



DART

Dart Web UI Codelab

March 2013

Dart Web UI Codelab



December 2012

Instructor

Shannon -jj Behrens

Developer Advocate

+Shannon Behrens

jjinux.blogspot.com



Introduction

This codelab will help you build a simple chat application using Web Components and the Dart Web UI package in Dart. Along the way, you will learn:

- How to set up pub
- How to use Web Components
- How to use dynamic templates and two-way data binding (inspired by Model-driven Views)
- How to build an application with multiple Web Components
- Where to get more information about Web Components and the Dart Web UI package
- What to do if you get stuck

Prerequisites

This codelab assumes you have already completed [Bullseye - Your first Dart app - Codelab - Google IO 2012](#) (or a more up-to-date version thereof) and that you still have an up-to-date version of Dart Editor installed. Although the code in this codelab is based on the earlier codelab, you'll be starting at a fresh, new starting point since the project layouts are somewhat different.

Additional materials

This codelab provides easy to follow, step-by-step instructions. However, there is a lot of online material that you can use to really master the subject.

Background material

- [Introduction to Web Components](#) is an easy-to-read document from the W3C.
- [The Web Platform's Cutting Edge](#) is a talk the Chrome team gave at Google IO about Web Components.
- [Dartisans Ep. 16: Dart and Web Components Reloaded](#) is an episode of Dartisans that we did with the Dart Web UI team.

Documentation

- [Dart Web Components](#) is the official documentation from the Dart Web UI team.
- [Tools for Dart Web Components](#) is the official documentation from the Dart Web UI team about the tools they have produced.
- Seth Ladd has a bunch of [blog posts](#) on Web Components that I found particularly helpful.
- [Getting Started with Pub](#) is the official documentation for pub, the package manager for Dart.
- [Package Layout Conventions in Pub](#) shows how applications should be laid out in Dart.

Code

- [Web UI](#) is the package I used to build this sample.
- [TodoMVC for Web UI](#) is the sample I referred to the most when writing this codelab.

Dart Language and Libraries

- [Dart language tour](#)¹
- [Dart library tour](#)²

What to do if you get stuck

See the Troubleshooting section. It's fairly extensive, so don't forget it's there!

Bootstrap: Download Dart Editor

Though you can develop Dart code with any text editor, you'll be more productive using a rich editing experience that understands the structure and syntax of your code. Dart Editor is a completely stripped down version of Eclipse, built from the ground up to help you build Dart apps.

Visit <http://www.dartlang.org/docs/editor/getting-started/> to download Dart Editor for your platform. You will need a JVM installed and working on your machine in order to run the editor.

¹ <http://www.dartlang.org/docs/dart-up-and-running/contents/ch02.html>

² <http://www.dartlang.org/docs/dart-up-and-running/contents/ch03.html>

Step 1: Import and run the chat app

This codelab walks you through building a custom chat application using Web Components and the Web UI package. You will now load this chat app into the editor and learn how to run both client and server Dart apps.

Objectives

In this section, you will download the source code for the codelab and try out the version in `finished`.

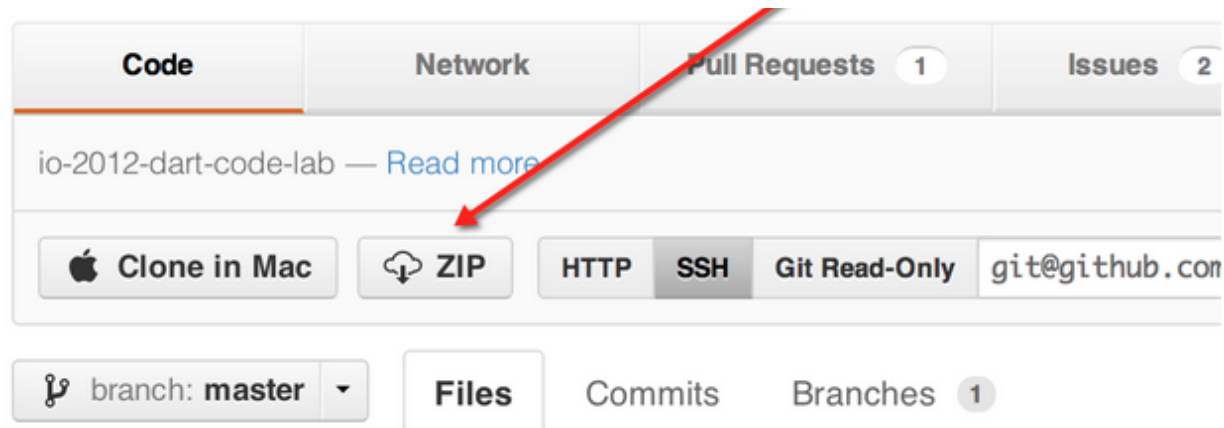
Walkthrough

Download the source code

You can find the source code for this codelab at:

<https://github.com/dart-lang/web-ui-code-lab>

You can use `git clone` to get a copy of the source code, or you can download a zip of the code by clicking the ZIP button.



If you download a ZIP, be sure to uncompress it.

Load the version in `finished` into Dart Editor

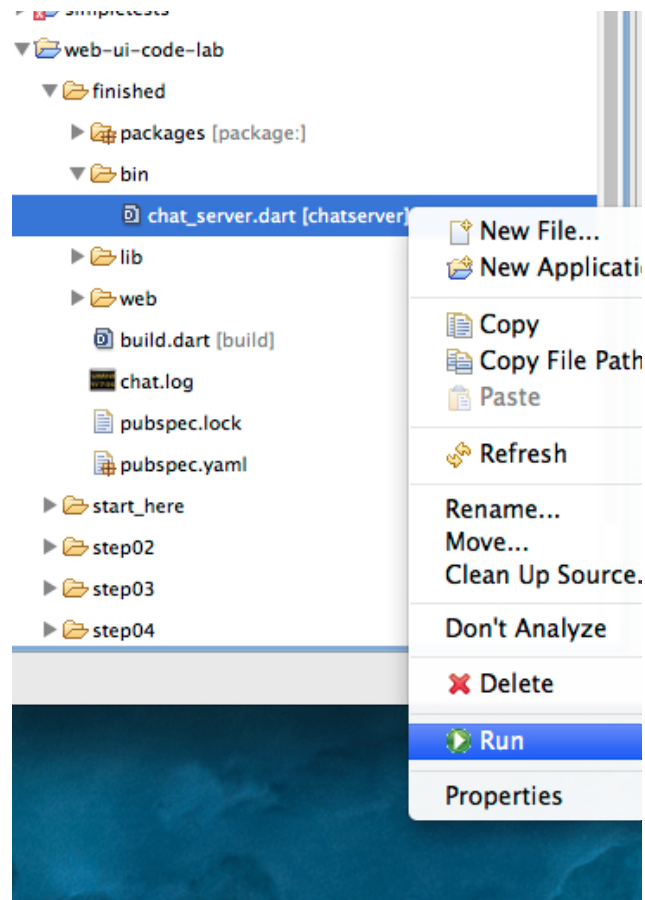
Do not open the entire codelab in Dart Editor. Each subdirectory is its own pub package, so each must be opened separately.

