REST and HTTP

Introduction to Angular 6

REST

- REpresentational State Transfer
  - architectural style for distributed hypermedia systems
  - simple way to organize interactions between independent systems with minimal overhead
- Widely used for Web APIS
  - REST is not tied to the web, but it's almost always implemented as such
  - Inspired by HTTP and can be used wherever HTTP can

REST guiding principles

- There are 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful.

- Client–server
  - By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components

- Stateless
  - Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
  - Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state.

REST guiding principles

- Cacheable
  - Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
  - If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

- Uniform interface
  - In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components
  - REST is defined by four interface constraints:
    - identification of resources;
    - manipulation of resources through representations;
    - self-descriptive messages;
    - hypermedia as the engine of application state.

REST guiding principles

- Layered system
  - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior
  - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
  - Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches.

- Code on demand (optional)
  - REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

Stateless principle in REST

- The necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers.
  - The URI uniquely identifies the resource and the body contains the state (or state change) of that resource.
  - Then after the server does it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body.
- A normal session maintains state across multiple HTTP requests
  - In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests.

7.4 REST interface constraints

- Resource-Based
  - Individual resources are identified in requests using URIs as resource identifiers
  - The resources themselves are conceptually separate from the representations that are returned to the client.

- Manipulation of Resources Through Representations
  - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the servers.

- Self-descriptive Messages
  - Each message includes enough information to describe how to process the message.

7.4 REST interface constraints

- Hypermedia as the Engine of Application State (HATEOAS)
  - Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name).
  - Services deliver state to clients via body content, response codes, and response headers.
  - Where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects
  - This is technically referred-to as hypermedia (or hyperlinks within hypertext).

HTTP and REST

- HTTP is not the same as REST
  - however it is the most widely used protocol reference implementation for REST.
- Data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs).
- The clients and servers exchange representations of resources by using suitable HTTP methods.
  - These methods comprise a major portion of our uniform interface constraint.
- The primary HTTP methods are POST, GET, PUT, PATCH, and DELETE.
  - These correspond to create, read, update, and delete (or CRUD) operations, respectively.
-

REST HTTP methods

- GET
  - Read a specific resource (by an identifier) or a collection of resources.
- PUT
  - Update a specific resource (by an identifier) or a collection of resources.
  - PUT can also be used to create a resource in the case where PUT is to a URI that contains the value of a non-existent resource ID
- PATCH
  - Used to modify a specific resource by containing only the changes needed, not the complete new resource.
  - This resembles PUT, but the body contains a set of instructions describing how a resource should be modified to produce a new version.
  - The body is in some kind of patch language like JSON Patch or XML Patch.

REST HTTP methods

- DELETE
  - Remove/delete a specific resource by an identifier.
- POST
  - Create a new resource, particularly subordinate resources
  - When creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent

