



DART

Dart Web Components Codelab

November 2012

Dart Web Components Codelab

November 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 in Dart. Along the way, you will learn:

- How to set up pub
- How to use Web Components
- How to use Model Driven Views
- How to build an application with multiple Web Components
- Where to get more information about Dart and Web Components
- 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 12 - Dart and Web Components](#) is an episode of Dartisans that we did with the Dart Web Components team.

Documentation

- [Dart Web Components](#) is the official documentation from the Dart Web Components team.
- [Tools for Dart Web Components](#) is the official documentation from the Dart Web Components team about the tools they have produced.
- [Your First Web Component with Dart](#) is a post on Seth Ladd's blog that I found particularly helpful.
- [Your First Model Driven View with Dart](#) is another post on Seth Ladd's blog 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

- [Dart Web Components](#) is the library I used to build this sample.
- [TodoMVC for Dart Web Components](#) is the code I referred to the most when writing this codelab.

What to do if you get stuck

See the [Troubleshooting](#) section. It's fairly extensive, so don't forget it's there!

Step 1: Import and run the chat app

This codelab walks you through building a custom chat application using Web Components. 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 finished version.

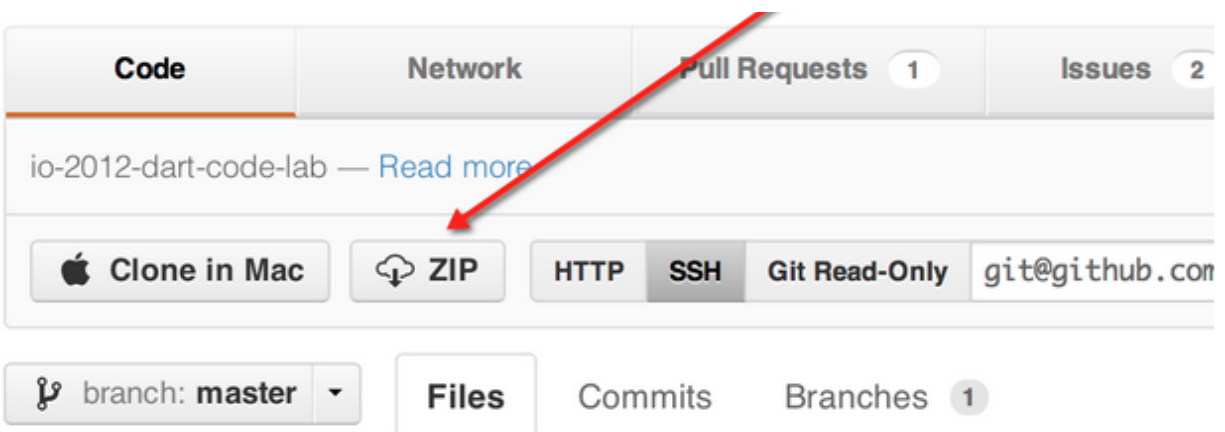
Walkthrough

Download the source code

You can find the source code for this codelab at:

<https://github.com/dart-lang/web-components-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 finished version into Dart Editor

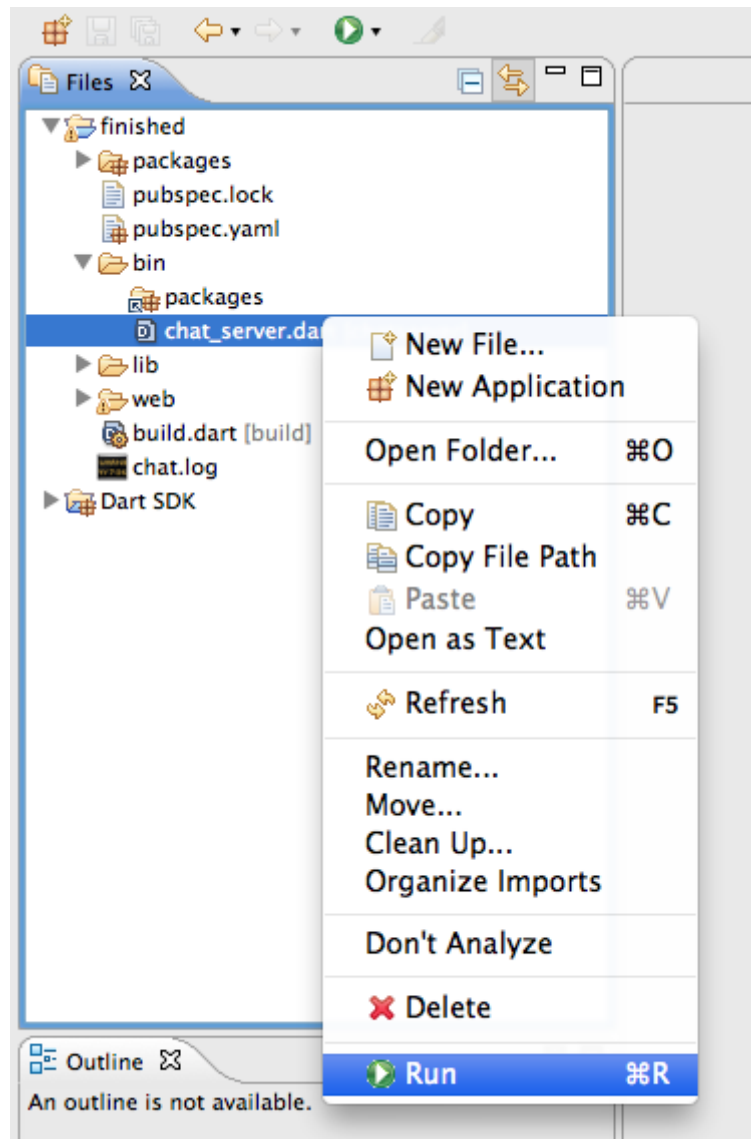
Do not open the entire codelab in Dart Editor. Each subdirectory is its own pub package, so each must be opened separately.