Let's start by opening the version in `finished` in Dart Editor. Select `File > Open Folder...` in the editor. Find the `web-ui-code-lab/finished` directory, select it, and click `Open`.

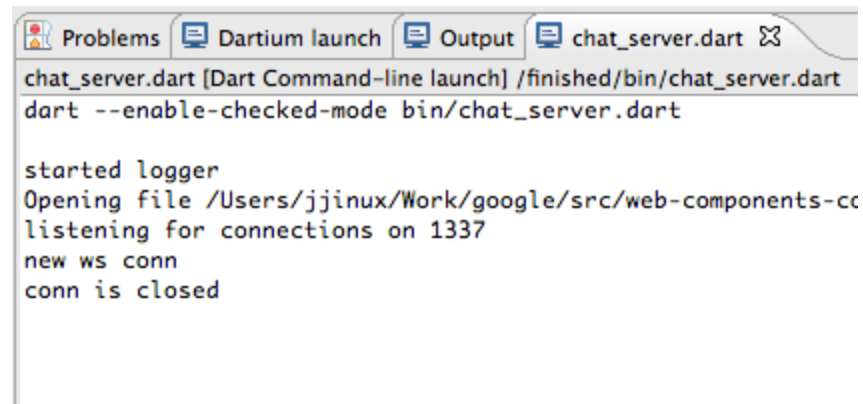
Launch the version of the server in `finished`

The sample chat app has both a client and a server component.

Run the server first. In the Files view on the left hand side of Dart Editor, navigate into the `finished` directory, and select `bin/chat_server.dart`. Right click `chat_server.dart` and select `Run`.



Verify the server is running by checking the `chat_server.dart` console output window at the bottom of your editor. You should see a message: "listening for connections on 1337".

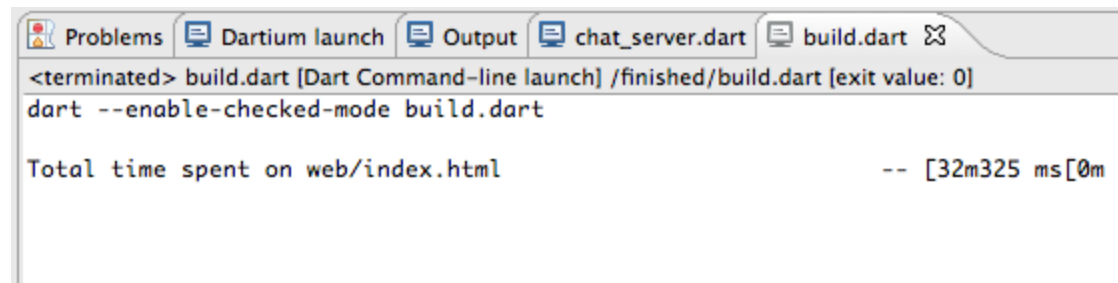


```
chat_server.dart [Dart Command-line launch] /finished/bin/chat_server.dart
dart --enable-checked-mode bin/chat_server.dart

started logger
Opening file /Users/jjinux/Work/google/src/web-components-cc
listening for connections on 1337
new ws conn
conn is closed
```

Building and running the client

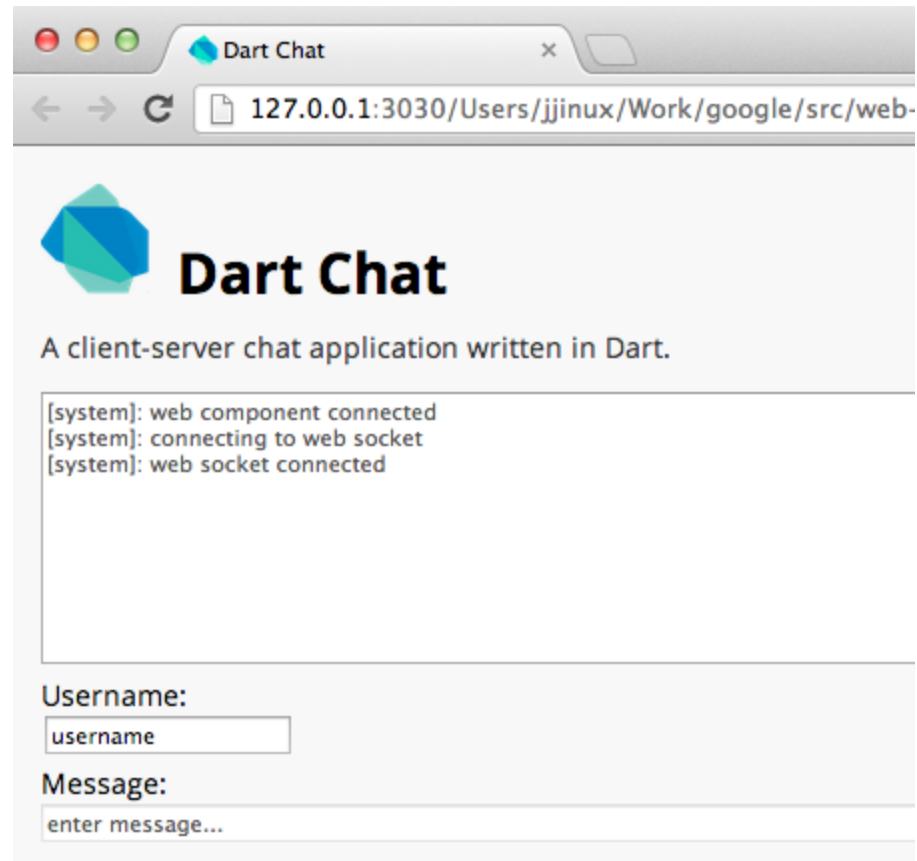
Although Dart does not need to be compiled to run in Dartium, the Dart Web UI package does use a build step in which the Web Component files are split into various Dart files. Right-click on `build.dart` in the `finished` directory and select `Run`. Look at the output of `build.dart` to check that it completed successfully.



```
<terminated> build.dart [Dart Command-line launch] /finished/build.dart [exit value: 0]
dart --enable-checked-mode build.dart

Total time spent on web/index.html -- [32m325 ms[0m
```

If things completed successfully, there should be a new directory named `web/out` containing the generated output. However, you can ignore those files while developing and debugging. Right-click on `web/index.html` and select `Run` in Dartium. If everything goes smoothly, Dartium should start, and you should see the client application.



