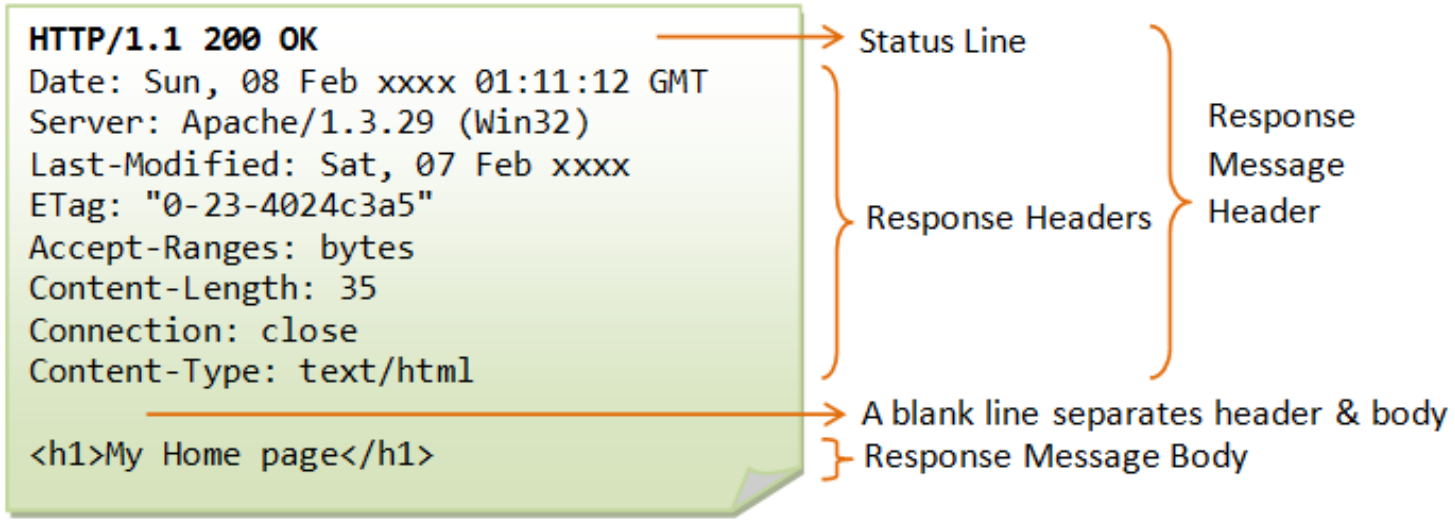


HTTP Response Message

Status code

200 OK

100
200
300
400
500



Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

HTTP Request Structure

A resource name within the service

GET /customer/1

Verb - what to do

Noun - what to do it to