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

Launch the finished version of the server

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`.



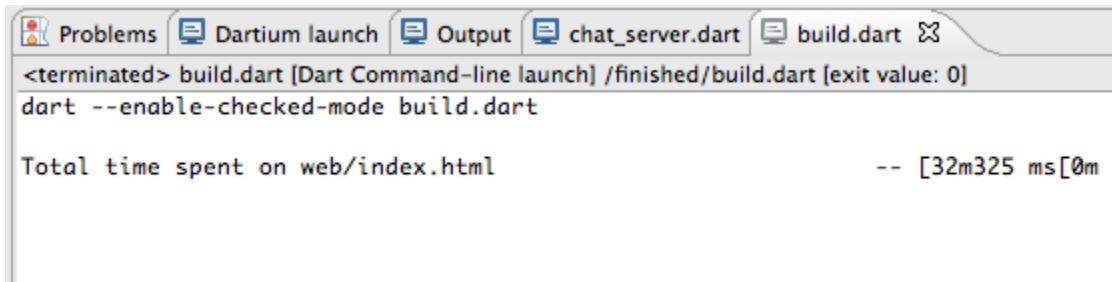
Verify the server is running by checking the `chat_server.dart` console output window at the bottom of your editor. You should see a message: "listening for connections on 1337".

```
Problems | Dartium launch | Output | chat_server.dart
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

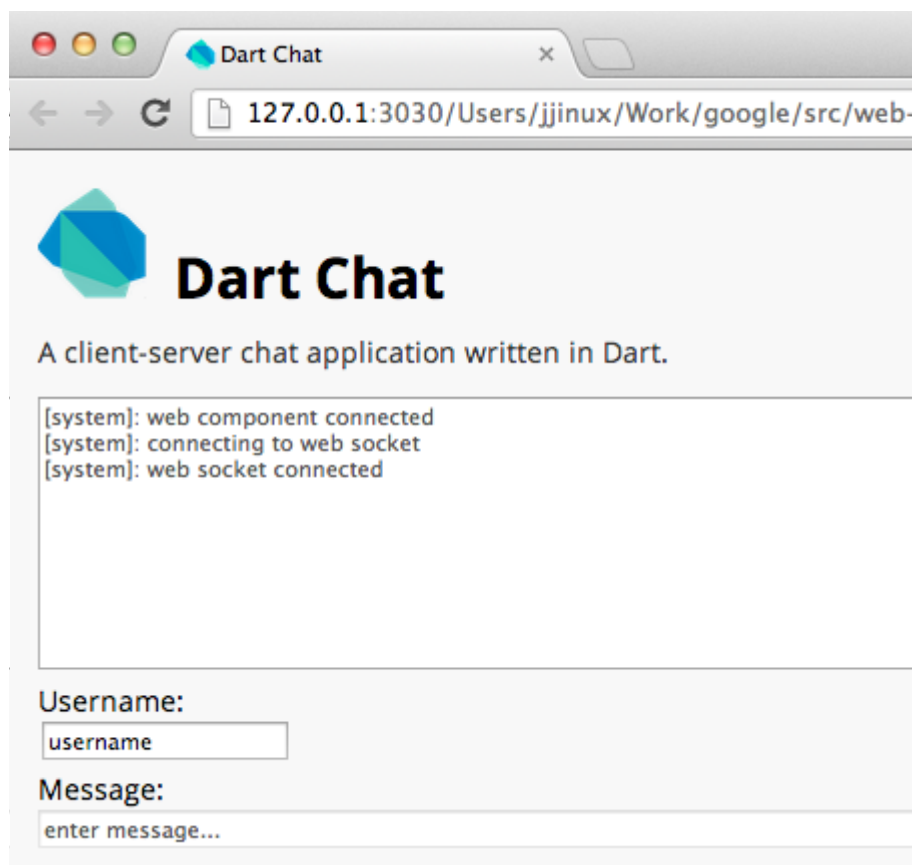
Although Dart does not need to be compiled to run in Dartium, Dart Web Components do have a build step in which the Web Component files are split into various Dart