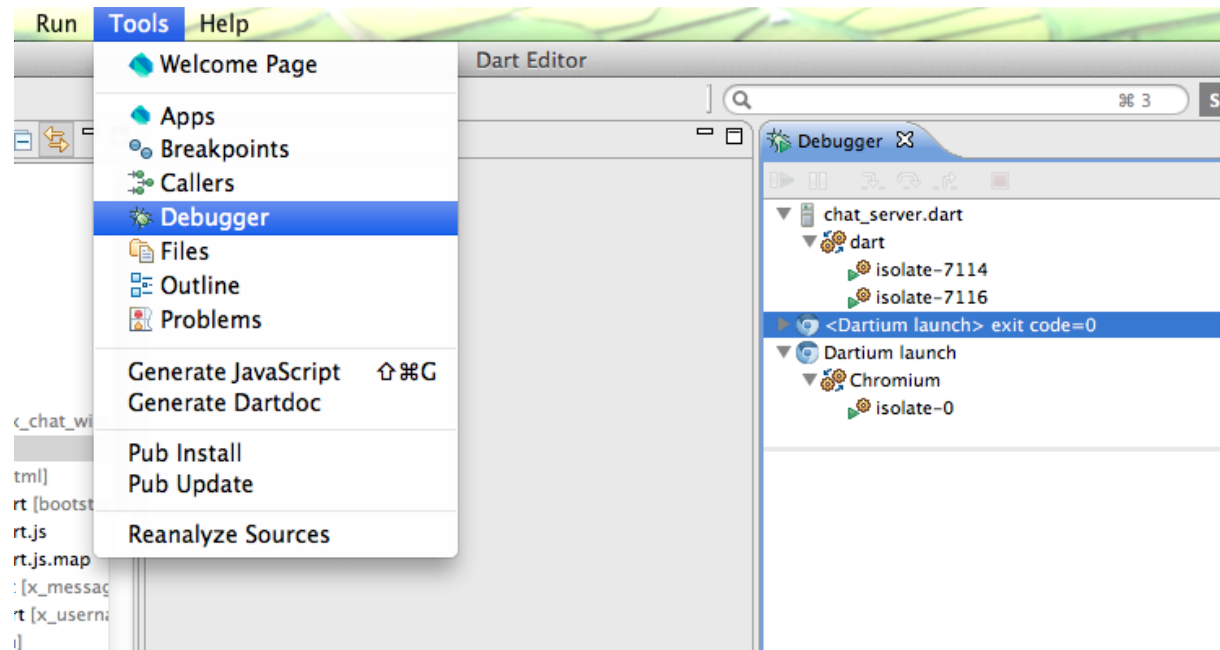
At this point, the client should be able to connect to the server. Type in a username and a message and hit enter. You can duplicate the tab (right click on the tab, duplicate) and have a meaningful conversation with yourself. :)

Run as JavaScript in other browsers

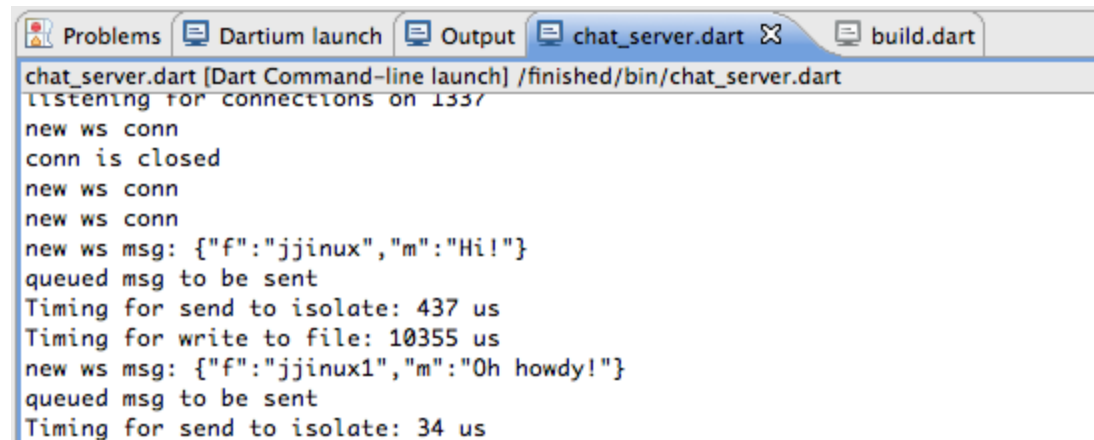
Right click on `web/index.html` and select `Run as JavaScript`. Copy the URL and try it in other browsers such as the stable version of Chrome or Firefox.

Debugger view and console output

Switch back to Dart Editor and select the `Tools > Debugger` in the top level menu. This lists the two processes that you started, the server and the client.



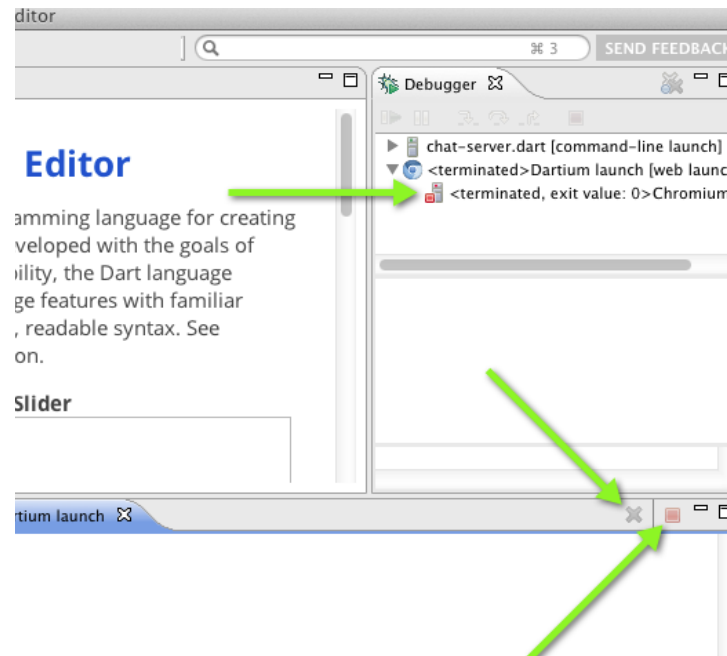
On the bottom of Dart Editor are two views, `chat_server.dart` and `Dartium launch`. Each view has the output from the respective process.