LAB

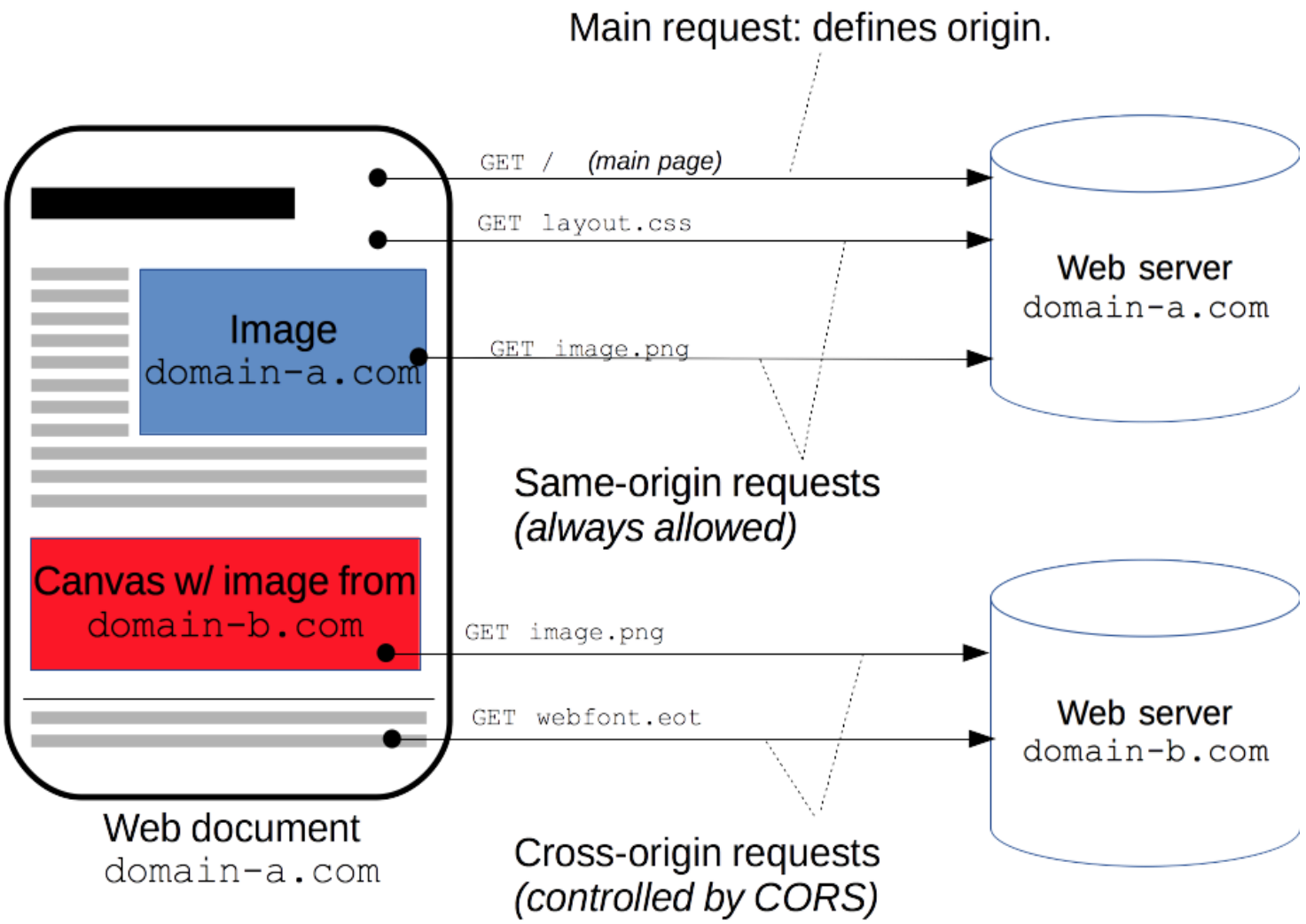
HTTPClient, HTTPService, HTTPResponse

- HTTPService can be injected into the constructor
- The HttpClient APIs directly mirror the HTTP methods
  - can call httpClient.get, httpClient.post, and httpClient.patch directly
  - each of them take the URL as the first argument, and a request body as the second (if the method supports it).
- HttpClient can give you type-assurance across your code.
- With HttpResponse for errors, the response body is available in the error key inside the response

Cross-Origin Resource Sharing

- Mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin.
- A web application makes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.
- An example of a cross-origin request:
  - The frontend JavaScript code for a web application served from http://domain-a.com uses XMLHttpRequest to make a request for http://api.domain-b.com/data.json.

CORS


Main request: defines origin.
GET / (main page)
GET layout.css
GET image.png
Web server
domain-a.com
Image
domain-a.com
Same-origin requests
(always allowed)
Canvas w/ image from
domain-b.com
GET image.png
GET webfont.eot
Web server
domain-b.com
Web document
domain-a.com
Cross-origin requests
(controlled by CORS)

Angular proxy for CORS

- For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts.
  - XMLHttpRequest and the Fetch API follow the same-origin policy
- This means that a web application using those APIs can only request HTTP resources from the same origin the application was loaded from
  - unless the response from the other origin includes the right CORS headers.
- Angular CLI has an option of proxying to support CORS

HTTP API additional options

- One of the common tasks that any developer has with an HTTP API is to send additional query parameters, or certain HTTP headers along with the request.
  - We can add a second argument to the http.get call, which is an options object.
  - There are certain keys we can pass to it, to configure the outgoing HTTP request.
- We can set the outgoing/request HTTP headers. There are two ways we can set both the headers as well as the parameters.
  - Pass an HttpHeaders object, which is a typed class instance, on which we can set the correct headers. It follows a builder pattern, so you can chain multiple headers
  - Pass it a plain-old JavaScript object

HTTP API additional options

- HTTP query parameters can also be configured in two ways:
  - using the typed, built-in HttpParams class
  - using a plain-old JavaScript object.