files. Right-click on `build.dart` in the `finished` directory and select `Run`. Look at the output of `build.dart` to check that it completed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

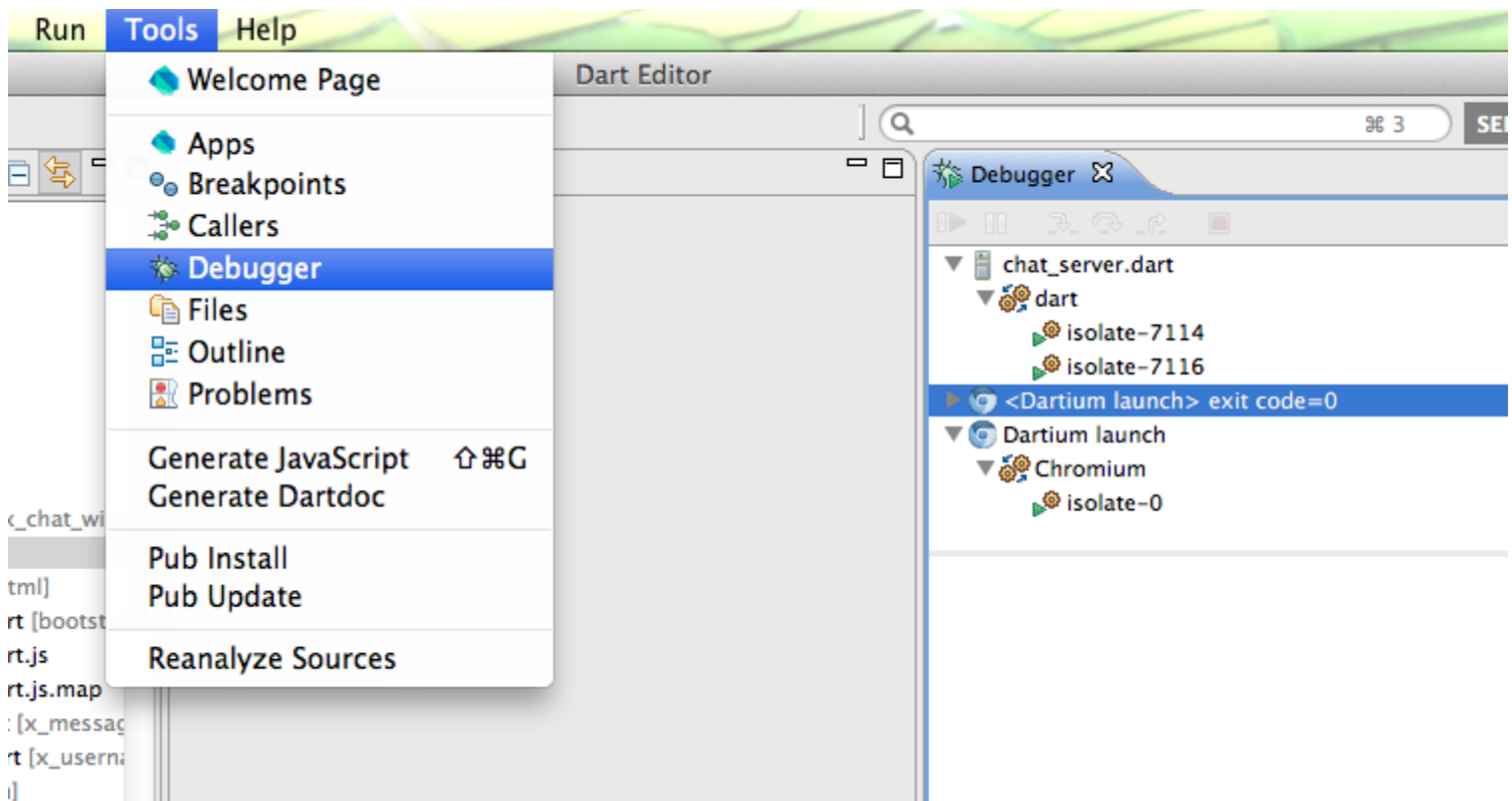
If things completed successfully, there should be a new directory named `web/out` containing the generated output. Right-click on `web/out/index.html` and select `Run`. If everything goes smoothly, Dartium should start, and you should see the client application.



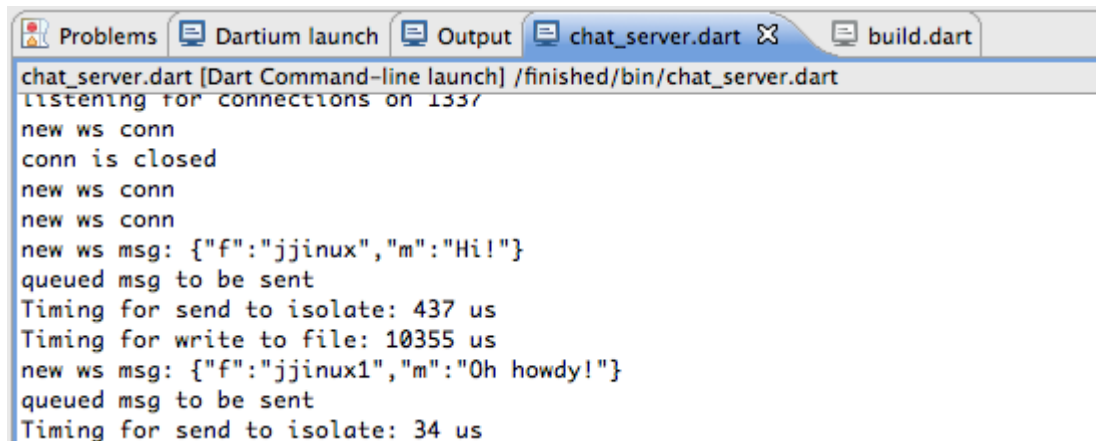
At this point, the client should be able to connect to the server. Type in a username and a message and hit enter. Copy the URL and try it in other browsers such as the normal version of Chrome or Safari.