The screenshot shows the Dart Editor's bottom panel with several tabs: Problems, Dartium launch, Output, chat_server.dart, and build.dart. The chat_server.dart tab is active, displaying the following output:

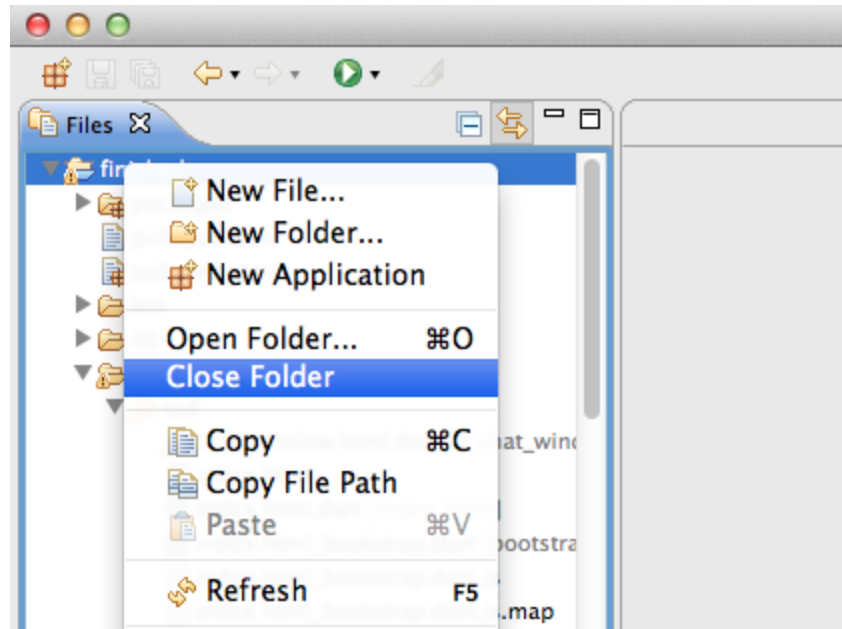
```
chat_server.dart [Dart Command-line launch] /finished/bin/chat_server.dart
listening for connections on 1337
new ws conn
conn is closed
new ws conn
new ws conn
new ws msg: {"f":"jjinux","m":"Hi!"}
queued msg to be sent
Timing for send to isolate: 437 us
Timing for write to file: 10355 us
new ws msg: {"f":"jjinux1","m":"Oh howdy!"}
queued msg to be sent
Timing for send to isolate: 34 us
```

To clear a console's output, click on the gray X icon in upper-right of the console output view. To kill the process, click on the red box in the upper-right of the console output view. After clicking on the red box, you will notice that the Debugger is updated to show that the process was killed.



Stop both processes now, first for the `Dartium launch`, and then `chat_server.dart`.

You should also close the `finished` folder in order to remove it from Dart Editor. Right-click on the `finished` folder, and select `Close Folder`.



Step 2: Getting started with pub

[pub](#) is the package manager for Dart. It is similar to npm in Node.js or RubyGems. A lot of libraries are distributed as pub packages, including the `web_ui` library.

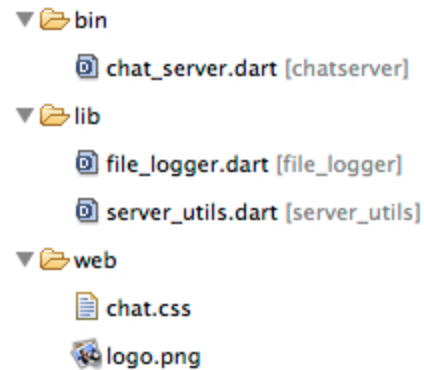
Objectives

We will start with a stripped-down version of the project in the `start_here` directory and add pub support to it in order to install the `web_ui` library.

Walkthrough

In the `web-ui-code-lab` directory, make a copy of the `start_start_here` directory called `mine`. Open up the `mine` folder in Dart Editor. From now on, you'll be working on that. If you get stuck anywhere, you can either refer to one of the directories, such as `step03`, or you can overwrite your version of `mine` with a copy of one of those directories if you need a fresh start.

At this point, the project is pretty bare. It has `bin/chat_server.dart` and its corresponding libraries in `lib`, and there are a few static files in `web`, but that's it.



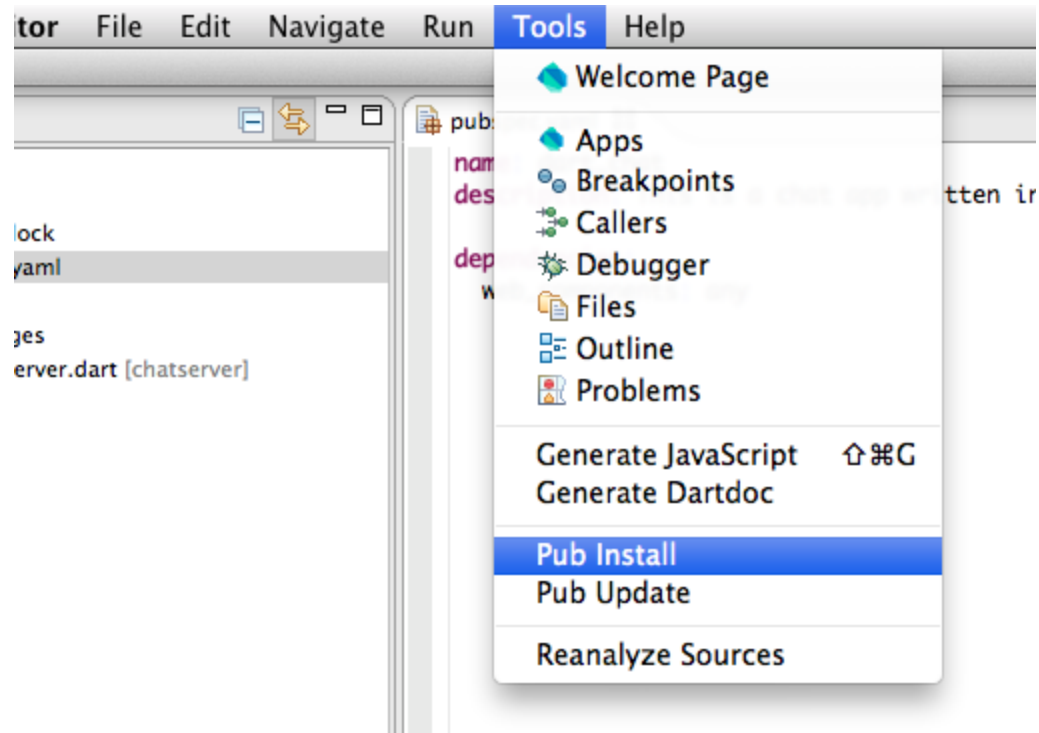
Right-click on the `mine` directory and select `New File...`. Create a new file named `pubspec.yaml`. Put the following in the file:

```
name: dart_chat
description: This is a chat app written in Dart using the Dart Web UI package

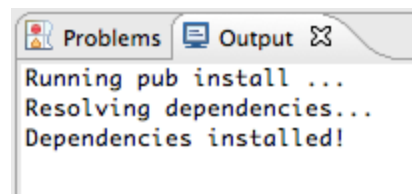
dependencies:
  web_ui: any
  browser: any
```

The "any" keyword asks pub to find and install the latest version of the package.