- The observe parameter takes one of three values:
- Body
  - This is the default value, which ensures that the observable's subscribe gets called with the body of the response. This body is auto-casted to the return type specified when calling the API.
- Response
  - This changes the response type of the HTTP APIs to return the entire HttpResponse instead of just the body.
  - Allows access to the headers of the response and the status code.

HTTP API additional options

- Event
  - This gets triggered on all HttpEvents, which would include the initialization event, as well as the request finished event.
  - This is more useful when we have an API that sends progress events

Interceptors

- Common use case is to be able to hook into all incoming and outgoing requests to be able to either:
  - modify them in flight
  - listen in on all responses to accomplish things like logging, handling authentication in a common manner, etc.
- With the HttpClient API, Angular allows easily defining interceptors
- This conceptually sit between the HttpClient and the server
  - allowing it to transform all outgoing requests, and listen in and transform if necessary all incoming responses before passing them on

HTTPInterceptor

- HttpInterceptor functions like a chain.
- Each interceptor is called with the request, and it is up to the interceptor to decide whether it continues in the chain or not.
  - Each interceptor can also decide to modify the request as well.
  - It can continue the chain by calling the handler provided to the intercept method with a request object.
- If there is only one interceptor, then the handler would simply call the backend with the request object.
  - If there are more, it would proceed to the next interceptor in the chain.

HTTPRequest and HTTPResponse

- HttpRequest (and HttpResponse) instances are immutable.
- We want them to be immutable because there are various Observable operators, some of which might want to retry the request.
  - If request instances were mutable, then on retrying the request through an interceptor chain, the request might be completely different.
- Any changes we make on them result in a new immutable instance.
  - This allows us to ensure that retrying a request through an interceptor chain would result in exactly the same request being made, and not result in unexpected behavior because of mutability

HTTPClient circular dependency

- If the dependency you are pulling in within the interceptor in turn depends on HttpClient
  - This results in a run-time error on circular dependency.
  - This is because the HttpClient underneath requires all the interceptors
  - when we add a service as a dependency that requires HttpClient, we end up with a circular request that breaks our application.
- There are a few approaches we can take:
  - Split apart your service if possible into a data service that does not depend on HttpClient and one that makes server calls using HttpClient. Then you can pull in the first service as a dependency only, which won't cause this circular dependency
  - Do not inject your HttpClient dependency in the constructor, but rather lazily inject it later as and when you need it.

Observables for search

- Search-as-you-type that uses observables will need to address these problems:
  - If you trigger an HTTP call every time someone performs a keypress, then you would end up with a lot of calls, most of which will need to be ignored.
  - Users rarely type correctly in one shot, often having to erase and retype. This will also result in unnecessary duplicate calls.
  - You need to worry about how to deal with out-of-order responses. If the result for the previous query returns after the current one, you need to handle it in your application logic.

Cold vs hot observables

- Angular observables are cold by default.
- An observable is cold if its underlying producer is created and activated during subscription.
  - A new producer is created for each instance of a subscription to the cold observable
  - Cold observables start running upon subscription, i.e., the observable sequence only starts pushing values to the observers when Subscribe is called.
- Hot observables involve a producer that is a created or activated outside of subscription
  - These producers are already publishing values before a subscription is created
  - Subscribing to these observables will share a reference to a common producer

Changing cold to hot

- Sometimes we should make a cold observable behave as a hot observable and have the same stream shared between multiple subscribers.
- Particularly true if a cold observable fetches data from an API
  - you'll want to limit the amount of network requests and share the stream with all subscribers

Switchmap operator for chain conversion

- Converting an observable chain from one type to another is usually the work of the map operator.
- switchMap has the additional capability to cancel old, in-flight subscriptions.
  - This helps to solve our out-of-order response problem in a nice, clean manner.
- This does not necessarily cancel the underlying HTTP request, but simply drops the subscription