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.

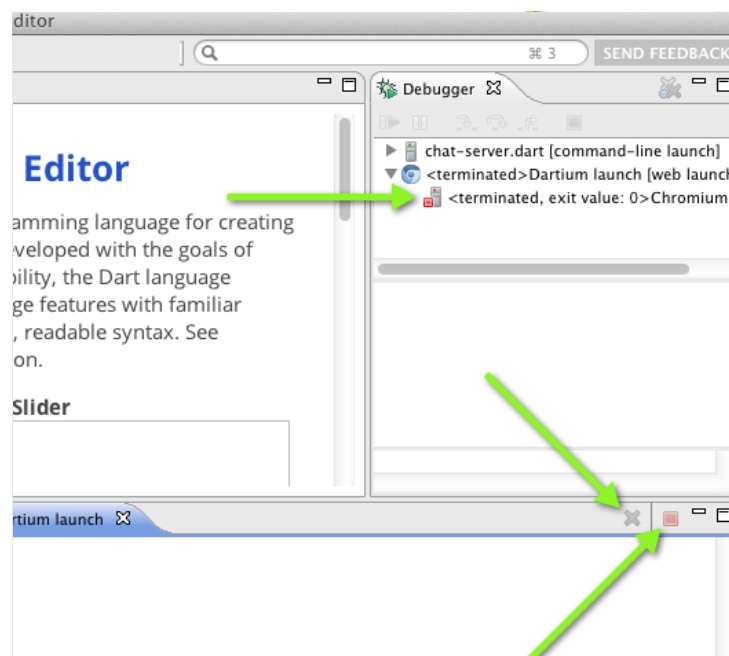


On the bottom of Dart Editor are two views, `chat_server.dart` and `Dartium launch`. Each view has the output from the respective process.



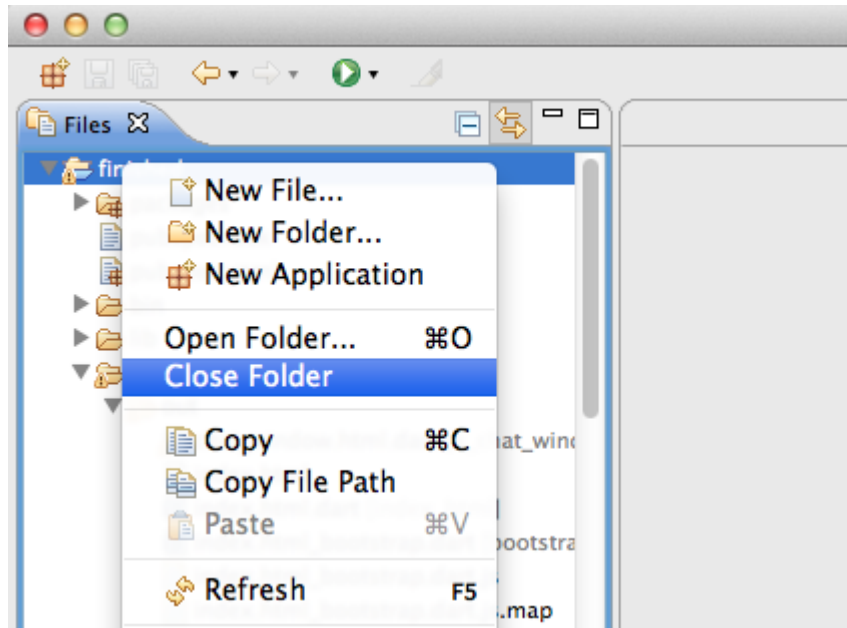
```
chat_server.dart [Dart Command-line launch] /finished/bin/chat_server.dart
listening for connections on 1337
new ws conn
conn is closed
new ws conn
new ws conn
new ws msg: {"f":"jjinux","m":"Hi!"}
queued msg to be sent
Timing for send to isolate: 437 us
Timing for write to file: 10355 us
new ws msg: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