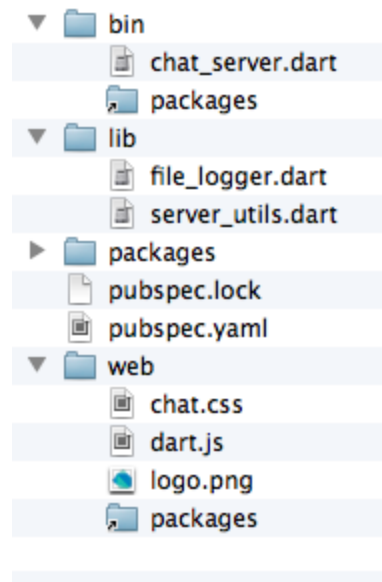
Save the file. Now, run `Tools > Pub Install`.



This will result in the following output:



It will also result in some new files and directories:



`pub` isn't just used to install and manage third-party libraries. It also makes it easier to refer to your own libraries when they are located in the `lib` directory. Open up `bin/chat_server.dart`, and change:

```
import '../lib/file_logger.dart' as log;
import '../lib/server_utils.dart';
```

to:

```
import 'package:dart_chat/file_logger.dart' as log;
import 'package:dart_chat/server_utils.dart';
```

It's convenient to use package-based URLs rather than relative URLs (when doing so is possible) just in case the file you're working on moves to a new directory.

Step 3: Your first Web Component

Web Components are stored in `.html` files. They contain HTML markup and, in the case of Dart, a link to Dart code. A build step is used to translate these `.html` files into `.dart` files (which can be translated into JavaScript in a second step).

Objectives

In this step, we'll build a basic Web Component. We'll also create a very simple `Application` class and a `build.dart` file that Dart Editor will use to build the project.

Code

If you need to catch up, you can make a copy of `step02` named `mine` for this portion of the codelab.

Walkthrough

Create `build.dart`

Right-click on `mine` and select `New File...` Name the file `build.dart`. This file must live at the root of your project.

Put the following in the file:

```
import 'package:web_ui/component_build.dart';
import 'dart:io';

void main() {
  build(new Options().arguments, ['web/index.html']);
}
```

Aside from the `['web/index.html']` part, this code is fairly boilerplate.

When Dart Editor detects one or more files have changed, it runs `build.dart`.

Create index.html

Now, create a file named `web/index.html` with the following:

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title>Dart Chat</title>
    <link rel="stylesheet" href="chat.css">
    <link rel="components" href="chat_window.html">
  </head>
  <body>
    <h1>Dart Chat</h1>

    <p>A client-server chat application written in Dart.</p>

    <x-chat-window id="chat-window"></x-chat-window>

    <script type="application/dart">
      import 'application.dart' as app;

      main() {
        app.init().then((_) => print("app ready")).catchError(print);
      }
    </script>
    <script type="text/javascript">
```

```
src="packages/browser/dart.js"></script>
</body>
</html>
```

There's a lot in this file, so let me point out a few things:

`<link rel="components" href="chat_window.html">` links to the Web Component.

`<x-chat-window id="chat-window"></x-chat-window>` is an example of using a Web Component. It is a custom element named `x-chat-window`.

`<script type="application/dart">...main() {...}...</script>` is the main for the application as a whole. You must have a main, even if it's empty. In this code, we're just calling `app.init()` which we'll create in just a minute. When the app is initialized, the callback registered with `then()` is called. If there is any error during initialization, the callback registered with `catchError()` is called.

Under the Covers: `init()` returns a `Future`, which is an object that represents a value to be returned in the future. Futures are a great way to compose asynchronous methods.

`<script type="text/javascript" src="packages/browser/dart.js"></script>` turns on the Dart VM if it exists, or loads the JavaScript equivalent code if the Dart VM does not exist. Use the `dart.js` file to allow your app to run in both Dartium and browsers without the Dart VM.

Create application.dart

Now, create `web/application.dart` with the following:

```
library application;

import 'dart:html';
import 'dart:async';
```

```

import 'package:web_ui/web_ui.dart';
import 'chat_window.dart';

@observable ChatWindowComponent chatWindow;

init() {
  // The Web Components aren't ready immediately in
  // index.html's main.
  return new Future.delayed(0, () {

    // xtag is how you get to the Dart object.
    chatWindow = query("#chat-window").xtag;
    chatWindow.displayNotice("web component connected");
  })
  .catchError((e) => print("Problem initing app: $e"));
}

```

The `application` library is how different Web Components can get a reference to one another. We haven't even built our first Web Component yet, so this is a bit overkill, but it'll come in handy later.

You should be getting an error similar to:

```

Cannot find referenced source: chat_window.dart.

```

Don't worry, we'll fix that in just a second.

So, what did you just type?

```

library application - this file is a library, named application.

```

```

import 'dart:async', etc - import other libraries.

```

`import 'package:web_ui/web_ui.dart'` - import a library that comes for a package.

`@observable` - a metadata annotation, intended to add hints to tools. In this case, the `chatWindow` variable is observable, which means the Web UI infrastructure should watch it for changes.

`new Future.delayed(0, () { ... })` - run a callback function on the next event loop tick. That is, delay the running of the callback until later. In this case, later is defined as 0 seconds, or "as soon as possible, but not now".

`chatWindow = query("#chat-window").xtag` - find an element in the DOM by ID, and use `xtag` to get the Dart object backing that element.

`.catchError((e) => print("Problem initing app: $e"))` - catch errors thrown from the callback run by the future.

Create a custom element

Now, create `web/chat_window.html` with the following contents:

```
<!DOCTYPE html>

<html><body>
  <element name="x-chat-window"
    constructor="ChatWindowComponent" extends="div">
    <template>
      <div>
        <textarea rows="10" class="chat-window"
          disabled>{{chatWindowText}}</textarea>
      </div>
    </template>
```

