Q: What are the ways to communicate between modules of your application using core AngularJS functionality? Name three ways.

Communication can happen:

Using services
Using events
By assigning models on $rootScope
Directly between controllers, using $parent, nextSibling, etc
Directly between controllers, using ControllerAs, or other forms of inheritence
In the community, there are also mentions of less popular methods such as using watches or the URL.

Q: Which means of communication between modules of your application are easily testable?

Using a service is definitely easy to test. Services are injected, and in a test either a real service can be used or it can be mocked.

Events can be tested. In unit testing controllers, they usually are instantiated. For testing events on $rootScope, it must be injected into the test.

Testing $rootScope against the existence of some arbitrary models is testable, but sharing data through $rootScope is not considered a good practice.

For testing direct communication between controllers, the expected results should probably be mocked. Otherwise, controllers would need to be manually instantiated to have the right context.

Q: The most popular e2e testing tool for AngularJS is Protractor. There are also others which rely on similar mechanisms. Describe how e2e testing of AngularJS applications work.

The e2e tests are executed against a running app, that is a fully initialized system. They most often spawn a browser instance and involve the actual input of commands through the user interface. The written code is evaluated by an automation program, such as a Selenium server (webdriver). That program sends commands to a browser instance, then evaluates the visible results and reports back to the user.

The assertions are handled by another library, for Protractor the default is Jasmine. Before Protractor, there was a module called Angular Scenarios, which usually was executed through Karma, and is now deprecated. Should you want to e2e test hybrid apps, you could use another Selenium server, called Appium.

Testing can be handled manually, or it can be delegated to continuous integration servers, either custom or ones provided by Travis, SauceLabs, and Codeship.

Q: This is a simple test written for Protractor, a slightly modified example from Protractor docs:

```
it('should find an element by text input model', function() {
  browser.get('/some-url');

  var login = element(by.model('username'));
  login.clear();
  login.sendKeys('Jane Doe');
  var name = element(by.binding('username'));
  expect(name.getText()).toEqual('Jane Doe');

  // Point A
});
```

Explain if the code is synchronous or asynchronous and how it works.

The code is asynchronous, although it is written in a synchronous manner. What happens under the hood is that all those functions return promises on the control flow. There is even direct access, using "protractor.promise.controlFlow()", and the two methods of the returned object, ".execute()" and ".await()".

Other webdriver libraries, such as wd https://github.com/admc/wd, require the direct use of callbacks or promise chains.

Q: When a scope is terminated, two similar "destroy" events are fired. What are they used for, and why are there two?

he first one is an AngularJS event, "$destroy", and the second one is a jqLite / jQuery event "$destroy". The first one can be used by AngularJS scopes where they are accessible, such as in controllers or link functions.

Consider the two below happening in a directive's postLink function. The AngularJS event:

```
scope.$on('$destroy', function () {
  // handle the destroy, i.e. clean up.
});
```

And

```
element.on('$destroy', function () {
  // respectful jQuery plugins already have this handler.
  // angular.element(document.body).off('someCustomEvent');
});
```

The jqLite / jQuery event is called whenever a node is removed, which may just happen without scope teardown.

Q: How do you reset a "$timeout", and disable a "$watch()"?

The key to both is assigning the result of the function to a variable.

To cleanup the timeout, just ".cancel()" it:

```
var customTimeout = $timeout(function () {
  // arbitrary code
}, 55);

$timeout.cancel(customTimeout);
```

The same applies to "$interval()".

To disable a watch, just call it.

```
// .$watch() returns a deregistration function that we store to a variable
var deregisterWatchFn = $rootScope.$watch('someGloballyAvailableProperty', function (newVal) {
  if (newVal) {
    // we invoke that deregistration function, to disable the watch
    deregisterWatchFn();
    ...
  }
});
```

Q: Name and describe the phases of a directive definition function execution, or describe how directives are instantiated.

The flow is as follows:

First, the "$compile()" function is executed which returns two link functions, preLink and postLink. That function is executed for every directive, starting from parent, then child, then grandchild.

Secondly, two functions are executed for every directive: the controller and the prelink function. The order of execution again starts with the parent element, then child, then grandchild, etc.

The last function postLink is executed in the inverse order. That is, it is first executed for grandchild, then child, then parent.

A great explanation of how directives are handled in AngularJS is available in the

AngularJS Tutorial: Demystifying Custom Directives post on the Toptal blog.

Q: How does interpolation, e.g. “{{ someModel }}”, actually work?

It relies on $interpolation, a service which is called by the compiler. It evaluates text and markup which may contain AngularJS expressions. For every interpolated expression, a “watch()” is set. $interpolation returns a function, which has a single argument, “context”. By calling that function and providing a scope as context, the expressions are “$parse()”d against that scope.

Q: How does the digest phase work?

In a nutshell, on every digest cycle all scope models are compared against their previous values. That is dirty checking. If change is detected, the watches set on that model are fired. Then another digest cycle executes, and so on until all models are stable.

It is probably important to mention that there is no “.$digest()” polling. That means that every time it is being called deliberately. As long as core directives are used, we don’t need to worry, but when external code changes models the digest cycle needs to be called manually. Usually to do that, “.$apply()” or similar is used, and not “.$digest()” directly.

Q: List a few ways to improve performance in an AngularJS app.

The two officially recommended methods for production are disabling debug data and enabling strict DI mode.

The first one can be enabled through the $compileProvider:

```
myApp.config(function ($compileProvider) {
  $compileProvider.debugInfoEnabled(false);
});
```

That tweak disables appending scope to elements, making scopes inaccessible from the console. The second one can be set as a directive:

```
<html ng-app=“myApp” ng-strict-di>
```

The performance gain lies in the fact that the injected modules are annotated explicitly, hence they don’t need to be discovered dynamically.

You don’t need to annotate yourself, just use some automated build tool and library for that.

Two other popular ways are:

Using one-time binding where possible. Those bindings are set, e.g. in “{{ ::someModel }}” interpolations by prefixing the model with two colons. In such a case, no watch is set and the model is ignored during digest.
Making $httpProvider use applyAsync:

```
myApp.config(function ($httpProvider) {
  $httpProvider.useApplyAsync(true);
});
```

… which executes nearby digest calls just once, using a zero timeout.