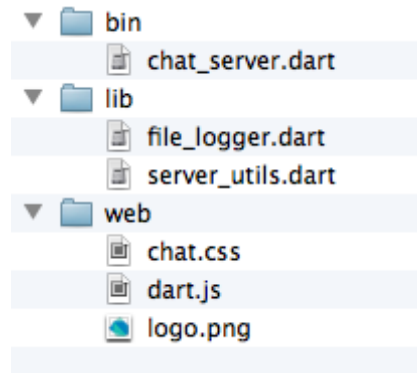
Objectives

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_components` library.

Walkthrough

In the `web-components-code-lab` directory, make a copy of the `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.



At this point, `chat_server.dart` should be complaining that it can't find libraries such as `package:dart_chat/file_logger.dart`. That problem will go away as soon as we setup pub.

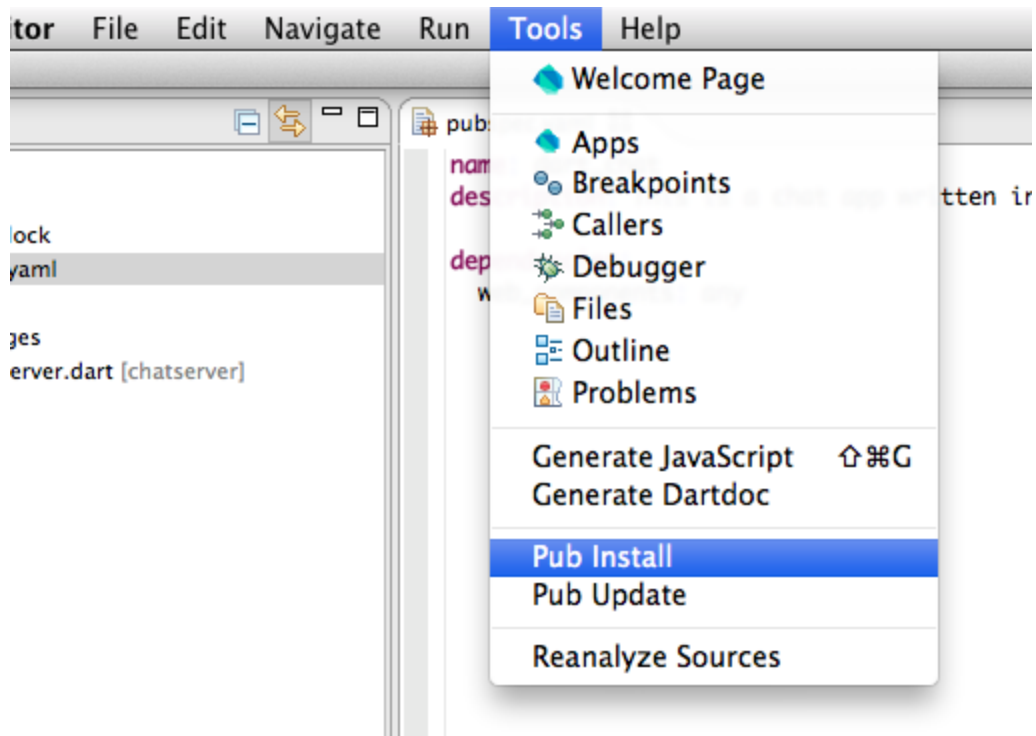
Right-click on the `mine` directory and select New File... Create a new file named `pubspec.yaml`. Put the following in the file:

```
name: dart_chat
description: This is a chat app written in Dart using Web
Components

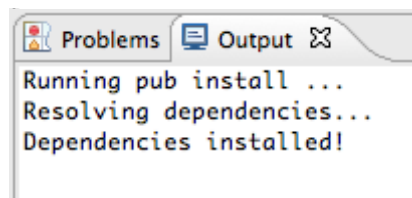
dependencies:
  web_components: ">=0.2.6 <0.2.7"
```

Notice that we're picking a fairly specific version of the `web_components` library to avoid breakages when backwards-incompatible changes are made.

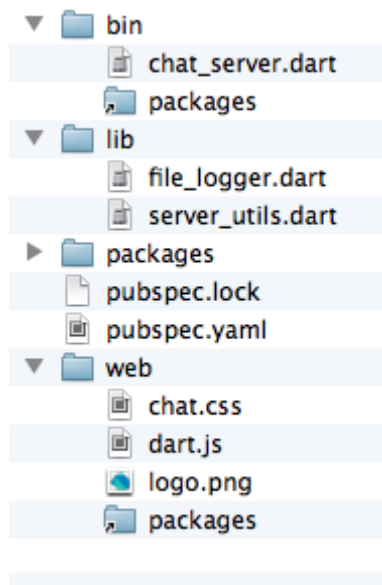
Save the file. Now, run Tools > Pub Install.