```
<script type="application/dart"
      src="chat_window.dart"></script>

</element>
</body></html>
```

That's a lot of code, so let me break it down.

Web Components are HTML documents. Hence, they start with:

```
<!DOCTYPE html>

<html><body>
```

Web Components enable you to define new HTML elements. Here is the code where we create the `x-chat-window` element:

```
<element name="x-chat-window"
      constructor="ChatWindowComponent" extends="div">
```

Notice that the element uses the Dart constructor `ChatWindowComponent`. Also note that all of the examples in this codelab use `extends="div"`.

The next part is the `<template>` for the Web Component. Inside the Web Component is:

```
<textarea rows="10" class="chat-window"
  disabled>{{chatWindowText}}</textarea>
```

Using MDV (Model Data Views), the `<textarea>` will automatically stay in sync with updates to `chatWindowText`. That makes keeping your user interface up to date a snap!

Each Web Component has corresponding behavior. In this case, we have a `<script>` tag that links to Dart code.

Create behavior for the custom element

Create a file named `web/chat_window.dart` with the following contents:

```
library chat_window;

import 'package:web_ui/web_ui.dart';

class ChatWindowComponent extends WebComponent {
  @observable
  String chatWindowText = '';

  displayMessage(String from, String msg) {
    _display("$from: $msg\n");
  }

  displayNotice(String notice) {
    _display("[system]: $notice\n");
  }

  _display(String str) {
    chatWindowText = "${chatWindowText}${str}";
  }
}
```

Each Web Component has a class:

```
class ChatWindowComponent extends WebComponent {
```

At this point, that class must always inherit from `WebComponent`.

Under the Hood: We expect to allow your custom element to extend the actual HTML element in the future.

The `ChatWindowComponent` class has some data:

```
@observable
String chatWindowText = new String();
```

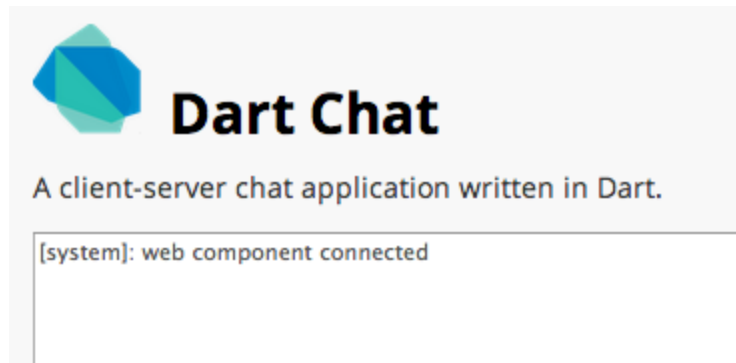
Updates to the `chatWindowText` object will automatically result in updates to the `<textarea>` as mentioned above.

Dart does not concatenate strings with the `+` operator, due to puzzlers made famous in other languages. There are numerous options for string concatenation, including string interpolation, which you see in action here: `chatWindowText = "${chatWindowText}${str}"`.

Building and running

Right-click on `build.dart` and select `Run`. Check the `build.dart` tab at the bottom of Dart Editor to make sure there weren't any errors. Also check the `Problems` tab to make sure you don't see any warnings.

Now, right-click on `web/out/index.html`, and select `Run in Dartium`. You should see:



If everything worked correctly, congratulations! You've just built your first Web Component! If you encountered any problems, now's a great time to check out the Troubleshooting section.

Step 4: More, more, more!!!

Objectives

In this step, we'll add two more Web Components, add a Web Sockets chat client, and finish the application.

Code

If you need to catch up, you can make a copy of `step03` named `mine` for this portion of the codelab.

Walkthrough

Add more components to `index.html`

Edit `web/index.html` (not the one in the `out` directory) and add the lines in bold. These lines correspond to the two new Web Components that we'll be creating.

...

```
<link rel="components" href="chat_window.html">
<link rel="components" href="username_input.html">
<link rel="components" href="message_input.html">
</head>
<body>
  <h1>Dart Chat</h1>

  <p>A client-server chat application written in Dart.</p>

  <x-chat-window id="chat-window"></x-chat-window>
  <x-username-input id="username-input"></x-username-input>
  <x-message-input id="message-input"></x-message-input>
```

...

Update application.dart

Now edit `web/application.dart` and add the lines in bold. Once again, we have to add some code for the two Web Components. We also have to add some code for the Chat client.

```
...
import 'package:web_ui/web_ui.dart';
import 'chat_connection.dart';
import 'chat_window.dart';
import 'username_input.dart';
import 'message_input.dart';

const connectionUrl = "ws://127.0.0.1:1337/ws";
ChatConnection chatConnection;
```

```

@observable ChatWindowComponent chatWindow;
@observable UsernameInputComponent usernameInput;
@observable MessageInputComponent messageInput;

Future init() {
  // The Web Components aren't ready immediately in
  // index.html's main.
  return new Future.delayed(0, () {

    // xtag is how you get to the Dart object.
    chatWindow = query("#chat-window").xtag;
    usernameInput = query("#username-input").xtag;
    messageInput = query("#message-input").xtag;

    chatWindow.displayNotice("web component connected");
    chatConnection = new ChatConnection(connectionUrl);
  }).catchError((e) => print("Problem initing app: $e"));
}

```

Create chat_connection.dart

Now create web/chat_connection.dart with the following code:

```

library chat_connection;

import 'dart:html';
import 'dart:json' as JSON;
import 'dart:async' show Timer;
import 'application.dart' as app;

class ChatConnection {

```

```

WebSocket websocket;
String url;

ChatConnection(this.url) {
  _init();
}

send(String from, String message) {
  var encoded = JSON.stringify({'f': from, 'm': message});
  _sendEncodedMessage(encoded);
}

_receivedEncodedMessage(String encodedMessage) {
  Map message = JSON.parse(encodedMessage);
  if (message['f'] != null) {
    app.chatWindow.displayMessage(message['f'], message['m']);
  }
}

_sendEncodedMessage(String encodedMessage) {
  if (websocket != null && websocket.readyState == WebSocket.OPEN) {
    websocket.send(encodedMessage);
  } else {
    print('WebSocket not connected, message $encodedMessage not sent');
  }
}

_init([int retrySeconds = 2]) {
  bool encounteredError = false;
  app.chatWindow.displayNotice("connecting to web socket");
  websocket = new WebSocket(url);
}

```

```

scheduleReconnect() {
  app.chatWindow.displayNotice('web socket closed, retrying in $retrySeconds seconds');
  if (!encounteredError) {
    new Timer(new Duration(seconds:retrySeconds), () => _init(retrySeconds * 2));
  }
  encounteredError = true;
}

websocket.onOpen.listen((e) => app.chatWindow.displayNotice('web socket connected'));
websocket.onClose.listen((e) => scheduleReconnect());
websocket.onError.listen((e) => scheduleReconnect());

websocket.onMessage.listen((MessageEvent e) {
  print('received message ${e.data}');
  _receivedEncodedMessage(e.data);
});
}
}

```

Whoa, long code is long! Let's break it down.

```
JSON.stringify({'f': from, 'm': message})
```

Encode Dart objects into JSON strings. JSON is a web-friendly text format for encoding and decoding structured data like maps, lists, numbers, booleans, strings, etc.

```
websocket.send(encodedMessage)
```

Send a message over a Web Socket, which is a bi-directional communication channel. A Web Socket can send text messages, and

low-level binary data.

```
scheduleReconnect()
```

This is an example of a *nested function*, a function defined inside another function. Thanks to lexical scoping, a nested function can access variables in its parent function.

```
new Timer(new Duration(seconds:retrySeconds)
```

A timer waits for a duration before it runs a callback function. Here we wait for `retrySeconds` before attempting to reconnect to the Web Socket server.

```
websocket.onOpen.listen((e) => ...
```

Event handling in Dart takes the form of `eventSource.onSomeEvent.listen((event) => doSomethingWithTheEvent(event))`. The `onSomeEvent` method returns a *stream* of events that you can listen to.

Create the username input Web Component

Create `web/username_input.html` with the following code:

```
<!DOCTYPE html>

<html><body>
  <element name="x-username-input" constructor="UsernameInputComponent" extends="div">
    <template>
      <div>
        <label for="username">Username:</label>
        <input id="username" name="username" type="text" bind-value="username">
      </div>
    </template>
  </element>
</body></html>
```

```
<script type="application/dart" src="username_input.dart"></script>
</element>
</body></html>
```

This Web Component is fairly similar to the first Web Component we saw. However, there are a couple differences.

Look at this line:

```
<input id="username" name="username" type="text" bind-value="username">
```

`bind-value="username"` means that every time the input's value is updated (by typing into the text field), the `username` field (in the `UsernameInputComponent` class) is automatically updated.

Now, create a new file named `web/username_input.dart` with the following code:

```
library username_input;

import 'package:web_ui/web_ui.dart';

class UsernameInputComponent extends WebComponent {
  @observable
  String username = "";
}
```

Create the message input Web Component

Create `web/message_input.html` with the following code:

```
<!DOCTYPE html>
```

```

<html><body>
  <element name="x-message-input" constructor="MessageInputComponent" extends="div">
    <template>
      <div>
        <label for="message">Message:</label>
        <input id="message" class="chat-message" type="text" bind-value="message">
        <button on-click="sendMessage() "
                  disabled="{{sendDisabled}}">Send</button>
      </div>
    </template>

    <script type="application/dart" src="message_input.dart"></script>
  </element>
</body></html>

```

This Web Component is very similar to the previous ones. However, there are a couple interesting parts.

```
on-click="sendMessage() "
```

This code declaratively states that when the button is clicked, run the `sendMessage()` method of the `MessageInputComponent` class.

Now, create a file named `web/message_input.dart` with the following contents:

```

library message_input;

import 'dart:html';
import 'package:web_ui/web_ui.dart';
import 'application.dart' as app;

```



```

class MessageInputComponent extends WebComponent {
  @observable
  String message = "";

  void sendMessage() {
    app.chatConnection.send(app.usernameInput.username, message);
    app.chatWindow.displayMessage(app.usernameInput.username, message);
    message = '';
  }

  bool get sendDisabled => (app.usernameInput == null ||
    app.usernameInput.username == null ||
    app.usernameInput.username.isEmpty ||
    app.messageInput == null ||
    app.messageInput.message == null ||
    app.messageInput.message.isEmpty);
}

```

Here's how the Web Component talks to the `chatConnection`, by way of the `application` library:

```
app.chatConnection.send(app.usernameInput.username, message);
```

Remember this line?

```
disabled="{{sendDisabled}}">
```

The `{{sendDisabled}}` is a data binding to the `sendDisabled` field. Dart supports real getters and setters, and so you'll notice that `sendDisabled` is a *getter*, and not a real field. The business logic reads, "If username *and* messageInput both have values, then enable the button. Otherwise, keep the button disabled.

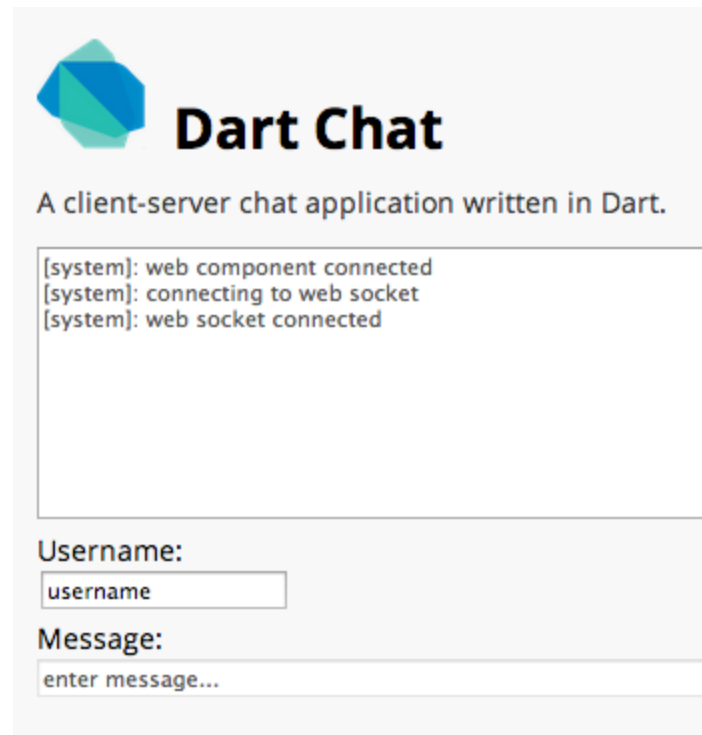
Building and running

First we need to fire up the server. Make sure you don't already have a copy of the server running, and then right-click on `bin/chat_server.dart` and select **Run**.

Next, just right-click on `web/index.html` and select **Run in Dartium**.

Under the Hood: Dart Editor maps the files that you edit to the files that `build.dart` generates. It knows that when you run `web/index.html`, that you really want to run `web/out/index.html`. How cool is that?

If all goes well, Dartium should appear, and you should see:



Copy-and-paste the URL into another tab, and see if you can send messages back-and-forth. If everything worked correctly, congratulations! You’ve just built your first full application using Web Components!

If you encountered any problems, now’s a great time to check out the Troubleshooting section.

What’s next?

If you made it this far, I hereby pronounce that you are awesome! Here is some Dart code to celebrate your awesomeness!