This will result in the following output:



It will also result in some new files and directories:



Step 3: Your first web component

In Dart, Web Components are stored in `.html` files. They contain both HTML markup as well as Dart code. A build step is used to translate these `.html` files into `.dart` files.

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`. Put the following in the file:

```
import 'package:web_components/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.

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();
      }
    </script>
    <script src="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">` is the code used to link in 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.

`<script src="dart.js"></script>` works with the JavaScript output of `dart2js` so that the application will run in browsers that don't natively support Dart.

Create `application.dart`

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

```
library application;

import 'dart:html';
import 'dart:isolate' show Timer;
import 'package:web_components/web_components.dart';
import 'out/chat_window.html.dart';

ChatWindowComponent chatWindow;

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

    // xtag is how you get to the Dart object.
    chatWindow = query("#chat-window").xtag;
    chatWindow.displayNotice("web component connected");

    dispatch();
  });
}
```

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:

```
Cannot find referenced source: out/chat_window.html.dart.
```

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

Create `chat_window.html`

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

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

      <script type="application/dart">
        import 'package:web_components/web_components.dart';

        class ChatWindowComponent extends WebComponent {
          StringBuffer chatWindowText = new StringBuffer();

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

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

          _display(String str) {
            chatWindowText.add(str);

            // You have to call dispatch whenever Web Component
            // data changes and it's not the result of a user
            // event. That happens a lot for this method.
            dispatch();
          }
        }
      </script>
    </element>
  </body></html>
```

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

Web Components are HTML documents. They are a mix of HTML and Dart code. Hence, they start with:

```
<html><body>
```


Web Components enable you to create 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.toString()}}</textarea>
```

Using MDV (Model Data Views), the `<textarea>` will automatically stay in sync with updates to `chatWindowText.toString()`. 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 with some Dart code.

Each Web Component has a class:

```
class ChatWindowComponent extends WebComponent {
```

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

The `ChatWindowComponent` class has some data:

```
  StringBuffer chatWindowText = new StringBuffer();
```

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

Next, the class has three very normal Dart methods:

```
  displayMessage(String from, String msg) {...}
  displayNotice(String notice) {...}
  _display(String str) {...}
```

The code calls `dispatch()` after any changes to `chatWindowText`. Generally, this isn't required as long as the modification happens as the result of a user event. However, in this

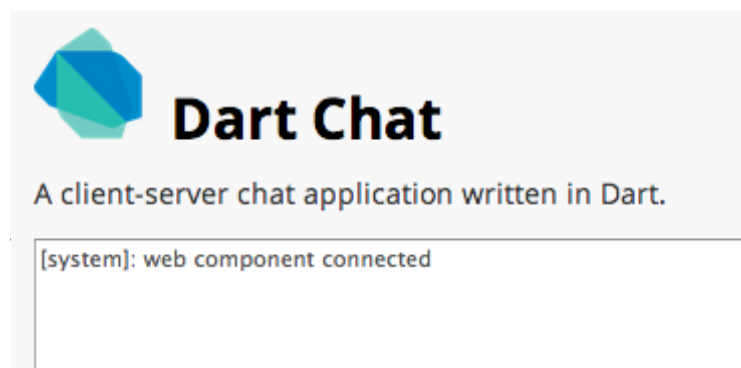
application, the `_display()` method also gets called on application startup as well as when a message is received on the Web Socket. Hence, it is necessary to call `dispatch()` explicitly.

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, aside from the following, which you can safely ignore:

```
Concrete class ChatWindowComponent has unimplemented member(s) # From
Node: int $dom_nodeType # From Element: Future<ElementRect> rect void
addHTML(String)
```

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

Update 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.

```
<!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">
    <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>

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

      main() {
        app.init();
      }
    </script>
    <script src="dart.js"></script>
  </body>
</html>
```

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. However, we also have to add some code for the Chat client.

```
library application;

import 'dart:html';
```

```

import 'dart:isolate' show Timer;
import 'package:web_components/web_components.dart';
import 'chat_connection.dart';
import 'out/chat_window.html.dart';
import 'out/username_input.html.dart';
import 'out/message_input.html.dart';

const connectionUrl = "ws://127.0.0.1:1337/ws";
ChatConnection chatConnection;
ChatWindowComponent chatWindow;
UsernameInputComponent usernameInput;
MessageInputComponent messageInput;

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

    // 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);

    dispatch();
  });
}

```

Create chat_connection.dart

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

```

library chat_connection;

import 'dart:html';
import 'dart:json';
import 'dart:isolate' 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(1000 * retrySeconds,
                (timer) => _init(retrySeconds * 2));
        }
        encounteredError = true;
    }

    webSocket.on.open.add((e) =>
        app.chatWindow.displayNotice('web socket connected'));
    webSocket.on.close.add((e) => scheduleReconnect());
}

```

```

        websocket.on.error.add((e) => scheduleReconnect());

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

```

This code is fairly similar to what we used in the first code lab. The only major difference is that the code makes use of `app.chatWindow.displayNotice()` to display messages to the user.

Create username_input.html

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

```

<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"
          data-bind="value:username"
          data-action="change:onUsernameChange">
      </div>
    </template>

    <script type="application/dart">
      import 'dart:html';
      import 'package:web_components/web_components.dart';
      import 'application.dart' as app;

      class UsernameInputComponent extends WebComponent {
        String username = "username";

        onUsernameChange(Event e) {
          if (!username.isEmpty) {
            app.messageInput.enable();
          } else {
            app.messageInput.disable();
          }
        }
      }
    </script>
  </element>
</body>
</html>

```

```

    </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"
      data-bind="value:username"
      data-action="change:onUsernameChange">

```

`data-bind="value:username"` means that every time the input is updated, the username field (in the `UsernameInputComponent` class) will automatically be updated.

`data-action="change:onUsernameChange"` means that every time the field is changed, the `onUsernameChange` method (in the `UsernameInputComponent` class) will be called. The syntax for this will soon be changing to `on-change="onUsernameChange () "`.

Notice the following code:

```
app.messageInput.enable();
```

This is an example of one Web Component talking to another Web Component (by way of the application library).

Create message_input.html

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

```

<html><body>
  <element name="x-message-input"
    constructor="MessageInputComponent" extends="div">
    <template>
      <div>
        <label for="message">Message:</label>
        <input id="message" class="chat-message"
          name="message" type="text" disabled
          data-bind="value:message"
          data-action="change:sendMessage">
      </div>
    </template>

    <script type="application/dart">

```

```

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

class MessageInputComponent extends WebComponent {
  String message = "enter message...";

  disable() {
    messageElement.disabled = true;
    message = 'Enter username';
  }

  enable() {
    messageElement.disabled = false;
    message = '';
  }

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

  InputElement _messageElement;
  get messageElement {
    if (_messageElement == null) {
      _messageElement = query("#message");
    }
    return _messageElement;
  }
}
</script>
</element>
</body></html>

```

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

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

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


This code is a getter called `messageElement`. It looks up `#message` and caches it in `_messageElement`:

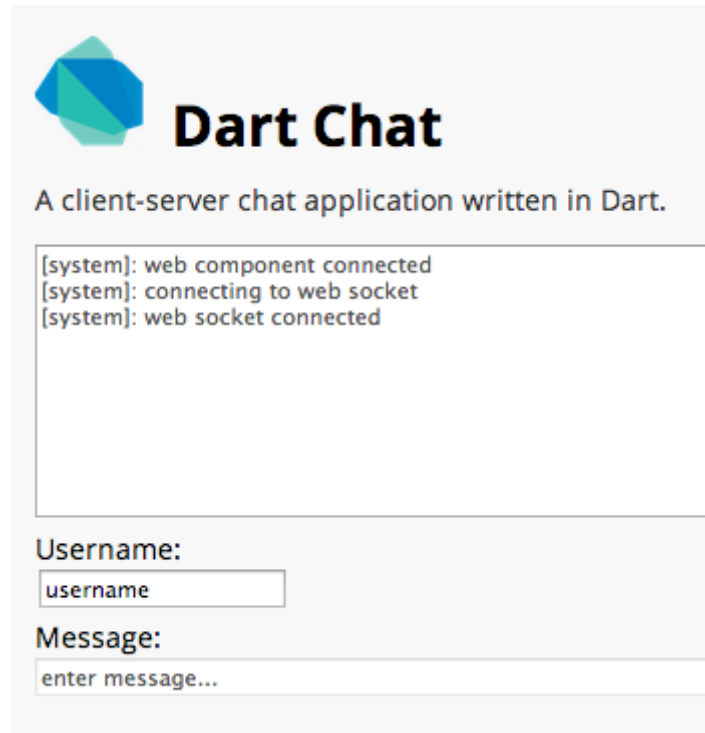
```
InputElement _messageElement;
get messageElement {
  if (_messageElement == null) {
    _messageElement = query("#message");
  }
  return _messageElement;
}
```

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**.

Since `web/out/index.html` already exists, it's even easier to build and run the client code than last time. Rather than running `build.dart` manually, you can just right-click on `web/out/index.html` and select **Run** in Dartium.

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



Copy-and-paste the URL into another browser, 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: (dart & webComponents));  
assert(i.amAwesome());
```

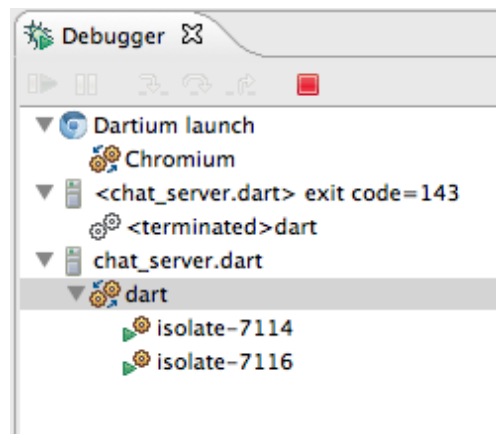
I suggest you post it to Google+ ;) Just make sure you cc me ([+Shannon Behrens](#)) and tag it #dartlang!