```
i.did(aWebApp, using: (dartWebUI & webComponents));  
assert(i.amAwesome());
```

I suggest you post it to Google+ ;) Just make sure you cc me ([+Shannon Behrens](#)) and tag it #dartlang!

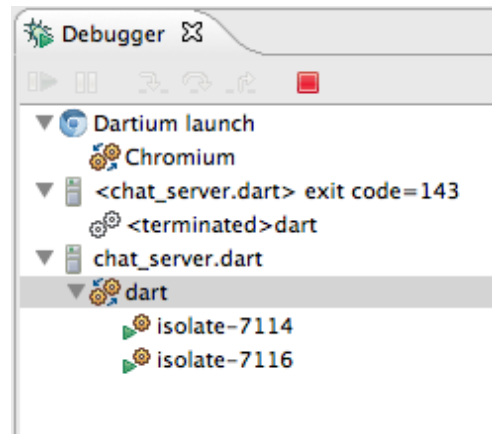
Troubleshooting

The “Dart Web UI” package is fairly new, and it’s changing rapidly. This codelab is also very new. Here are some things to try if you get stuck.

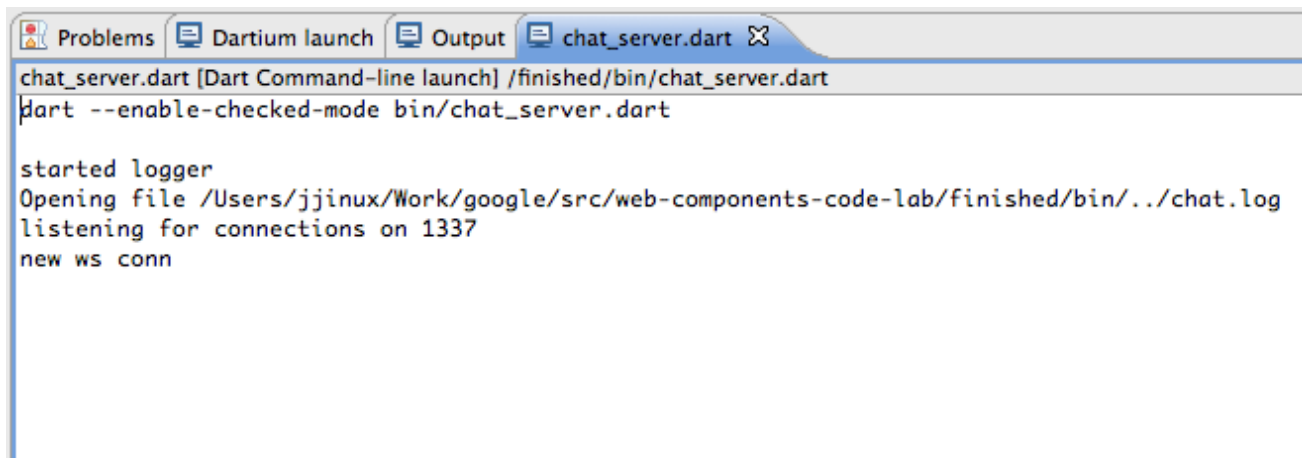
- Unlike the last codelab, each subdirectory of [web-ui-code-lab](#) is a separate pub package. That means you **should not** open up the whole project in Dart Editor. Instead, you should open up subdirectories one at a time. For instance, you should open `start_here`, `step02`, and `mine` separately.
- The Dart `web_components` package has [recently been renamed](#) `web_ui`. Keep that in mind if you see `web_components` mentioned in any older blog posts, etc. I’ve done my best to update this codelab.
- If you are working on (for instance) `step03` when you get stuck, look at the code in (for instance) `finished`.
- If Dart Editor can’t find a library that you installed via pub, try running `pub install` on the command line or “Tools / Pub

Install” in Dart Editor.

- If pub gets really confused (for instance, if you move your application’s directory), try deleting all the directories named `package` and run `pub install` again.
- If Dart Editor is complaining about something that you have already fixed, make sure all of your files are saved and then use “Tools > Reanalyze Sources”.
- If Dart Editor is still complaining about something you have already fixed, it sometimes helps to close the project entirely and open it up again.
- To rebuild all of your Web Components, right-click on `build.dart` in your project, and select “Run”. In OS X, you can just click on the file and hit Command-R.
- The quickest way to test out your code in OS X is to click on `web/index.html` and hit Command-R (to see the shortcut corresponding to your operating system, right click on `web/index.html`).
- **Make sure** you are viewing, editing, and running the right files:
 - Edit the files in the `web` directory, not the `web/out` directory. The ones in the `web/out` directory are autogenerated.
 - You can view the files in the `web/out` directory to see what your code has been compiled to.
- Remember to start `chat_server.dart`. Otherwise, your chat client won’t be able to connect to it.
- You can only run one version of `chat_server.dart` at a time. Make sure you click the red stop button in the debugger to terminate the existing version before starting another. Make sure you are running the version that you want to be running.



- There are a bunch of tabs at the bottom of Dart Editor. If you encounter problems, check those tabs. The Problems tab will tell you if there are problems in your code. **Warning:** at this point, Dart Editor will tell you if there are problems in the generated Dart code, not in the original Web Component. Hence, you'll need to edit the original file and rebuild. Also keep an eye on the other tabs since they will have output from Dartium, build.dart, and chat_server.dart.



- If you are querying for a Web Component by `id`, and you keep getting `null`, it could be because you are calling `query` in `main` before the Web Component has finished loading. Try adding:

```
import 'dart:async';
```

And then wrap your code with:

```
// The Web Components aren't ready immediately
// in index.html's main.
new Future.delayed(0, () {

    // xtag is how you get to the Dart object.
    var someComponent = query("#some-id").xtag;
    ...
});
```

This level of indirection will be going away hopefully very soon.

- Make sure the version of `dart-web-ui` that you are using matches the documentation you're reading. Look at the `pubspec.yaml` in this document to see which version this documentation was written for.
- The `dart-web-ui` library is always developed with a specific version of the SDK and Dart Editor in mind. You may get warnings if your version of the SDK or Dart Editor are much newer than your version of `dart-web-ui` or vice versa. The Dart platform is still moving very rapidly!
- Remember that there may be mistakes in the codelab itself. You get bonus points if you can spot them out and submit a bug on dartbug.com!
- If you get stuck while doing this codelab on your own, post your question to [Stack Overflow](https://stackoverflow.com) using the “dart” tag.