Troubleshooting

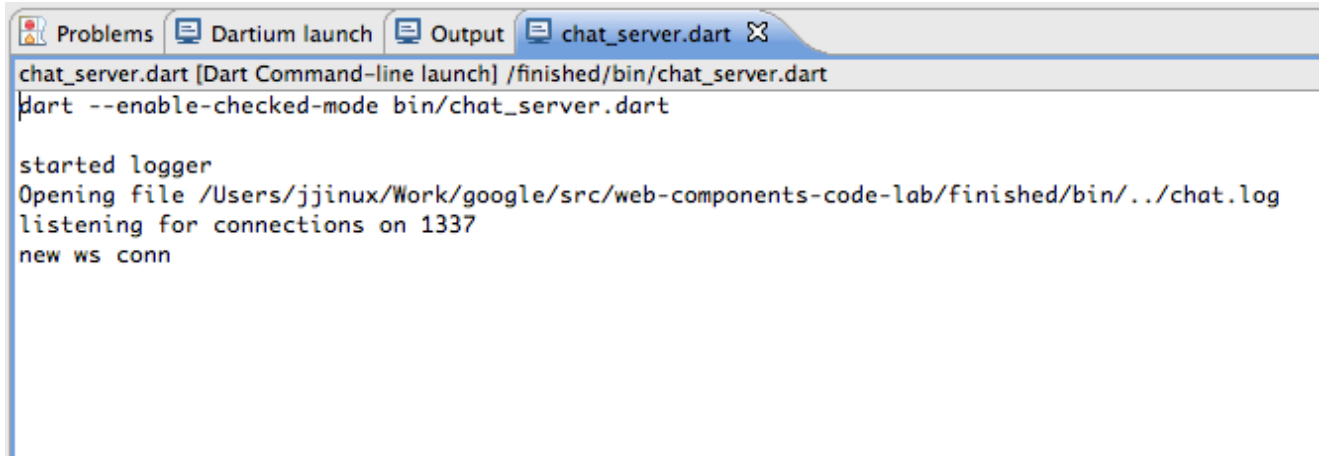
The “Dart Web Components” library is fairly new, and it’s changing rapidly. This codelab is also very new and has not received a lot of testing. Here are some things to try if you get stuck.

- Unlike the last codelab, each subdirectory of [web-components-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`, `step01`, and `finished` separately. It’ll probably be a lot less confusing if you only open one at a time.
- If you are working on (for instance) `step03` when you get stuck, look at the code in (for instance) `step04`. [step04](#) is currently the last step, so it’s the most likely to be 100% correct.
- If Dart Editor can’t find a library that you installed via pub, try running `pub install` on the command line and/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 `out/index.html` and hit Command-r (to see the shortcut corresponding to your operating system, right click on “out/index.html”). In most cases, this will automatically run `build.dart`.
- **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.
 - Run the files in the web/out directory, not the web directory. You can’t run these files directly until you have used the Web Component tools to compile them.
 - 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 see a message the following warning in the Problems tab, just ignore it. This will be fixed soon:

```
Concrete class ChatWindowComponent has unimplemented member(s) #  
From Node: int $dom_nodeType # From Element: Future<ElementRect>  
rect void addHTML(String)
```

- If you modifying the data for a Web Component, and the user interface doesn't update itself, it could be because you're modifying the data at a time when the Web Components library doesn't expect it (i.e. not as the direct result of a user event). Try adding:

```
import 'package:web_components/web_components.dart';
```

And then call the following after you are done modifying the data:

```
dispatch();
```

- 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:isolate' show Timer;
```

And then wrap your code with:

```
// The Web Components aren't ready immediately  
// in index.html's main.  
new Timer(0, (timer) {  
  
    // xtag is how you get to the Dart object.
```

```
var someComponent = query("#some-id").xtag;  
...  
});
```

This level of indirection will be going away very soon.

- The syntax for Dart Web Components is not completely set in stone. At some point, code such as `data-action="change:foo"` will be changing to `on-change="foo()"`.
- Remember that there may be mistakes in the codelab itself. Bonus points if you can spot them out!
- If you get stuck while doing this codelab on your own, post your question to [Stack Overflow](#) using the “dart” tag.